Федеральное агентство связи

Сибирский государственный университет телекоммуникаций и информатики

Кафедра прикладной математики и кибернетики

Курсовой проект

«Структуры и алгоритмы обработки данных»

Вариант

Выполнил: студент группы

Проверил: ассистент кафедры ПМиК

Новосибирск 2021

Оглавление

[1. Постановка задачи 2](#_Toc533614154)

[2. Основные идеи и характеристики применяемых методов 3](#_Toc533614155)

[2.1. Метод сортировки 3](#_Toc533614156)

[2.2. Двоичный поиск 3](#_Toc533614157)

[2.3. Дерево и поиск по нему 4](#_Toc533614158)

[2.4. Метод кодирования 4](#_Toc533614159)

[3. Особенности реализации алгоритмов 6](#_Toc533614160)

[4. Описание программы 8](#_Toc533614161)

[4.1. Основные переменные и структуры 8](#_Toc533614162)

[4.2. Описание подпрограмм 9](#_Toc533614163)

[5. Текст программы 12](#_Toc533614164)

[6. Результаты 25](#_Toc533614165)

[7. Выводы 29](#_Toc533614166)

# Постановка задачи

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

Упорядочить данные с помощью цифровой сортировки (Digital sort), построить по отсортированным данным индексный массив. 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 ключу (приближенный алгоритм А2) по должности, вывести на экран содержимое дерева и предусмотреть возможность поиска в дереве по запросу.

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

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

Ваpианты условий упоpядочения и ключи поиска (К):

по дате(!) pождения, К = год pождения. Ключ в дереве = должность

# Основные идеи и характеристики применяемых методов

## 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.

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.

## 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 f_02. Таким образом, трудоёмкость двоичного поиска в обоих случаях

f_03

## Дерево и поиск по нему

ДОП А2

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).

Второй приближённый алгоритм (А2) использует предварительно упорядоченный набор вершин. В качестве корня выбирается такая вершина, что разность весов левого и правого поддеревьев была минимальна. Для этого путем последовательного суммирования весов определим вершину Vk, для которой справедливы неравенства:

 и .

Тогда в качестве "центра тяжести" может быть выбрана вершина Vk, Vk-1 или Vk+1, т. е. вершина, для которой разность весов левого и правого поддерева минимальна. Далее действия повторяются для каждого поддерева.

## Метод кодирования

Алфавитный код Гилберта – Мура

Е. Н. Гилбертом и Э. Ф. Муром был предложен метод построения алфавитного кода, для которого *.*

Пусть символы алфавита некоторым образом упорядочены, например, *a*1*≤a*2*≤…≤an*. Код  называется *алфавитным*, если кодовые слова лексикографически упорядочены, т.е. *.*

Процесс построения кода происходит следующим образом.

1. Вычислим величины *Qi, i=*1*,n*:

*Q*1*=p*1*/*2*,*

*Q*2*=p*1*+p2/*2*,*

*Q*3*=p*1*+p*2*+p*3*/*2*,*

…

*Qn=p*1*+p*2*+…+pn-*1*+pn/*2.

1. Представим суммы *Qi* в двоичном виде.
2. В качестве кодовых слов возьмем  младших бит в двоичном представлении *Qi*, .

**Пример**. Пусть дан алфавит A={*a*1*, a*2*, a*3*, a*4*, a*5*, a*6} с вероятностями *p*1=0.36, *p*2=0.18, *p*3=0.18, p4=0.12, *p*5=0.09, *p*6=0.07. Построенный код приведен в таблице 8.

Таблица 8 Код *Гилберта-Мура*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *ai* | *Pi* | *Qi* | *Li* | кодовое слово |
| *a*1  *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. 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

# 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*

Интерфейс программы реализован в бесконечном цикле с возможностью выхода из него, при нажатии клавиши «Esc». Выбор пункта меню реализован с помощью функции switch( ).

1. *Загрузка и вывод базы данных*

База данных открывается в int main( ). Считывается база данных в список структур в функции void Read\_base(FILE \*fp, list \*base), 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 DigitalSort(list \*&head). Она сортирует список сначала по полю должости, а потом по полю даты. 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).

1. *Особенности реализации бинарного поиска*

Бинарный поиск по отсортированной базе данных осуществляется в процедуре void BSearch(list \*\*A,char year[2]). Результатом работы процедуры void BSearch(list \*\*A,char year[2]) 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.

1. *Вспомогательные функции и процедуры для построения дерева оптимального поиска (приближенный алгоритм А2).*

Построение дерева осуществляется в процедуре void A2(int L, int R, list \*\*mas). Записи заносятся в дерево в процедуре void add\_vertex(vertex \*&p, list \*\*mas, int w). Присутствует процедура и для удаления дерева void delete\_tree(vertex \*&p). Обход дерева слева направо осуществляется в процедуре void LRprint(vertex \*x). Поиск по дереву выполняется в процедуре void TreeSearch(vertex \*p, char \*data).

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.

# Описание программы

## Основные переменные и структуры

struct record

{

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;

struct list {

record \*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

## Описание подпрограмм

Процедуры начальной обработки базы данных:

1. void Read\_base(FILE \*fp, list \*base) – считывает базу данных и создает список.
2. Void Print(list \*base); - визуальный вывод списка на консоль.

Процедуры сортировки:

1. void copy\_base(list \*a, list \*b) – делает копию списка, в котором хранится база данных.
2. void DigitalSort(list \*&head) – сортирует базу данных по дате и должностям.

Процедуры и функции для поиска в отсортированной базе данных:

1. void BSearch(list \*\*A, char year[2]) – бинарный поиск по ключу года.

Процедуры построения дерева оптимального поиска (А2):

1. void add\_vertex(vertex \*&point, list \*mas) – добавление элемента в дерево.
2. void A2(int L, int R, list \*\*mas) – построение дерева оптимального поиска, приближенный алгоритм А2.
3. void LRprint(vertex \*x) – вывод дерева.
4. void TreeSearch(vertex \*p, char \*data) – поиск в дереве.
5. void delete\_tree(vertex \*&p) – удаление дерева.

Процедуры и функции кодирования базы данных:

1. void GilbertMoorCode() – считывание символов базы данных, подсчет их вероятностей и преобразование алфавита, создание кодовых слов.
2. void CodePrint() – вывод статистики и алфавита с вероятностями.

Основная программа:

1. main( ) - основная программа, в которой выводится меню, а также в зависимости от выбранного пункта меню вызываются соответствующие процедуры и функции.

# Текст программы

# #include <fstream>

# #include <iostream>

# #include <conio.h>

# #include <Windows.h>

# #include <iomanip>

# #include <cstdio>

# #include <cstring>

# #include <cstdlib>

# #include <cmath>

# #include <iomanip>

# #include <algorithm>

# using namespace std;

# const int N=4000;

# const int M=256;

# const int alphabet\_num=81;

# int sum=0;

# int code[M][M];

# float entropy = 0, lgm = 0;

# int \*W;

# struct record

# {

# char FIO[30];

# short int Department;

# char Position[22];

# char Date[10];

# };

# struct list

# {

# record \*data;

# list\* prev;

# list\* next;

# };

# struct queue {

# list \*head;

# list \*tail;

# };

# struct vertex

# {

# record \*data;

# vertex \*next;

# int weight;

# vertex \*left;

# vertex \*right;

# };

# vertex \*root = NULL;

# struct GM\_code {

# float p;

# float q;

# int l;

# char a;

# };

# GM\_code A[M];

# GM\_code B[alphabet\_num];

# void Read\_base(FILE \*fp, list \*base);

# void copy\_base(list\*, list\*);

# void Print(list \*base);

# void DigitalSort (list \*&head);

# int BSearch (list \*\*A, char year[2]);

# void A2(int L, int R, list \*\*mas);

# void add\_vertex(vertex \*&p, list\* mas, int w);

# void LRprint(vertex\*);

# void TreeSearch (vertex\* p, char \*data);

# void weight(list \*mas[N]);

# void delete\_tree(vertex \*&p);

# int size(vertex\*);

# int height(vertex\*);

# int max(int, int);

# int sdp(vertex\*, int);

# void GilbertMoorCode();

# void CodePrint();

# int main()

# {

# setlocale(LC\_ALL, "Russian");

# SetConsoleCP(1251);

# FILE \*fp;

# W = new int[N];

# int enter = 0;

# 

# fp = fopen("testBase2.dat", "rb");

# list \*base = new list;

# list \*base\_sort= new list;

# list \*base\_index =new list;

# Read\_base(fp,base);

# fclose(fp);

# 

# copy\_base(base, base\_sort);

# DigitalSort(base\_sort);

# copy\_base(base\_sort, base\_index);

# list \*mas[N];

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

# mas[i]=base\_index;

# base\_index=base\_index->next;

# }

# GilbertMoorCode();

# while (true) {

# system("CLS");

# enter = 0;

# char \*spc = " ";

# int search\_start=0;

# int search=N-1;

# 

# SetConsoleCP(866);

# cout << "\t\t\t\tМЕНЮ" << endl;

# cout << "\t1. Посмотреть БД" << endl;

# cout << "\t2. Просмотр отсортированной БД (Digital sort по дате рождения и должности)" << endl;

# cout << "\t3. Поиск в отсортированной БД по ключу (год рождения)" << endl;

# cout << "\t4. Поиск в дереве по ключу (должность)" << endl;

# cout << "\t5. Результаты кодирования" << endl;

# cout << "\tEsc. Выход " << endl;

# cout << endl;

# while ((enter != 27) && (enter != 49) && (enter != 50) && (enter != 51) && (enter != 52) && (enter != 53)) {

# if (\_kbhit()) {

# enter = \_getch();

# }

# }

# switch (enter) {

# case 27: return 0;

# case 49:

# Print(base);

# \_getch();

# break;

# case 50:

# Print(base\_sort);

# \_getch();

# break;

# case 51:

# cout <<endl;

# cout << "Enter year"<<endl;

# char year[2];

# cin >> year;

# search=BSearch(mas,year);

# if(search == -1){

# cout << "No data" << endl;

# }

# else{

# SetConsoleCP(1251);

# do{

# if(search==0){

# break;

# }

# else{

# search--;

# }

# 

# if(strncmp(&mas[search]->data->Date[6],year,2) != 0){

# search++;

# break;

# }

# }

# while(true);

# 

# search\_start=search;

# 

# do{

# search++;

# if(search==N){

# search--;

# break;

# }

# if(strncmp(&mas[search]->data->Date[6],year,2) != 0){

# break;

# }

# }

# while(true);

# 

# int\* mass\_random = new int[N];

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

# mass\_random[i] = rand() % 100;

# }

# sort(mass\_random, mass\_random + N - 1, greater<int>());

# if (root != NULL) {

# delete\_tree(root);

# root = NULL;

# }

# cout <<endl <<endl<<"Founded "<<search-search\_start <<" pozitions ("<< search\_start<< " "<< search-1<<")"<<endl;

# for(int i=search\_start;i<search;i++){

# cout<<i<<" "<<mas[i]->data->FIO<<" "<<mas[i]->data->Department<<"\t"<<mas[i]->data->Position<<" "<<mas[i]->data->Date<<endl;

# W[i] = rand() % 99 + 1;

# }

# A2(search\_start, search,mas);

# }

# \_getch();

# break;

# 

# case 52:

# cout << endl << "LR-print - должность"<<endl;

# LRprint(root);

# cout<<endl;

# printf("+------+----------+----------------+----------+----------------+\n");

# printf("|%6d| Size | Height | Mid. height |\n", N);

# printf("+------+----------+----------------+----------+----------------+\n");

# printf("| A2 |%10d|%10d|%16.2f|\n", size(root), height(root), (double)sdp(root, 1) / size(root));

# SetConsoleCP(866);

# cout<<endl<< "Поиск в дереве: "<< endl;

# char j[22];

# cin.sync();

# cin.getline(j,22);

# SetConsoleCP(1251);

# for(int i = strlen(j); i<21;i++){

# strcat(j, spc);

# }

# if(strcmp(j,"0")!=0){

# TreeSearch(root,j);

# }

# \_getch();

# break;

# case 53:

# CodePrint();

# \_getch();

# break;

# }

# }

# system ("Pause");

# return 0;

# }

# void CodePrint(){

# lgm=0;

# SetConsoleCP(866);

# printf("\n\nКод Гилберта-Мура: \n\n");

# printf("| № | Symbol | Propability | Code word | Length |\n");

# SetConsoleCP(1251);

# for (int i = 0; i < alphabet\_num; i++)

# {

# printf("| %2d | %c | %2.6f | ",i, B[i].a, B[i].q);

# for (int j = 1; j <= B[i].l; j++)

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

# for (int j = B[i].l + 1; j < 14; j++)

# printf(" ");

# printf(" | %4d |\n", B[i].l);

# lgm += B[i].p \* B[i].l;

# }

# SetConsoleCP(866);

# 

# printf("\n Энтропия \t Средняя длина\n");

# printf(" %10f %10.5f \n", entropy, lgm);

# 

# cout << endl << endl << entropy+2 <<" > "<< lgm <<endl <<endl;

# 

# }

# void GilbertMoorCode(){

# int i,j;

# FILE \*fp;

# fp = fopen("testBase2.dat", "rb");

# for (i = 0; i < M; i++) {

# A[i].p = 0;

# A[i].q = 0;

# A[i].l = 0;

# A[i].a = (char)(i-128);

# }

# while (!feof(fp)) {

# char c;

# fscanf(fp, "%c", &c);

# if (feof(fp))

# break;

# A[c+128].p +=1;

# A[c+128].a = c;

# sum++;

# }

# fclose(fp);

# for (i = 0, j = 0; i < M; i++){

# if(A[i].p!=0){

# A[i].p /=sum;

# B[j]=A[i];

# entropy += A[i].p \* abs(log(A[i].p) / log(2));

# j++;

# }

# }

# 

# for (i = 0; i < alphabet\_num; i++){

# B[i].q = B[i-1].q + B[i].p/2;

# B[i].l = ceil(-log(B[i].p) / log(2)) + 1;

# }

# 

# for (i = 0; i < alphabet\_num; i++)

# {

# for (j = 0; j <= B[i].l; j++)

# {

# B[i].q \*= float(2);

# code[i][j] = floor(B[i].q);

# while (B[i].q >= 1)

# B[i].q -= 1;

# }

# }

# }

# void TreeSearch (vertex\* p, char \*data){

# if(p!=NULL) ///Поиск элемента

# {

# if(strcmp(p->data->Position,data) > 0){

# // cout << data << "-l" << endl << p->data->Position << "-l" << endl << endl;

# TreeSearch(p->left,data);

# }else{

# if(strcmp(p->data->Position,data) < 0){

# // cout << data << "-r" << endl << p->data->Position << "-r" << endl << endl;

# TreeSearch(p->right,data);

# }else{

# cout<<p->data->FIO<<" "<<p->data->Department<<"\t"<<p->data->Position<<" "<<p->data->Date<<endl;

# TreeSearch(p->next,data);

# }

# }

# }

# }

# int BSearch (list \*\*A, char year[2]) {

# int l = 0, r = N, m = 0, fam\_pos=0;

# char buffer[32];

# while (l < r) {

# m=(l+r)/2;

# if (strncmp(&A[m]->data->Date[6],year,2) == 0){

# return m;

# }

# if (strncmp(&A[m]->data->Date[6],year,2) < 0)

# l = m+1;

# else r = m-1;

# }

# return -1;

# }

# void Read\_base(FILE \*fp, list \*base){

# record \*mas = new record();

# fread((record \*)mas, sizeof(record), 1, fp);

# base->data=mas;

# base->prev=NULL;

# base->next=NULL;

# 

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

# mas = new record();

# list \*base\_prev;

# fread((record\*)mas, sizeof(record), 1, fp);

# base\_prev = base;

# base = base->next = new list();

# base->data = mas;

# base->prev = base\_prev;

# base->next = NULL;

# }

# }

# void Print(list \*base){

# int i;

# char n;

# SetConsoleCP(1251);

# cout << "1 for 4000 / 2 for 20 per click/ 3 for nothing"<< endl;

# cin>>n;

# if(n=='1'){

# i=-1;

# while(i<N){

# i++;

# cout<<i<<" "<<base->data->FIO<<" "<<base->data->Department<<"\t"<<base->data->Position<<" "<<base->data->Date<<endl;

# if(int result=\_kbhit()){

# printf( "\nKey '%c' was pressed. Press 'e' key to exit.\n", \_getch());

# char something;

# cin>>something;

# if(something=='e'){

# return;

# }

# }

# base=base->next;

# }

# }

# if(n=='2'){

# i=-1;

# int g=0;

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

# while((g++)<20)

# {

# cout<<i+g<<" "<<base->data->FIO<<" "<<base->data->Department<<"\t"<<base->data->Position<<" "<<base->data->Date<<endl;

# base=base->next;

# }

# if(g>=20){

# cout << "y for continue" << endl;

# cin >> n;

# if (n=='y')

# g=0;

# else

# return;

# }

# }

# }

# 

# 

# }

# void copy\_base(list \*a, list \*b) {

# b->prev = NULL;

# b->data = a->data;

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

# a = a->next;

# b->next = new list;

# b->next->prev = b;

# b = b->next;

# b->data = a->data;

# }

# b->next = NULL;

# }

# int DateMatch(record \*a, record \*b)

# {

# if(strcmp(&a->Date[6], &b->Date[6]) != 0){

# return strcmp(&a->Date[6], &b->Date[6]);

# }

# else if (a->Date[3] - b->Date[3] != 0) {

# return a->Date[3] - b->Date[3];

# }

# else if (a->Date[4] - b->Date[4] != 0) {

# return a->Date[4] - b->Date[4];

# }

# else if (a->Date[0] - b->Date[0] != 0) {

# return a->Date[0] - b->Date[0];

# }

# else if (a->Date[1] - b->Date[1] != 0) {

# return a->Date[1] - b->Date[1];

# }

# else if (strncmp(a->Position, b->Position,3) != 0) {

# return strncmp(a->Position, b->Position,3);

# }

# return 0;

# }

# void DigitalSort (list \*&head) {

# int i, j, k, l=10;

# char buffer[32];

# int d;

# list \*p, \*tail;

# queue q[256];

# for (j=0; j<=21; j++)

# {

# d=0;

# for (i=0; i<256; i++)

# {

# q[i].tail=q[i].head=NULL;

# }

# p=head;

# while (p!=NULL)

# {

# k=0;

# 

# if(p->data->Position[j] == ' '){

# d=0;

# }

# else

# {

# d=p->data->Position[j]+129;

# }

# 

# if(q[d].tail==NULL){

# k=1;

# }

# 

# q[d].head=p;

# if(k==1){

# q[d].tail=q[d].head;

# q[d].tail->prev=NULL;

# }

# else{

# q[d].tail->next=q[d].head;

# q[d].tail->next->prev=q[d].tail;

# q[d].tail=q[d].tail->next;

# }

# q[d].head=q[d].head->next;

# p=p->next;

# }

# for (i=0; i<256; i++) {

# if(q[i].tail !=NULL){

# q[i].tail->next=NULL;

# while(q[i].tail->prev != NULL){

# q[i].tail=q[i].tail->prev;

# }

# }

# 

# }

# 

# tail=NULL;

# for(i=0;i<256;i++)

# {

# if(q[i].tail!=NULL){

# head=q[i].tail;

# head->prev=tail;

# if(tail!=NULL)

# head->prev->next=head;

# while(q[i].tail!=NULL){

# tail=head;

# head=head->next;

# q[i].tail=q[i].tail->next;

# }

# }

# }

# while(tail->prev!=NULL){

# tail=tail->prev;

# }

# head=tail;

# }

# for (j=1; j>-1; j--)

# {

# d=0;

# for (i=0; i<256; i++)

# {

# q[i].tail=q[i].head=NULL;

# }

# p=head;

# while (p!=NULL)

# {

# k=0;

# if(p->data->Date[j]=='-'){

# d=0;

# }

# else

# {

# d=p->data->Date[j]+129;

# }

# 

# if(q[d].tail==NULL){

# k=1;

# }

# 

# q[d].head=p;

# if(k==1){

# q[d].tail=q[d].head;

# q[d].tail->prev=NULL;

# }

# else{

# q[d].tail->next=q[d].head;

# q[d].tail->next->prev=q[d].tail;

# q[d].tail=q[d].tail->next;

# }

# q[d].head=q[d].head->next;

# p=p->next;

# }

# for (i=0; i<256; i++)

# {

# if(q[i].tail !=NULL){

# q[i].tail->next=NULL;

# while(q[i].tail->prev != NULL){

# q[i].tail=q[i].tail->prev;

# }

# }

# 

# }

# 

# tail=NULL;

# for(i=0;i<256;i++)

# {

# if(q[i].tail!=NULL){

# head=q[i].tail;

# head->prev=tail;

# if(tail!=NULL)

# head->prev->next=head;

# while(q[i].tail!=NULL){

# tail=head;

# head=head->next;

# q[i].tail=q[i].tail->next;

# }

# }

# }

# while(tail->prev!=NULL){

# tail=tail->prev;

# }

# head=tail;

# if(j==0){

# j=5;

# }

# if(j==3){

# j=8;

# }

# if(j==6){

# break;

# }

# }

# }

# void add\_vertex(vertex \*&p, list\* mas, int w) {

# if (p == NULL) {

# p = new vertex;

# p->data = mas->data;

# p->weight = w;

# p->next = NULL;

# p->left = NULL;

# p->right = NULL;

# }

# else if (strcmp(p->data->Position,mas->data->Position) == 0) {

# add\_vertex(p->next, mas, w);

# }

# else if (strcmp(p->data->Position,mas->data->Position) > 0) {

# add\_vertex(p->left, mas, w);

# }

# else if (strcmp(p->data->Position,mas->data->Position) < 0) {

# add\_vertex(p->right, mas, w);

# }

# }

# void A2(int L, int R, list \*\*mas) {

# int wes = 0, sum = 0;

# int i;

# if (L <= R) {

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

# wes = wes + W[i];

# }

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

# if ((sum < (wes / 2)) && (sum + W[i]) > (wes / 2)) {

# break;

# }

# sum = sum + W[i];

# }

# //cout << L << " - " << R << " - " << wes << " - " << i << " - " << W[i] << endl;

# add\_vertex(root, mas[i], W[i]);

# A2(L, i - 1,mas);

# A2(i + 1, R,mas);

# }

# }

# void delete\_tree(vertex \*&p) {

# if (p != NULL) {

# delete\_tree(p->left);

# delete\_tree(p->right);

# delete p;

# }

# }

# void LRprint(vertex\* x){

# SetConsoleCP(1251);

# if (x) {

# LRprint(x->left);

# cout<<" "<<x->data->FIO<<" "<<x->data->Department<<"\t"<<x->data->Position<<" "<<x->data->Date<<endl;

# LRprint(x->next);

# LRprint(x->right);

# }

# }

# int size(vertex \*x)

# {

# if (x == NULL) {

# return 0;

# }

# else {

# return 1 + size(x->left) + size(x->right);

# }

# }

# int max(int x, int y)

# {

# if (x > y) return x;

# return y;

# }

# int height(vertex \*x)

# {

# if (x == NULL) {

# return 0;

# }

# else {

# return 1 + max(height(x->left), height(x->right));

# }

# }

# int sdp(vertex \*x, int l)

# {

# if (x == NULL) {

# return 0;

# }

# else {

# return l + sdp(x->left, l + 1) + sdp(x->right, l + 1);

# }

}

# Результаты

Рисунок 1. Неотсортированная база данных.

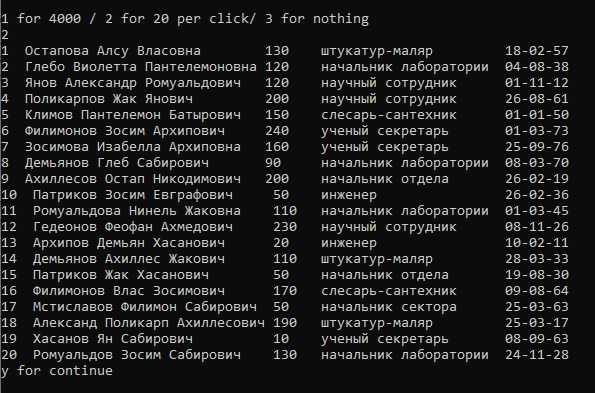


Рисунок 2. Отсортированная база данных по дате и должности.

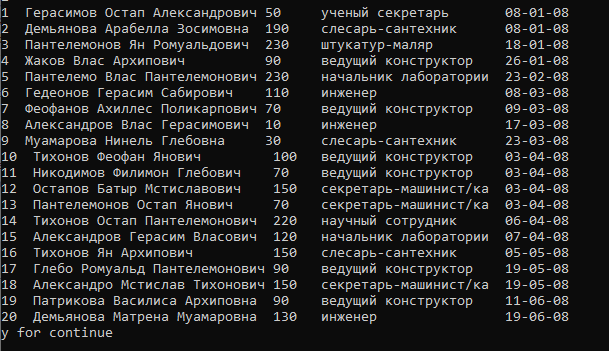


Рисунок 3. Бинарный поиск по году рождения

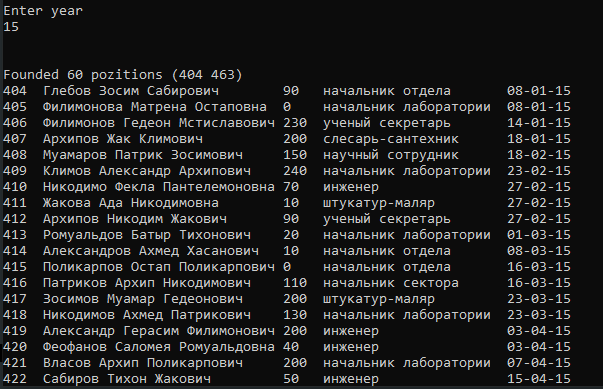


Рисунок 4. Дерево, ключ в дереве – должность.

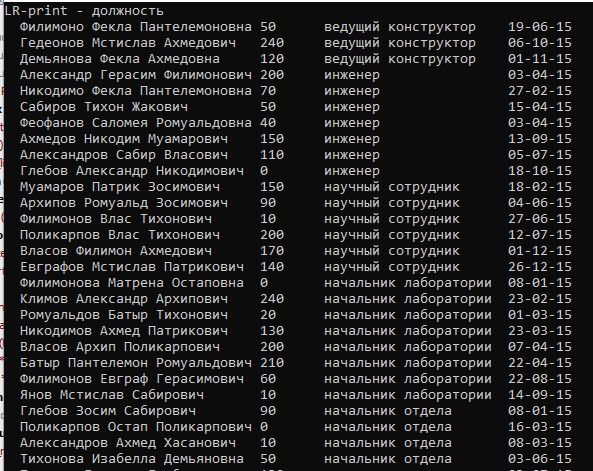


Рисунок 5. Поиск по дереву (элементы с одинаковым ключом).

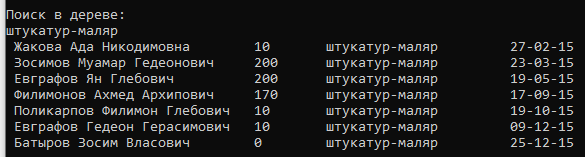
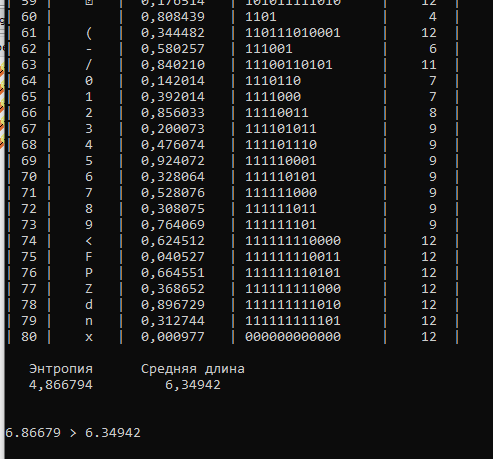


Рисунок 6. Кодовые слова, энтропия и средняя длина кодового слова.



# Выводы

В ходе выполнения курсовой работы были выполнены все поставленные задачи и реализованы необходимые алгоритмы: сортировки, поиска, построения дерева оптимального поиска (приближённый алгоритм А2), 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 + 1).

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

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