Министерство цифрового развития, связи

и массовых коммуникаций Российской Федерации

Сибирский Государственный Университет Телекоммуникаций и

Информатики

Кафедра прикладной математики и кибернетики

Курсовой проект

«Структуры и алгоритмы обработки данных»

Вариант 69

Выполнил студент 2 курса группы ИП-013:

Ириков Евгений Алексеевич

Преподаватель:

Дьячкова Ирина Сергеевна

Новосибирск 2021

Содержание:

1. *Постановка задачи……………………………………………………………..3*
2. *Основные идеи и хаpактеpистики пpименяемых методов…………….4*
   1. *Метод сортировки…………………………………………………4*
   2. *Двоичный поиск……………………………………………………...4*
   3. *Списки и очереди……………………………………………………..5*
   4. *Вид дерева и поиск…………………………………………………...5*
   5. *Метод кодирования………………………………………………….6*
3. *Описание структур данных и использованных алгоритмов……………..7*
   1. *Использованные структуры данных……………………………..7*
   2. *Особенности реализации алгоритмов……………………………7*
4. *Описание программы…………………………….………………………………..8*
   1. *Основные переменные и структуры……………………………..8*
   2. *Описание подпрограмм…………………………….………….…….9*
5. *Исходный текст пpогpаммы………………………….………….……………11*
6. *Результаты…………………………….………….………….………….………21*
7. *Выводы ……….………….………….………….………….………….…………..23*
8. Постановка задачи

Хранящуюся в файле базу данных (4000 записей) загрузить динамически в оперативную память компьютера в виде массива, вывести на экран по 20 записей (строк) на странице с возможностью отказа от просмотра. Упорядочить данные - **по сумме вклада и дате**, используя метод **Вилльямса-Флойда** в качестве метода сортировки. Упорядоченные данные вывести на экран.

Предусмотреть возможность быстрого поиска по **сумме вклада** 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 (приближеный алгоритм А2) по дате**, вывести на экран содержимое дерева и предусмотреть возможность поиска в дереве по запросу.

Закодировать файл базы данных статическим **кодом Хаффмена**, 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 "Обманутые вкладчики"

Стpуктуpа записи:

ФИО вкладчика: текстовое поле 30 символа

фоpмат <Фамилия>\_<Имя>\_<Отчество>

Сумма вклада: целое число (использовать unsigned short int)

Дата вклада: текстовое поле 10 символов

фоpмат дд-мм-гг

ФИО адвоката: текстовое поле 22 символа

фоpмат <Фамилия>\_<буква>\_<буква>

Пpимеp записи из БД:

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

130

15-03-46

Иванова\_И\_В\_\_\_\_\_\_\_\_\_\_\_

Варианты условий упорядочения и ключи поиска (К):

по сумме вклада и дате, К = сумма вклада;

1. Основные идеи и хаpактеpистики пpименяемых методов
   1. Метод сортировки

Метод Вилльямса-Флойда

Пирамидальная сортировка основана на алгоритме построения пирамиды. Последовательность ai, ai+1,…,ak называется *(i,k)-пирамидой*, если неравенство

aj≤min(a2j, а2j+1) (\*)

выполняется для каждого j, j=i,…,k для которого хотя бы один из элементов a2j, a2j+1 существует.

Например, массив А является пирамидой, а массив В ⎯ не является.

А=(а2 , а3 , а4 , а5 , а6 а7 , а8 )=(3, 2, 6, 4, 5, 7)

В=(b1, b2, b3, b4, b5, b6, b7)=(3, 2, 6, 4, 5, 7)

Процесс построения пирамиды выглядит следующим образом. Дана последовательность as+1,…,ak, которая является (*s+1, k*)-пирамидой. Добавим новый элемент х и поставим его на *s*-тую позицию в последовательности, т.е. пирамида всегда будет расширяться влево. Если выполняется (\*), то полученная последовательность – (*s, k*)-пирамида. Иначе найдутся элементы a2s+1,a2s такие, что либо a2s < as либо a2s+1 < as.

Пусть имеет место первый случай, второй случай рассматривается аналогично. Поменяем местами элементы as и a2s. В результате получим новую последовательность as’,as+1,…, a2s’,…, ak. Повторяем все действия для элемента a2s’ и т.д. пока не получим (*s, k*)-пирамиду.

Пирамидальная сортировка производится в два этапа. Сначала строится пирамида из элементов массива. По свойству (3) правая часть массива является (*n/2+1, n*)-пирамидой. Будем добавлять по одному элементу слева, расширяя пирамиду, пока в неё не войдут все элементы массива. Тогда по свойству (2) первый элемент последовательности – минимальный.

Произведём двустороннее усечение: уберём элементы a1,an. По свойству (1) оставшаяся последовательность является (2, n-1)-пирамидой. Элемент a1 поставим на последнее место, а элемент an добавим к пирамиде a2,…,an-1 слева. Получим новую (1, *n-1*)-пирамиду. В ней первый элемент является минимальным. Поставим первый элемент пирамиды на позицию *n-1*, а элемент an-1 добавим к пирамиде a2,…,an-1, и т.д. В результате получим обратно отсортированный массив.

Метод пирамидальной сортировки неустойчив, не зависит от исходного отсортированности массива.

* 1. 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

* 1. Списки и очереди

Списком называется последовательность однотипных элементов, связанных между собой указателями.

Поле next является указателем на элемент списка и может занимать произвольное место в структуре элемента. Однако если оно является первым элементом структуры, то его адрес совпадает с адресом элемента списка, и это позволяет оптимизировать многие операции со списками. Поле Record содержит информацию.

Очередь характеризуется тем, что при добавлении в очередь новый элемент ставится в конец списка, удаляется первый элемент последовательности.

Для работы с очередью необходимо иметь указатель на начало списка. Обозначим его Head. Также требуется дополнительный указатель на конец очереди. Обозначим его Tail.

* 1. Вид дерева и поиск

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).

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

* 1. Метод кодирования

Алгоритм построения оптимального кода Хаффмена

Упорядочим символы исходного алфавита А={a1,…,an} по убыванию их вероятностей p1≥p2≥…≥pn.

Если А={a1,a2}, то a1→0, a2→1.

Если А={a1,…,aj,…,an} и известны коды <aj → bj >, j = 1,…,n ,то для {a1,…aj’ ,aj’’…,an}, p(aj)=p(aj’)+ p(aj’’), aj’ → bj0, aj’’ →bj1.

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. Будем складывать две наименьшие вероятности и включать суммарную вероятность на соответствующее место в упорядоченном списке вероятностей до тех пор, пока в списке не останется два символа. Тогда закодируем эти два символа 0 и 1. Далее кодовые слова достраиваются, как показано на рисунке:

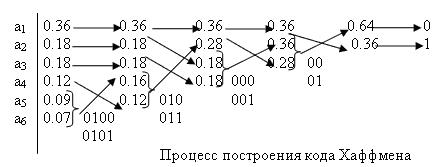
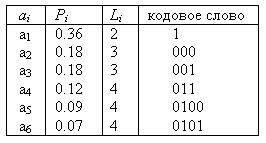


Таблица кода Хаффмена



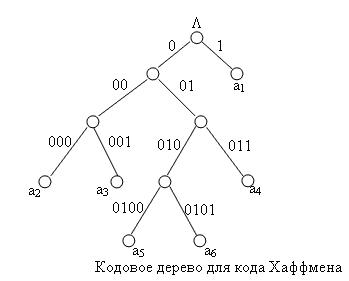
Посчитаем среднюю длину, построенного кода Хаффмена

Lср(P) = 1.0\*36 + 3\*0.18 + 3\*0.18 + 3\*0.12 + 4\*0.09 + 4\*0.07 = 2.44,

при этом энтропия данного источника равна

H = -(0.36\*log0.36 + 2\*0.18\*log0.18 +0.12\*log0.12+ 0.09\*log0.09 + 0.07\*log0.07) = 2.37

Код Хаффмена обычно строится и хранится в виде двоичного дерева, в листьях которого находятся символы алфавита, а на «ветвях» – 0 или 1. Тогда уникальным кодом символа является путь от корня дерева к этому символу, по которому все 0 и 1 собираются в одну уникальную последовательность.



1. Описание структур данных и использованных алгоритмов
   1. Использованные структуры данных

В курсовом проекте используются такие структуры данных как:

* Динамические массивы
* Статические
* Индексные массивы
* Индексные массивы указателей
* Списки (очередь)
* Деревья оптимального поиска (А1).
  1. Особенности реализации алгоритмов

Загрузка и вывод базы данных

Для загрузки базы данных разработана процедура scan\_database (), в которой производится считывание записей типа struct record. Данная процедура вызывается независимо от желания пользователя.

После загрузки в динамическую память выполняется построение индексного массива указателей.

За вывод элементов считанной базы данных отвечает процедура print\_database() Она предоставляет возможность постраничного просмотра базы данных(по 20 элементов на странице). Есть возможность прервать просмотр при помощи ввода любой другой клавиши.

Вспомогательные функции и процедуры для сортировки данных

При сортировке базы данных потребовалось реализовать дополнительные функции сравнения двух строк Compare\_deposit() и Compare\_date(), которые возвращают -1, если вторая строка больше, 1, если первая строка больше, 0, если они равны.

Вспомогательные функции для построения дерева оптимального поиска

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

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

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 Huffman().

1. Описание программы
   1. Основные переменные и структуры

struct record

{

char Name\_depositor[30];

unsigned short int Deposit;

char Date[10];

char Name\_lawyer[22];

};

Структура, используемая для работы с базой данных.

struct Node

{

record\* Record;

Node\* next;

};

Структура (список), используемая при поиске по базе данных.

record \*Record **–** поле данных

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

struct Vertex

{

record\* data;

Vertex\* left;

Vertex\* right;

};

Структура, представляющая дерево оптимального поиска (А1).

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

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

record Array[n] – массив структур

record \*Arr[n] – индексный массив.

int w[k], index[k]– массив весов и его индексный массив.

* 1. Описание подпрограмм

Процедуры обработки и вывода базы данных:

void scan\_database(record\* Array, record\*\* A, int n)

void print\_record(record\* Record)

void print\_database(record\* Array, int n)

void print\_array(record\*\* mas, int n)

scan\_database - считывание базы из файла и представление ее элементов в форме вышеперечисленных структур, возвращает указатель на массив записей.

print\_database – вывод неотсортированной базы данных

print\_array – вывод отсортированной базы данных

Функции и процедуры сортировки:

int Compare\_date(char\* A, char\* B)

int Compare\_deposit(record\* A, record\* B)

void Piram(record\*\* A, int L, int R)

void PiramSort(record\*\* A, int n)

Compare\_deposit – сортировка по сумме

Compare\_date – сравнение по дате

Piram/ PiramSort - Функции, сортирующие путём пирамидальной сортировки.

A - сортируемый массив. n - количество элементов в массиве.

Функции и процедуры для поиска в отсортированной базе данных:

int Binar(unsigned short int key, record\*\* A, int n)

void search\_array(record\*\* Search\_arr, Node\* root, int k)

void search(record\*\* Arr, int n, int &k, int &ind)

search – запрашивает у пользователя ключ поиска для поиска

Binar – бинарный поиск

search\_array – формирование списка по ключу

Процедуры и функции построения двоичного дерева оптимального поиска:

void tree(record\*\* Arr, int k)

void search\_in\_tree(Vertex\* root, char \*key)

void Print\_tree(Vertex\* p)

void delete\_tree(Vertex\* root)

void A1(record\*\* Arr, Vertex\*& root, int k)

void AddSDPTree(record\* D, Vertex\*& p)

search\_in\_tree – поиск в дереве элементов, соответствующих ключу. Выводит все найденные ключи.

Print\_tree - обход дерева, используемый для вывода на экран

A1 - непосредственно построение дерева

delete\_tree - освобождение памяти для построения дерева, чтобы не возникало проблем, в случае если до этого дерево уже создавалось

AddSDPTree - функция добавление вершины в дерево методом случайного дерева поиска

Процедуры и функции кодирования базы данных:

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

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

void coding(int n)

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

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

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

Get\_char\_counts\_from\_file – функция, подсчитывающая сколько раз каждый символ входит в базу данных, а также размер файла

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.

huffman / down / up - Функции, кодирующая символы методом Хаффмена

1. Исходный текст пpогpаммы

#include <vector>

#include <iostream>

#include <Windows.h>

#include <fstream>

#include <cmath>

#include <algorithm>

#include <unordered\_map>

using namespace std;

struct record

{

char Name\_depositor[30];

unsigned short int Deposit;

char Date[10];

char Name\_lawyer[22];

};

struct Node

{

record\* Record;

Node\* next;

};

struct Vertex

{

record\* data;

Vertex\* left;

Vertex\* right;

};

int Compare\_date(char\* A, char\* B)

{

for (int i = 6; i >= 0; i -= 3)

if (A[i] > B[i])

return 1;

else if (A[i] < B[i])

return -1;

else if (A[i + 1] > B[i + 1])

return 1;

else if (A[i + 1] < B[i + 1])

return -1;

return 0;

}

void print\_record(record\* Record)

{

cout << Record->Name\_depositor

<< " " << Record->Deposit

<< " " << Record->Date

<< " " << Record->Name\_lawyer << "\n";

}

void AddSDPTree(record\* D, Vertex\*& p)

{

if (p == NULL)

{

p = new Vertex;

p->data = D;

p->left = p->right = NULL;

}

else

{

if (Compare\_date(D->Date, p->data->Date) == -1)

AddSDPTree(D, p->left);

else

AddSDPTree(D, p->right);

}

}

void A1(record\*\* Arr, Vertex\*& root, int k)

{

int w[k], index[k], temp;

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

{

w[i]=rand()%100;

index[i]=i;

}

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

for( int j = 0; j < k - 1; j++){

if(w[index[j]] < w[index[j + 1]]){

temp = index [j];

index[j] = index[j + 1];

index[j + 1] = temp;

}

}

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

AddSDPTree(Arr[index[i]], root);

}

void delete\_tree(Vertex\* root)

{

if (root) {

delete\_tree(root->right);

delete\_tree(root->left);

delete root;

}

}

void Print\_tree(Vertex\* p)

{

if (p!=NULL) {

Print\_tree(p->left);

print\_record(p->data);

Print\_tree(p->right);

}

}

void search\_in\_tree(Vertex\* root, char \*key)

{

if (root)

{

if (Compare\_date(root->data->Date, key) == 1) {

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

}

else if (Compare\_date(root->data->Date, key) == -1) {

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

}

else if (Compare\_date(root->data->Date, key) == 0) {

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

printf("\n");

print\_record(root->data);

printf("\n");

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

}

}

}

void tree(record\*\* Arr, int k)

{

Vertex\* root = NULL;

A1(Arr, root, k);

Print\_tree(root);

char key[10];

printf("input key: ");

cin >> key;

search\_in\_tree(root, key);

system("pause");

delete\_tree(root);

}

int Compare\_deposit(record\* A, record\* B)

{

if (A->Deposit > B->Deposit)

return 1;

else if (A->Deposit < B->Deposit)

return -1;

else

return Compare\_date(A->Date, B->Date);

}

void scan\_database(record\* Array, record\*\* A, int n)

{

FILE\* database;

database = fopen("testBase3.dat", "rb");

fread((record\*)Array, sizeof(record), n, database);

fclose(database);

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

{

A[i] = &Array[i];

}

}

void print\_database(record\* Array, int n)

{

int temp;

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

{

cout << i << ". " << Array[i].Name\_depositor << " " << Array[i].Deposit << " " << Array[i].Date << " " << Array[i].Name\_lawyer << "\n";

if ((i % 20 == 0) && (i != 0))

{

cout << "\nContinue? 0/1: ";

scanf("%d", &temp);

if (temp == 0) break;

else system("cls");

}

}

system("pause");

}

void print\_array(record\*\* mas, int n)

{

int temp;

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

{

cout << i << ". " << mas[i]->Name\_depositor << " " << mas[i]->Deposit << " " << mas[i]->Date << " " << mas[i]->Name\_lawyer << "\n";

if ((i % 20 == 0) && (i != 0))

{

cout << "\nContinue? 0/1: ";

scanf("%d", &temp);

if (temp == 0) break;

else system("cls");

}

}

system("pause");

}

void Piram(record\*\* A, int L, int R)

{

int j, i;

record\* temp;

temp = A[L];

i = L;

while (1)

{

j = 2 \* i;

if (j >= R)

break;

if (j < R)

{

if (Compare\_deposit(A[j + 1], A[j]) != -1)

j++;

}

if (Compare\_deposit(temp, A[j]) != -1)

break;

A[i] = A[j];

i = j;

}

A[i] = temp;

}

void PiramSort(record\*\* A, int n)

{

int i, j, R = n - 1;

record\* temp;

int L = (n - 1) / 2;

while (L >= 0)

{

Piram(A, L, R);

L--;

}

while (R > 0)

{

temp = A[R];

A[R] = A[0];

A[0] = temp;

R--;

Piram(A, 0, R);

}

}

int Binar(unsigned short int key, record\*\* A, int n)

{

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

while (L < R)

{

m = (L + R) / 2;

if (A[m]->Deposit < key)

L = m + 1;

else

R = m;

}

if (key == A[R]->Deposit)

return R;

else

return -1;

}

void search\_array(record\*\* Search\_arr, Node\* root, int k)

{

Node\* p = root;

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

{

Search\_arr[i] = (p->Record);

p = p->next;

}

}

void search(record\*\* Arr, int n, int &k, int &ind)

{

unsigned short int key;

Node \*head, \*tail, \*p;

head = tail = p = NULL;

printf("input key:");

scanf("%hu", &key);

ind = Binar(key, Arr, n);

if (ind == -1) {

printf("Not found\n");

printf("Press Any Button\n\n");

system("pause");

}

else

{

head = new Node{

Arr[ind], NULL

};

tail = head;

int i = ind + 1;

while (Arr[i]->Deposit == key)

{

tail->next = new Node{

Arr[i], NULL

};

tail = tail->next;

i++;

}

k = i - ind;

record\* Search\_arr[k];

search\_array(Search\_arr, head, k);

print\_array(Search\_arr, k);

}

}

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

int i = 0, j = 0;

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

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

else {

j = i;

break;

}

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

j = 0;

break;

}

}

array[j] = q;

return j;

}

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

char pref[20];

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

int l = Length[j];

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

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

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

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

}

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

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

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

}

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

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

Length[n-1] = l + 1;

Length[n-2] = l + 1;

}

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

if (n == 2) {

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

Length[0] = 1;

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

Length[1] = 1;

} else {

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

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

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

down(n, j, Length, c);

}

}

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

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

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

for(int i=0; i<sizeof(record) \* n; i++)

ch\_arr[i]='\0';

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 n) {

int file\_size;

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

counter\_map = get\_char\_counts\_from\_file("testBase3.dat", file\_size, n);

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

counter\_map.clear();

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

cout << "Probabil. char\n";

for (auto i : probabilities) {

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

}

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

char c[N][20];

int Length[N];

for (auto &i : Length) {

i = 0;

}

auto p = new double[N];

double chance\_l[N];

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

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

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

}

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

cout << "\nHaffmanCode:\n";

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

double avg\_len = 0;

double entropy = 0;

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

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

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

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

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

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

}

cout << '\n';

}

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

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

<< "Average length < entropy + 1\n"

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

system("pause");

}

int main()

{

SetConsoleCP(1251);

int n = 4000, temp, ind, k = -1;

record Array[n];

record\* Arr[n];

scan\_database(Array, Arr, n);

PiramSort(Arr, n);

while (1)

{

cout << "1 - Print BD " << endl;

cout << "2 - Print Array " << endl;

cout << "3 - Search " << endl;

cout << "4 - Tree " << endl;

cout << "5 - Coding " << endl;

cout << "0 - End " << endl;

scanf("%d", &temp);

switch (temp)

{

case 1:

print\_database(Array, n);

break;

case 2:

print\_array(Arr, n);

break;

case 3:

search(Arr, n, k, ind);

break;

case 4:

if(k < 0)

cout << "Search first\n";

else

tree(&Arr[ind], k);

break;

case 5:

coding(n);

break;

default:

return 1;

}

system("cls");

}

}

1. Результаты

Рисунок 1. Неотсортированная база данных.

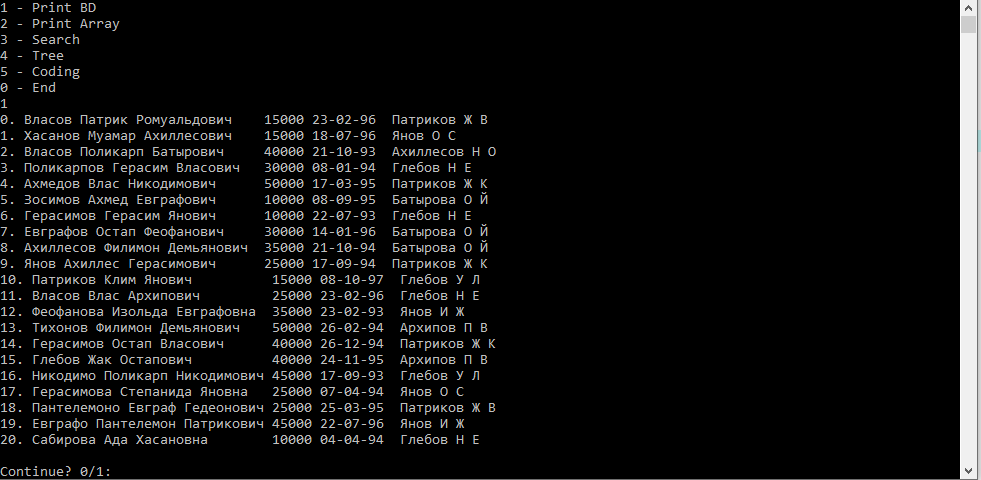


Рисунок 2. Отсортированная по сумме и дате база данных.

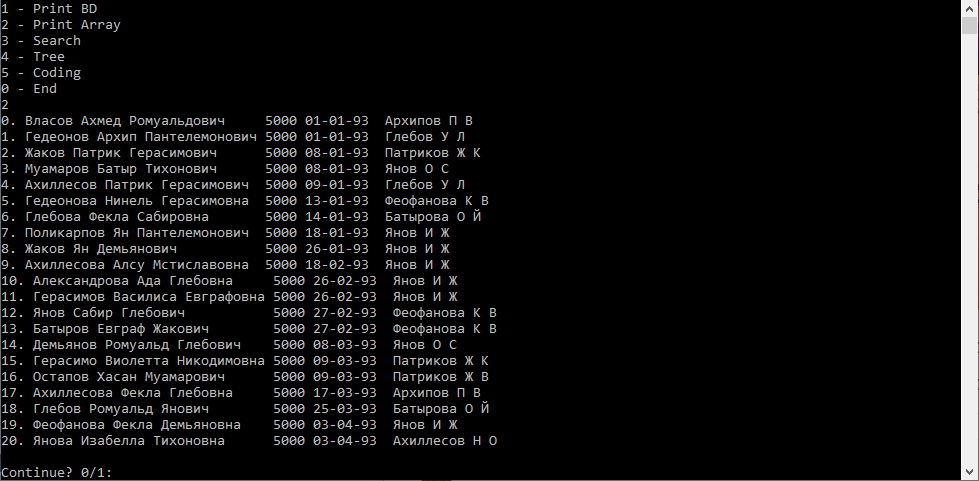


Рисунок 3. Очередь из записей, полученных в результате поиска (5000).

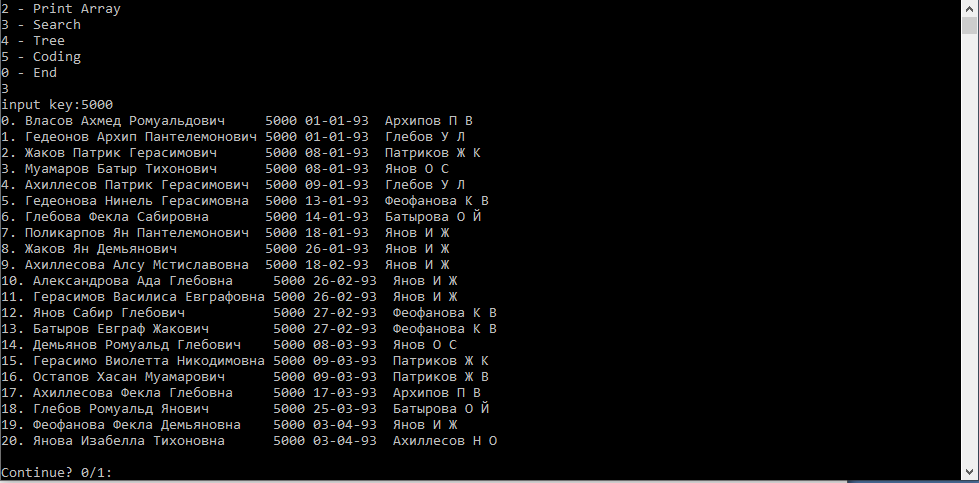


Рисунок 4. Пример дерева и поиск по ключу (дате)

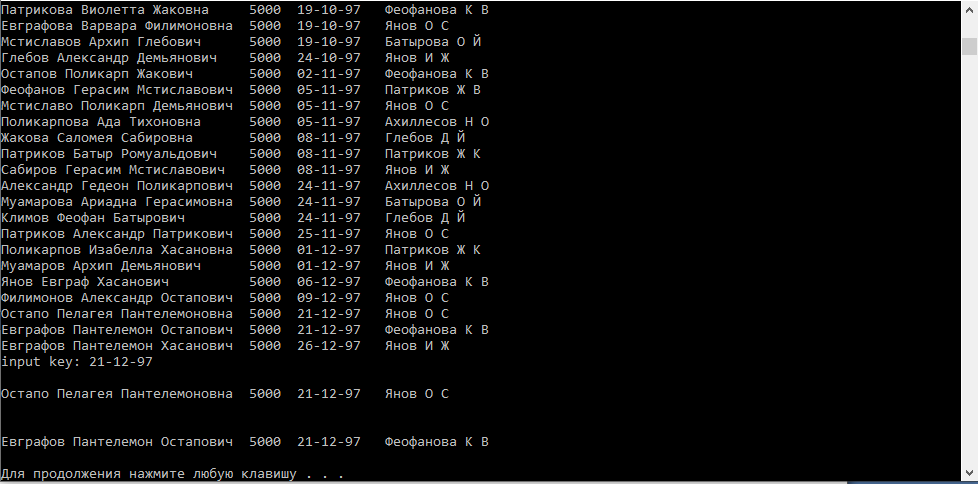


Рисунок 6. Примеры кодовых слов, энтропия и средняя длина кодового слова.



1. Выводы

В ходе выполнения курсовой работы были выполнены все поставленные задачи и реализованы необходимые алгоритмы: сортировки, поиска, построения дерева оптимального поиска, 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 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.