Федеральное агентство связи

Сибирский государственный университет телекоммуникаций и информатики

Кафедра прикладной математики и кибернетики

Курсовой проект

«Структуры и алгоритмы обработки данных»

Вариант 23

Выполнил: студент группы ИП-714

Баклан Н. А.

Проверил: ассистент кафедры ПМиК

Турцев А.А.

Новосибирск 2018

Содержание

[1. ПОСТАНОВКА ЗАДАЧИ 3](#_Toc533143399)

[2. ОСНОВНЫЕ ИДЕИ И ХАРАКТЕРИСТИКИ ПРИМЕНЯЕМЫХ МЕТОДОВ 4](#_Toc533143400)

[2.1. МЕТОД СОРТИРОВКИ 4](#_Toc533143401)

[2.2 ДВОИЧНЫЙ ПОИСК 4](#_Toc533143402)

[2.3 ДЕРЕВО И ПОИСК ПО ДЕРЕВУ 5](#_Toc533143403)

[3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ АЛГОРИТМОВ 7](#_Toc533143404)

[4. ОПИСАНИЕ ПРОГРАММЫ 8](#_Toc533143405)

[4.1. ОСНОВНЫЕ ПЕРЕМЕННЫЕ И СТРУКТУРЫ 8](#_Toc533143406)

[4.2. ОПИСАНИЕ ПОДПРОГРАММ 8](#_Toc533143407)

[5. ТЕКСТ ПРОГРАММЫ 11](#_Toc533143408)

[6. РЕЗУЛЬТАТЫ 22](#_Toc533143409)

[7. ВЫВОДЫ 25](#_Toc533143410)

1. ПОСТАНОВКА ЗАДАЧИ

Хранящуюся в файле базу данных загрузить в оперативную память компьютера и построить индексный массив, упорядочивающий данные по дате поселения и названию улицы, используя **метод прямого слияния** в качестве метода сортировки.

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

Из записей очереди построить **ДБД-дерево по номеру дома**, и предусмотреть возможность поиска в дереве по запросу.

Закодировать файл базы данных статическим **кодом Гильерта-Мура**, предварительно оценив вероятности всех встречающихся в ней символов. Построенный код вывести на экран.

База данных "Населенный пункт"

Стpуктуpа записи:

ФИО гражданина: текстовое поле 32 символа

фоpмат <Фамилия>\_<Имя>\_<Отчество>

Название улицы: текстовое поле 18 символов

Номер дома: целое число

Номер квартиры: целое число

Дата поселения: текстовое поле 10 символов

фоpмат дд-мм-гг

Пpимеp записи из БД:

Петpов\_Иван\_Федоpович\_\_\_\_\_\_\_\_\_\_\_

Ленина\_\_\_\_\_\_\_\_\_\_\_\_\_\_

10

67

29-02-65

по дате поселения и названию улицы, К = год поселения.

2. ОСНОВНЫЕ ИДЕИ И ХАРАКТЕРИСТИКИ ПРИМЕНЯЕМЫХ МЕТОДОВ

2.1. МЕТОД СОРТИРОВКИ

Метод прямого слияния

В основе метода прямого слияния лежит операция слияния серий. р-серией называется упорядоченная последовательность из р элементов.

Пусть имеются две упорядоченные серии a и b длины q и r соответственно. Необходимо получить упорядоченную последовательность с, которая состоит из элементов серий a и b. Сначала сравниваем первые элементы последовательностей a и b. Минимальный элемент перемещаем в последовательность с. Повторяем действия до тех пор, пока одна из последовательностей a и b не станет пустой, оставшиеся элементы из другой последовательности переносим в последовательность с. В результате получим (q+r)-серию.

Для алгоритма слияния серий с длинами q и r необходимое количество сравнений и перемещений оценивается следующим образом

min(q, r) ≤ C ≤ q+r-1, M=q+r

Пусть длина списка S равна степени двойки, т.е. 2k, для некоторого натурального k. Разобьем последовательность S на два списка a и b, записывая поочередно элементы S в списки а и b. Сливаем списки a и b с образованием двойных серий, то есть одиночные элементы сливаются в упорядоченные пары, которые записываются попеременно в очереди c0 и c1. Переписываем очередь c0 в список a, очередь c1 – в список b. Вновь сливаем a и b с образованием серий длины 4 и т. д. На каждом итерации размер серий увеличивается вдвое. Сортировка заканчивается, когда длина серии превысит общее количество элементов в обоих списках. Если длина списка S не является степенью двойки, то некоторые серии в процессе сортировки могут быть короче.

Трудоёмкость метода прямого слияния определяется сложностью операции слияния серий. На каждой итерации происходит ровно n перемещений элементов списка и не более n сравнений. Как нетрудно видеть, количество итераций равно .Тогда

Дополнительные n перемещений происходят во время начального расщепления исходного списка. Асимптотические оценки для М и С имеют следующий вид

Метод обеспечивает устойчивую сортировку. При реализации для массивов, метод требует наличия второго вспомогательного массива, равного по размеру исходному массиву. При реализации со списками дополнительной памяти не требуется.

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:

Выбранный элемент равен X. Поиск завершён.

Выбранный элемент меньше X. Продолжаем поиск в правой половине массива.

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

Из-за необходимости найти все элементы соответствующие заданному ключу поиска в курсовой работе использовалась вторая версия двоичного поиска, которая из необходимых элементов находит самый левый, в результате чего для поиска остальных требуется просматривать лишь оставшуюся правую часть массива.

Верхняя оценка трудоёмкости алгоритма двоичного поиска такова. На каждой итерации поиска необходимо два сравнение для первой версии, одно сравнение для второй версии. Количество итераций не больше, чем . Таким образом, трудоёмкость двоичного поиска в обоих случаях



2.3 ДЕРЕВО И ПОИСК ПО ДЕРЕВУ

Двоичное Б-дерево

Двоичное Б-дерево состоит из вершин (страниц) с одним или двумя элементами. Следовательно, каждая страница содержит две или три ссылки на поддеревья. На рисунке показаны примеры страниц Б – дерева при m = 1.

Поэтому вновь рассмотрим задачу построения деревьев поиска в оперативной памяти компьютера. В этом случае неэффективным с точки зрения экономии памяти будет представление элементов внутри страницы в виде массива. Выход из положения – динамическое размещение на основе списочной структуры, когда внутри страницы существует список из одного или двух элементов.

Таким образом, страницы Б-дерева теряют свою целостность и элементы списков начинают играть роль вершин в двоичном дереве. Однако остается необходимость делать различия между ссылками на потомков (вертикальными) и ссылками на одном уровне (горизонтальными), а также следить, чтобы все листья были на одном уровне.

Очевидно, двоичные Б-деревья представляют собой альтернативу АВЛ-деревьям. При этом поиск в двоичном Б-дереве происходит как в обычном двоичном дереве.

Высота двоичного Б-дерева

.

Если рассматривать двоичное Б-дерево как обычное двоичное дерево, то его высота может увеличиться вдвое, т.е. . Для сравнения, в АВЛ-дереве даже в самом плохом случае h<1.44 log n. Поэтому сложность поиска в двоичном Б-дереве и в АВЛ-дереве одинакова по порядку величины.

При построении двоичного Б-дерева реже приходится переставлять вершины, поэтому АВЛ-деревья предпочтительней в тех случаях, когда поиск ключей происходит значительно чаще, чем добавление новых элементов. Кроме того, существует зависимость от особенностей реализации, поэтому вопрос о применение того или иного тапа деревьев следует решать индивидуально для каждого конкретного случая.

2.4 МЕТОД КОДИРОВАНИЯ

Алфавитный код Гилберта – Мура

Рассмотрим источник с алфавитом А={*a1,a2,…,an*} и вероятностями *p1,…pn*. Пусть символы алфавита некоторым образом упорядочены, например, *a1≤a2≤…≤an*. *Алфавитным* называется код, в котором кодовые слова лексико-графически упорядочены, т.е. *φ(a1)≤φ(a2)≤…≤φ(an).*

Е.Н. Гилбертом и Э.Ф. Муром предложили метод построения алфавитного кода, для которого *Lср < H+2.* Процесс построения происходит следующим образом.

1. Составим суммы *Qi, i=1,n*, вычисленные следующим образом:

*Q1=p1/2, Q2=p1+p2/2, Q3=p1+p2+p3/2,…, Qn=p1+p2+…+pn-1+pn/2*.

1. Представим суммы *Qi* в двоичном виде.
2. В качестве кодовых слов возьмем -log2*pi* +1 младших бит в двоичном представлении *Qi*.

**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.

Таблица 1 Код *Гилберта-Мура*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *ai* | *Pi* | *Qi* | *Li* | кодовое слово |
| *a1*  *a2*  *a3*  *a4*  *a5*  *a6* | 1/23≤0.18  1/23≤0.18<1/22  1/22≤0.36<1/21  1/24≤0.07  1/24≤0.09  1/24≤0.12 | 0.09  0.27  0.54  0.755  0.835  0.94 | 4  4  3  5  5  5 | 0001  0100  100  11000  11010  11110 |

Средняя длина кодового слова не превышает значения энтропии плюс 2

*Lср*=4**.**0.18+4**.**0.18+3**.**0.36+5**.**0.07+5**.**0.09+5**.**0.12=3.92<2.37+2

3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ АЛГОРИТМОВ

В ходе выполнения курсовой работы, помимо основных алгоритмов, потребовалось реализовать также несколько вспомогательных, необходимых для корректной работы программы.

1. Интерфейс программы

Для организации интерфейса использовалась процедура void mainloop(), которая обеспечивает корректное и незатруднительное использование программы и предоставляет возможность многократного выбора различных вариантов обработки базы данных, в зависимости от задач пользователя.

2. Загрузка и вывод базы данных

Для загрузки базы данных разработана процедура Node \*load\_to\_memory(), в которой производится считывание записей типа struct Record, а из них формируется очередь Node record. Здесь же предусмотрена проверка на наличие файла, откуда выполняется считывание. Данная процедура вызывается независимо от желания пользователя, в то время как остальные он может выбрать посредствам меню.

После загрузки в динамическую память выполняется построение индексного массива по списку.

За вывод элементов считанной базы данных отвечает процедура void show\_list Она предоставляет возможность постраничного просмотра базы данных(по 20 элементов на странице), смена страниц осуществляется вводом «w» и «s». Есть возможность прервать просмотр при помощи ввода любой другой клавиши.

3. Вспомогательные функции и процедуры для сортировки данных

При сортировке базы данных потребовалось реализовать дополнительную функцию сравнения двух строк (int strcomp()), которая возвращает -1, если вторая строка больше, 1, если первая строка больше, 0, если они равны.

4. Особенности реализации бинарного поиска

Для того чтобы без проблем многократно осуществлять поиск элементов, соответствующих разным ключам создаётся индексный массив при помощи функции viod make index array, в котором можно икать множество раз.

5. Вспомогательные функции для построения ДБД-дерева

Также как и для очереди, при неоднократном построении дерева требуется освобождать память, эту функцию выполняет процедура void rmtree. Для вывода дерева на экран используется процедура void Print\_tree, представляющая собой обход дерева слева – направо.

Аналогичная процедура void search\_in\_tree выполняет вывод результатов поиска в дереве.

6. Кодирование данных

При побуквенном кодировании существует необходимость знать вероятности встречаемости символов. Для их подсчета создана процедура void calc\_probabilities(), в которой происходит вычисление вероятностей на основе частоты встречающихся символов. Для создания кодовых слов используется функция void mur().

4. ОПИСАНИЕ ПРОГРАММЫ

4.1. ОСНОВНЫЕ ПЕРЕМЕННЫЕ И СТРУКТУРЫ

int N = 4000;

struct Record {  
 char author[12];  
 char title[32];  
 char publisher[16];  
 short int year;  
 short int num\_of\_page;  
};

Запись, используемая для работы с базой данных «Запись».

int [N] –глобальная переменная для массива на N элементов.

struct Node {  
 Record record;  
 Node \*next;  
};

Структура (список), используемая при сортировке базы данных.

Node \*next **–** указатель на следующие элемент;

Record record **–** поле данных

struct Vertex { // структура дерева  
 Record \*data;  
 Vertex \*left;  
 Vertex \*right;  
};

Структура, представляющая дерево оптимального поиска (А2).

Vertex \*left, Vertex \*right – указатели на левое и правое поддеревья.

Record \*data - поле данных (адрес элемента в основном массиве структур).

Record records[N] – массив структур

Record \*arr[] – индексный массив.

Int \*w – массив весов.

Record \*ind\_arr[] индексный массив для построения дерева.

Record \*unsorted\_ind\_arr[N] – массив указателей на не отсортированные записи

4.2. ОПИСАНИЕ ПОДПРОГРАММ

Функция вывода меню:

1. void mainloop();

Функция отображает меню и принимает ввод пользователя, после чего запускает подпрограмму, зависящую от того, что ввёл пользователь.

Процедуры начальной обработки базы данных:

1. Node \*load\_to\_memory();
2. make\_index\_array(Record \*arr[], Node \*root, int n = N);

Load\_to\_memory – считывание базы из файла и представление ее элементов в форме вышеперечисленных структур, возвращает указатель на массив записей.

Make\_index\_array - создание массива указателей на список.

arr - индексный массив, в который будут записаны указатели на каждую из вершин. Root - список, состоящий из записей, соединённых между собой указателями.

Функции и процедуры сортировки:

1. **void quick\_sort**(Record \*array[], int n)
2. **int strcomp**(const string &str1, const string &str2, int len);

**Quick\_sort** – Функция, сортирующая список методом Хоара.

Array - сортируемый массив. Int n - количество элементов в массиве.

**strcomp** – определение операции сравнения двух строк (str1 и str2): размерностью len. Возвращает 1 в случае, если a > b и -1, если a < b и 0, если а = b.

Функции и процедуры для поиска в отсортированной базе данных:

1. **int quick\_search**(Record \*arr[], const string &key);
2. void search(Record \*arr[], int &ind, int &n);

**Quick\_search** – процедура двоичного поиска (версия 2), Record \*arr[] – указатель на массив записей, в котором осуществляется поиск, **key** – ключ поиска. Возвращает позицию найденного элемента и -1, в случае его отсутствия.

**search** – Фунцкия, вызываемая из основного меню. Запрашивает у пользователя ключ поиска и с ним вызывает функцию **Quick\_search.** Arr - отсортированный индексный массив. Ind - индекс первого вхождения ключа, n - количество вхождений.

Процедуры и функции построения АВЛ-дерева:

1. void rmtree(Vertex \*root);
2. void dbd\_add(Vertex \*&p, Record \*data);
3. void printTree(Vertex \*p);
4. void Search\_In\_Tree(Vertex \*root, int key);

**rmtree** – освобождение памяти для построения дерева, чтобы не возникало проблем, в случае если до этого дерево уже создавалось (**root** – указатель на корень дерева).

**dbd\_add**– непосредственно построение, p - указатель на корень дерева. data - добавляемые данные.

**PrintTree** – обход дерева с корнем **p**, используемый для вывода на экран отсортированных по году элементов базы данных.

**Search\_In\_Tree** – поиск в дереве с корнем **root** элементов, соответствующих ключу key. Выводит все найденные ключи.

Процедуры и функции кодирования базы данных:

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

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

22. void **coding**()

23. void mur(const int n, double p[], int Length[], char c[][20])

**Get\_char\_counts\_from\_file** – функция, подсчитывающая сколько раз каждый символ входит в базу данных, а также размер файла. File\_name - название файла, file\_size - размер файла, n - количество записей.

**calc\_probabilities** – Функция, подсчитывающая вероятности вхождения каждого символа. Возвращает вектор, содержащий пару: символ и количество его вхождений. Counter\_map - хеш таблица, ключом в которой является символ, а значением - сколько раз этот символ входит в БД.

**coding** – Основная функция для кодировки. Вызывается из главного меню.

**mur -** Функция, кодирующая символы методом Шеннона, согласно их вероятностям. **N** - количество различных символов, p - массив, содержащий вероятности этих символов, **Length** - массив, содержащий длины кодовых слов, char c[][20] - массив, содержащий коды символов.

Основная программа

int **main**() – в основной программе вызывается только меню.

5. ТЕКСТ ПРОГРАММЫ

#include <iostream>

#include <iomanip>

#include <fstream>

#include <unordered\_map>

#include <vector>

#include <cmath>

#include <algorithm>

using namespace std;

const int N = 4000;

struct Record {

char fio[32];

char street[18];

short int home;

short int appartament;

char date[10];

};

struct Node {

Record record;

Node \*next;

};

string prompt(const string &str) {

cout << str;

cout << "\n> ";

string ans;

cin >> ans;

return ans;

}

int strcomp(const string &str1, const string &str2, int len = -1) {

if (len == -1) {

len = (int) str1.length();

}

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

if (str1[i] == '\0' and str2[i] == '\0') {

return 0;

} else if (str1[i] == ' ' and str2[i] != ' ') {

return -1;

} else if (str1[i] != ' ' and str2[i] == ' ') {

return 1;

} else if (str1[i] < str2[i]) {

return -1;

} else if (str1[i] > str2[i]) {

return 1;

}

}

return 0;

}

int compare(const Record &record1, const Record &record2) {

int date\_comparate\_arr[] = {6, 7, 3, 4, 0, 1};

for (int i : date\_comparate\_arr) {

if (record1.date[i] > record2.date[i]) {

return -1;

} else if (record1.date[i] < record2.date[i]) {

return 1;

}

}

if (strcomp(record1.street, record2.street) > 0) {

return -1;

} else if (strcomp(record1.street, record2.street) < 0) {

return 1;

}

return 0;

}

struct body {

Node \*head;

Node \*tail;

};

void MergeSort(Node \*&S, int n) {

int t, q, r, i, m;

Node \*a, \*b, \*k, \*p;

body c[2];

a = S;

b = S->next;

k = a;

p = b;

int x = 0;

while (p != NULL) {

k->next = p->next;

k = p;

p = p->next;

}

t = 1;

Node \*temp1 = a, \*temp2 = b;

while (temp1 != NULL) {

temp1 = temp1->next;

}

while (temp2 != NULL) {

temp2 = temp2->next;

}

while (t < n) {

c[0].tail = c[0].head = NULL;

c[1].tail = c[1].head = NULL;

i = 0;

m = n;

while (m > 0) {

if (m >= t) q = t;

else q = m;

m = m - q;

if (m >= t) r = t;

else r = m;

m = m - r;

while (q != 0 && r != 0) {

if (compare(a->record, b->record) > 0) {

if (c[i].tail == NULL) {

c[i].tail = c[i].head = a;

} else {

c[i].tail->next = a;

c[i].tail = a;

}

a = a->next;

q--;

} else {

if (c[i].tail == NULL) {

c[i].tail = c[i].head = b;

} else {

c[i].tail->next = b;

c[i].tail = b;

}

b = b->next;

r--;

}

}

while (q > 0) {

if (c[i].tail == NULL) {

c[i].tail = c[i].head = a;

} else {

c[i].tail->next = a;

c[i].tail = a;

}

a = a->next;

q--;

}

while (r > 0) {

if (c[i].tail == NULL) {

c[i].tail = c[i].head = b;

} else {

c[i].tail->next = b;

c[i].tail = b;

}

b = b->next;

r--;

}

i = 1 - i;

x++;

}

a = c[0].head;

b = c[1].head;

t = 2 \* t;

}

c[0].tail->next = NULL;

S = c[0].head;

}

Node \*load\_to\_memory() {

Node \*root = NULL;

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

if (not file.is\_open()) {

return NULL;

}

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

Record record;

file.read((char \*) &record, sizeof(Record));

root = new Node{record, root};

}

file.close();

return root;

}

void print\_head() {

cout << "Record Full Name Street Home Apt Date\n";

}

void print\_record(Record \*record, int i) {

cout << "[" << setw(4) << i << "] "

<< record->fio

<< " " << record->street

<< " " << record->home

<< " " << setw(3) << record->appartament

<< " " << record->date << "\n";

}

void show\_list(Record \*records[], int n = N) {

int ind = 0;

while (true) {

print\_head();

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

Record \*record = records[ind + i];

print\_record(record, ind + i + 1);

}

string chose = prompt("w: Next page\t"

"q: Last page\t"

"e: Skip 10 next pages\n"

"s: Prev page\t"

"a: First page\t"

"d: Skip 10 prev pages\n"

"Any key: Exit");

switch (chose[0]) {

case 'w':

ind += 20;

break;

case 's':

ind -= 20;

break;

case 'a':

ind = 0;

break;

case 'q':

ind = n - 20;

break;

case 'd':

ind -= 200;

break;

case 'e':

ind += 200;

break;

default:

return;

}

if (ind < 0) {

ind = 0;

} else if (ind > n - 20) {

ind = n - 20;

}

}

}

int quick\_search(Record \*arr[], const std::string &key) {

int l = 0;

int r = N - 1;

while (l < r) {

int m = (l + r) / 2;

if (strcomp(&arr[m]->date[6], key, 2) < 0) {

l = m + 1;

} else {

r = m;

}

}

if (strcomp(&arr[r]->date[6], key, 2) == 0) {

return r;

}

return -1;

}

void make\_index\_array(Record \*arr[], Node \*root, int n = N) {

Node \*p = root;

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

arr[i] = &(p->record);

p = p->next;

}

}

struct Vertex {

Record \*data;

Vertex \*left;

Vertex \*right;

int balance;

};

void dbd\_add(Record \*data, Vertex \*&p) {

static int vr = 1;

static int hr = 1;

if (!p) {

p = new Vertex{data, NULL, NULL, 0};

vr = 1;

} else if (data->home < p->data->home) {

dbd\_add(data, p->left);

if (vr == 1) {

if (p->balance == 0) {

Vertex \*q = p->left;

p->left = q->right;

q->right = p;

p = q;

q->balance = 1;

vr = 0;

hr = 1;

} else {

p->balance = 0;

vr = 1;

hr = 0;

}

} else {

hr = 0;

}

} else if (data->home >= p->data->home) {

dbd\_add(data, p->right);

if (vr == 1) {

p->balance = 1;

hr = 1;

vr = 0;

} else if (hr == 1) {

if (p->balance == 1) {

Vertex \*q = p->right;

p->balance = 0;

q->balance = 0;

p->right = q->left;

q->left = p;

p = q;

vr = 1;

hr = 0;

} else {

hr = 0;

}

}

}

}

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

if (p) {

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

print\_record(p->data, i++);

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

}

}

void search\_in\_tree(Vertex \*root, int key, int &i) {

if (root) {

if (key < root->data->home) {

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

} else if (key > root->data->home) {

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

} else if (key == root->data->home) {

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

print\_record(root->data, i++);

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

}

}

}

void rmtree(Vertex \*root) {

if (root) {

rmtree(root->right);

rmtree(root->left);

delete root;

}

}

void tree(Record \*arr[], int n) {

Vertex \*root = NULL;

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

dbd\_add(arr[i], root);

}

print\_head();

int i = 1;

Print\_tree(root, i);

int key;

do {

key = stoi(prompt("Input search key (home), 0 - exit"));

print\_head();

i = 1;

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

} while (key != 0);

rmtree(root);

}

void search(Record \*arr[], int &ind, int &n) {

string key;

do {

key = prompt("Input search key (year)");

} while (key.length() != 2);

ind = quick\_search(arr, key);

if (ind == -1) {

cout << "Not found\n";

} else {

int i;

for (i = ind + 1; strcomp(&arr[i]->date[6], key, 2) == 0; ++i) {}

n = i - ind;

show\_list(&arr[ind], n);

}

}

int Med(int L, int R, double \*p) {

double SL = 0;

for (int i = L; i <= R; i++) {

SL = SL + p[i];

}

double SR = p[R];

int m = R;

while (SL >= SR) {

m--;

SL = SL - p[m];

SR = SR + p[m];

}

return m;

}

void Purrrr(const int n, double p[], int Length[], char c[][20]) {

double \*q = new double[n];

Length[0] = -floor(log2(p[0])) + 1;

q[0] = p[0] / 2;

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

double tmp = 0;

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

tmp += p[k];

q[i] = tmp + p[i] / 2;

Length[i] = -floor(log2(p[i])) + 1;

}

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

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

q[i] \*= 2;

c[i][j] = floor(q[i]) + '0';

if (q[i] >= 1) {

q[i] -= 1;

}

}

}

}

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("Could not open file");

}

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<>());

cout << "Probabil. char\n";

for (auto i : probabilities) {

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

}

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

auto c = new char[n][20];

auto Length = new int[n];

auto p = new double[n];

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

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

}

double shen = 0;

Purrrr(n, p, Length, c);

cout << "\nMur Code:\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';

shen += p[i];

}

cout << "Average length = " << avg\_len << '\n'

<< "Entropy = " << entropy << '\n'

<< "Average length < entropy + 2\n"

<< "N = " << n << endl;

}

void mainloop(Record \*unsorted\_ind\_array[], Record \*sorted\_ind\_array[]) {

int search\_ind, search\_n = -1;

while (true) {

std::string chose = prompt("1: Show unsorted list\n"

"2: Show sorted list\n"

"3: Search\n"

"4: Tree\n"

"5: Coding\n"

"Any key: Exit");

switch (chose[0]) {

case '1':

show\_list(unsorted\_ind\_array);

break;

case '2':

show\_list(sorted\_ind\_array);

break;

case '3':

search(sorted\_ind\_array, search\_ind, search\_n);

break;

case '4':

if (search\_n == -1) {

std::cout << "Please search first\n";

} else {

tree(&sorted\_ind\_array[search\_ind], search\_n);

}

break;

case '5':

coding();

break;

default:

return;

}

}

}

int main() {

Node \*root = load\_to\_memory();

if (!root) {

cout << "File not found" << endl;

return 1;

}

Record \*unsorted\_ind\_arr[N];

Record \*sorted\_ind\_arr[N];

make\_index\_array(unsorted\_ind\_arr, root);

MergeSort(root, N);

make\_index\_array(sorted\_ind\_arr, root);

mainloop(unsorted\_ind\_arr, sorted\_ind\_arr);

}

6. РЕЗУЛЬТАТЫ

Рисунок 1. Неотсортированная база данных.

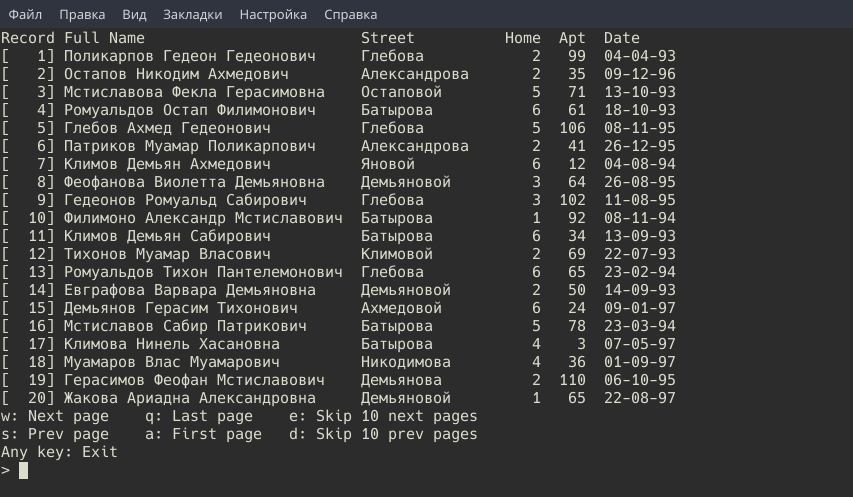


Рисунок 2. Отсортированная по году и автору база данных.

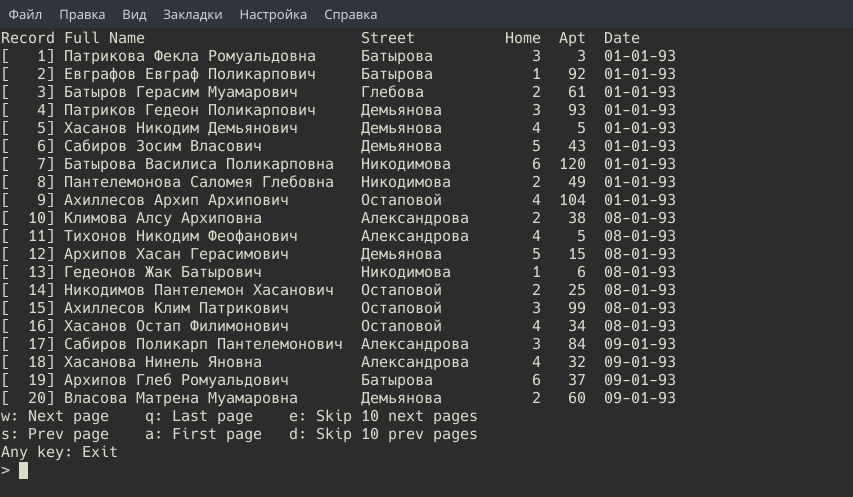


Рисунок 3. Очередь из записей, полученных в результате поиска (95).

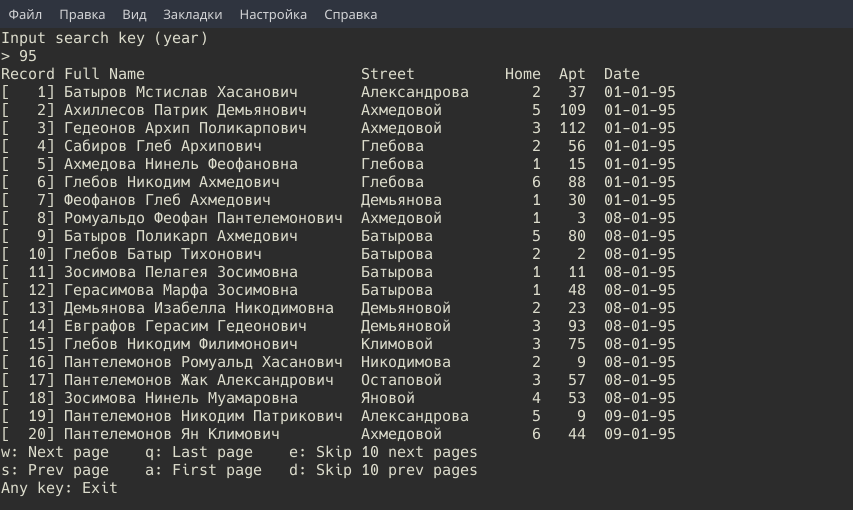


Рисунок 4. Дерево, ключ в дереве – дом.

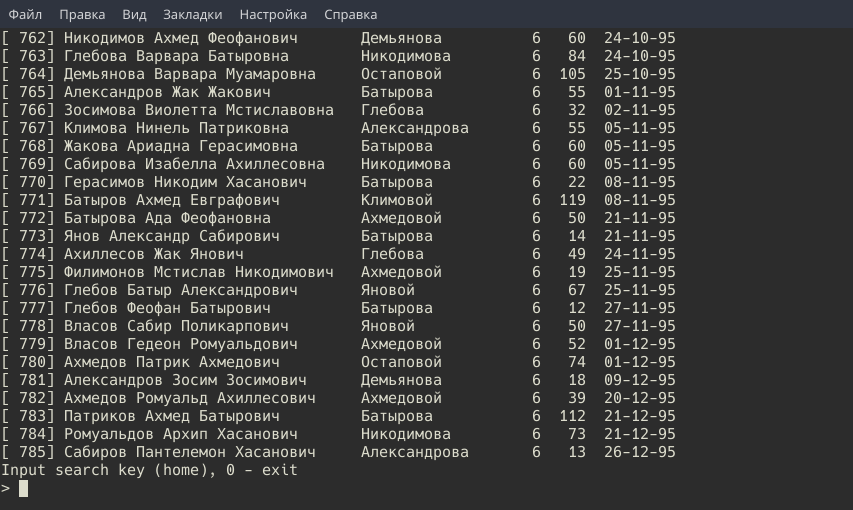


Рисунок 5. Поиск по дереву (4).

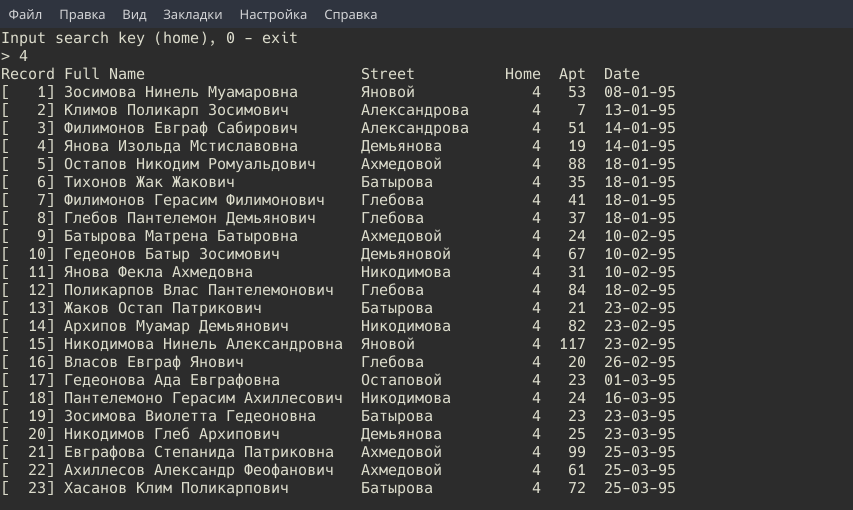
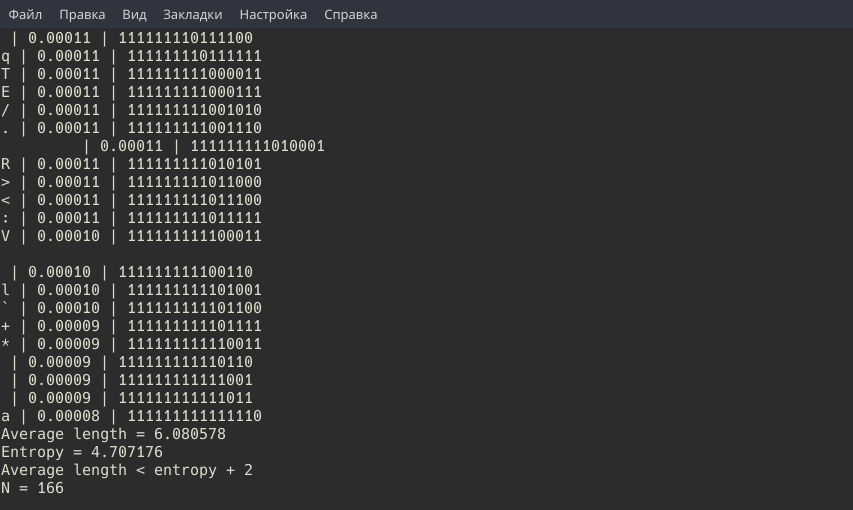


Рисунок 6. Примеры кодовых слов, энтропия и средняя длина кодового слова.



7. ВЫВОДЫ

В ходе выполнения курсовой работы были выполнены все поставленные задачи и реализованы необходимые алгоритмы: сортировки, поиска, построения дерева, поиска по дереву и кодирование базы данных.

В результате кодирования были получены данные подтверждающие теоретические сведения. К таковым относятся: величины средней длины кодового слова и энтропии (Lср ≤ H + 2).

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

Реализованные алгоритмы представляют минимальный набор функций для представления и обработки базы данных, а также отличаются достаточно высоким быстродействием и эффективностью.