Федеральное агентство связи
Сибирский государственный университет телекоммуникаций и информатики

Кафедра прикладной математики и кибернетики

Курсовая работа
«Структуры и алгоритмы обработки данных»
Вариант 161

Выполнил: студент группы ИП-917
Лобакин Н.К
Проверил:
Турцев Андрей Андреевич

Новосибирск 2020

Содержание

1. ПОСТАНОВКА ЗАДАЧИ

Хранящуюся в файле базу данных загрузить в оперативную память компьютера и построить индексный массив, упорядочивающий данные **по дням рождения и ФИО**, 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 **АВЛ-дерево по автору** , и предусмотреть возможность поиска в дереве по запросу.

Закодировать файл базы данных **кодом Хаффмана**, предварительно оценив вероятности всех встречающихся в ней символов. Построенный код вывести на экран, упакованную базу данных записать в файл, вычислить коэффициент сжатия данных.

Библиографическая база данных "Жизнь замечательных людей"
Структура записи:
Автор: текстовое поле 12 символов
формат <Фамилия>_<буква>_<буква>
Заглавие: текстовое поле 32 символа
формат <Имя>_<Отчество>_<Фамилия>
Издательство: текстовое поле 16 символов
Год издания: целое число
Кол-во страниц: целое число
Пример записи из БД:
Кловский_В_Б
Лев_Николаевич_Толстой_____
Молодая_гвардия_
1963
864
Варианты условий упорядочения и ключи поиска:
С = по издательству и автору, К = три первые буквы
издательства.

2. ОСНОВНЫЕ ИДЕИ И ХАРАКТЕРИСТИКИ ПРИМЕНЯЕМЫХ МЕТОДОВ

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

Например, массив A является пирамидой, а массив B не является.

A=(a2 , a3 , a4 , a5 , a6 a7 , a8)=(3, 2, 6, 4, 5, 7)

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

Свойства пирамиды:

1. Если последовательность ai, ai+1,…,ak-1, ak является (i, k)-пирамидой, то последовательность ai+1,…,ak-1, полученная усечением элементов с обоих концов последовательности, является (i+1, k-1)пирамидой.

2. Если последовательность a1…an – (1, n)-пирамида, то a1 – минимальный элемент последовательности.

3. Если a1, a2…,an/2,an/2+1,…an-произвольная последовательность, то последовательность an/2+1,…,an является (n/2+1, n)-пирамидой.

Процесс построения пирамиды выглядит следующим образом:

Дана последовательность as+1,…,ak, которая является (s+1, k)-пирамидой. Добавим новый элемент x 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.

Общее количество операций сравнений и пересылок для пирамидальной сортировки:

C ≤ 2n log n+n+2, M ≤ n log n+6.5n-4.

Таким образом, **C=O(n log n), M=O(n log n) при n** → ∞. Отметим некоторые свойства пирамидальной сортировки.

Метод пирамидальной сортировки неустойчив и не зависит от исходной отсортированности массива.

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 $\lceil \log_2 n \rceil$. Таким образом, трудоёмкость двоичного поиска в обоих случаях

$C=O(\log n),\ n \to \infty.$

2.3 ДЕРЕВО И ПОИСК ПО ДЕРЕВУ

АВЛ-дерево — сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем на 1.

Повороты при балансировке. Рассмотрим, что может произойти при включении новой вершины в сбалансированное по высоте дерево. Пусть r – корень АВЛ-дерева, у которого имеется левое поддерево (TL) и правое поддерево (TR). Если добавление новой вершины в левое поддерево приведет к увеличению его высоты на 1, то возможны три случая:

1) если hL = hR, то TL и TR станут разной высоты, но баланс не будет нарушен;
2) если hL < hR, то TL и TR станут равной высоты, т. е. баланс даже улучшится;
3) если hL > hR, то баланс нарушиться и дерево необходимо перестраивать.

Если в какой-либо вершине баланс высот нарушается, то необходимо так перестроить имеющееся дерево, чтобы восстановить баланс в каждой вершине. Для восстановления баланса будем использовать **процедуры поворотов** АВЛ-дерева.

Поиск элемента с заданным ключом, включения нового элемента, удаления элемента – каждое из этих действий в АВЛ-дереве можно произвести в худшем случае за **O(log n)** операций.

2.4 МЕТОД КОДИРОВАНИЯ

Алфавитный код Гилберта – Мура

Оптимальный код Хаффмана обладает минимальной средней длиной кодового слова среди всех побуквенных кодов для данного источника с алфавитом A={a1,…,an} и вероятностями pi=P(ai).

Рассмотрим алгоритм построения оптимального кода Хаффмана, который основывается на утверждениях лемм предыдущего параграфа.

1. Упорядочим символы исходного алфавита A={a1,…,an} по убыванию их вероятностей p1≥p2≥…≥pn.
2. Если A={a1,a2}, то a1→0, a2→1.
3. Если A={a1,…,aj ,…,an} и известны коды , j = 1,…,n, то для алфавита {a1,…aj’ , aj"…,an} с новыми символами aj' и aj" вместо aj , и вероятностями p(aj)=p(aj')+ p(aj"), код символа aj заменяется на коды aj' → bj0, aj" →bj1.

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. Здесь символы источника уже упорядочены в соответствии с их вероятностями. Будем складывать две наименьшие вероятности и включать 20 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 4.

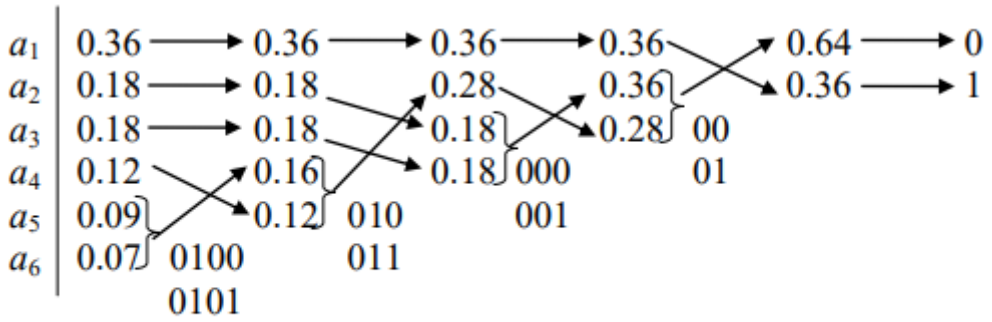


Рисунок 4 Процесс построения кода Хаффмана

Таблица 5 Код Хаффмана

a_i	p_i	L_i	кодовое слово
a_1	0.36	2	1
a_2	0.18	3	000
a_3	0.18	3	001
a_4	0.12	4	011
a_5	0.09	4	0100
a_6	0.07	4	0101

Посчитаем среднюю длину, построенного кода Хаффмана:

Lcp(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:

H(p1,…,p6) = −0.36*log0.36 – 2*0.18*log0.18 − 0.12*log0.12 − 0.09*log0.09 − 0.07*log0.07=2.37

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 использовалось меню**,** которое обеспечивает корректное и незатруднительное использование программы и предоставляет возможность многократного выбора различных вариантов обработки базы данных, в зависимости от задач пользователя. Меню выводится отдельной процедурой void menu().

2. Загрузка и вывод базы данных

Загрузка базы данных осуществляется с помощью fread.
За вывод элементов считанной базы данных отвечает процедура ***void*** *PrintData (record *tt, int size).* Она предоставляет возможность просмотра базы данных по 20 элементов, Клавиша «а» предоставляет возможность прервать просмотр в любой момент времени.

3. Вспомогательные функции и процедуры для сортировки данных

Для сортировки данных используется процедура **void** heapsort (record a[], int size, int keyIndex). Доступ к записям базы данных осуществляется через индексный массив, для сортировки по издательству и автору используется string key(int index)

4. Особенности реализации бинарного поиска

Бинарный поиск по отсортированной базе осуществляется в процедуре **void** Search_Binary(record arr[], int left, int right, string key, int keyIndex), доступ к записям ведётся через индексный массив **tt**, найденные записи заносятся в другой индексный массив **temp.** Для вывода на монитор найденных записей используется процедура void PrintData (record *tt, int size).

5. Вспомогательные функции и процедуры для построения Авл-дерева

Построение дерева осуществляется в функции **void** add_AVL(vertex *(&p), record x,int index). С помощью функций **void** LL1 (vertex *(&p)), **void** RR1 (vertex *(&p)), **void** LR (vertex *(&p)), **void** RL (vertex *(&p)) осуществляется балансировка дерева.
Одинаковы элементы записываются в одну вершину.

6. Кодирование данных

Функция void Down (int n, int j, vector<vector<int>> &c) формирует кодовые слова.
Функция int Up(int n, vector<double> &p, double q) находит в массиве P место, куда вставить число q, и вставляет его, сдвигая вниз остальные элементы.

4. ОПИСАНИЕ ПРОГРАММЫ

4.1. ОСНОВНЫЕ ПЕРЕМЕННЫЕ И СТРУКТУРЫ

```
struct record {
    char author[12];
    char title[32];
    char publisher[16];
    short int year;
    short int num_of_page;
    string key( int index ) {
        string input;
        switch ( index ) {
            case KEY_SORT:
                 return string( publisher ) + string( author );
            case KEY_SPIS:
                string data;
                for (int i = 0; i < 3; ++i)
                    data += publisher[i];
                return string(data);
                                }
        return "sa";

    }
    string keySpis(int index) {
        string data;
        for (int i = 0; i < 3; ++i)
            data += publisher[i];
        return data;
    }

    string keyTree() {
        string dat;
        for (int i = 0; i < 3; ++i)
            dat+= author[i];
        return dat;
    }
    string toString() {
        return string(author)  + (' ') + title + publisher+to_string(year) + (' ')+
to_string(num_of_page) +(' ') + ("\n");
    }
} ttNo[4000], tt[4000], temp[4001];;
```

Запись, используемая для работы с базой данных «Жизнь замечательных людей». ttNO- оригинальная БД. TT- для сортировка БД. Temp- для бинарного поиска

```
struct vertex {
        string fas;
        bool ch;
        int i;
        record data;
        int balance;
        vertex *left;
        vertex *right;
};
```

Структура, представляющая Авл-дерево.
int fas – записываются все одинаковые элементы.
bool ch - указывает на содержание одинаковых элементов в вершине.
int i – номер элемента.
vertex *left, vertex *right – указатели на левое и правое поддеревья.
record data - поле данных (адрес элемента в основном массиве структур «Жизнь замечательных людей»).
root – указатель на корень дерева.

4.2. ОПИСАНИЕ ПОДПРОГРАММ

Процедуры, вывода меню:

1. void menu() – выводит меню.

Процедуры начальной обработки базы данных:

2. void PrintData(record *tt, int size) – осуществляет просмотр базы данных.

Функции и процедуры сортировки:

3. void heapSort(record a[], int size, int keyIndex) – сортирует базу данных по издательству и автору.
4. void downHeap(record a[], int k, int n, int keyIndex) – построение пирамиды.

Функции и процедуры для поиска в отсортированной базе данных:

5. void Search_Binary(record arr[], int left, int right, string key, int keyIndex) – осуществляет поиск в базе.

Процедуры и функции построения дерева оптимального поиска (A2):

6. void add_AVL(vertex *(&p), record x,int index) –добавление вершины в дерево
7. LL1(vertex *(&p)), RR1(vertex *(&p)), LR(vertex *(&p)), RL(vertex *(&p)) – осуществляют балансировку дерева
8. vertex *search(vertex *root, string key)- поиск по дереву и ключу

Процедуры и функции кодирования базы данных:

9.void Down (int n, int j, vector<vector<int>> &c) формирует кодовые слова.
10.int Up(int n, vector<double> &p, double q) находит в массиве P место, куда вставить число q, и вставляет его, сдвигая вниз остальные элементы.
11. string printHuffman(string text)- вывод кода.
12. void Huffman(int n, vector<double> &p, vector<vector<int>> &c) – построение кода Хаффмана

Основная программа:

13. main() – основная программа, в которой вызываются процедура вывода меню, а также в зависимости от выбранного пункта меню вызываются соответствующие процедуры.

5. ТЕКСТ ПРОГРАММЫ

```
#include <iostream>
#include <cstring>
#include <string.h>
#include <sstream>
#include <conio.h>
#include <vector>
#include <math.h>
#include <map>
#include <algorithm>
#include <iomanip>
void clear() {
   system("cls");
}

using namespace std;
const int KEY_SORT = 0;
const int KEY_SPIS = 1;
struct record {
   char author[12];
   char title[32];
   char publisher[16];
   short int year;
   short int num_of_page;
   string key( int index ) {
      string input;
      switch ( index ) {
         case KEY_SORT:
               return string( publisher ) + string( author );
         case KEY_SPIS:
            string data;
            for (int i = 0; i < 3; ++i)
               data += publisher[i];
            return string(data);
                              }
      return "sa";

   }
   string keySpis(int index) {
      string data;
      for (int i = 0; i < 3; ++i)
         data += publisher[i];
      return data;
   }

   string keyTree() {
```

```
        string dat;
        for (int i = 0; i < 3; ++i)
            dat+= author[i];
        return dat;
    }
    string toString() {
        return string(author)  + (' ') + title + publisher+to_string(year) + (' ')+
to_string(num_of_page) +(' ') + ("\n");
    }
} ttNo[4000], tt[4000], temp[4001];;
//--------------------PRINTdATA---------------//
void printData( record *tt, int size ) {
    if ( size == -20 ) return;
    int index = 0;
    char letter;
    do {
        clear();
        cout<<"w-next    s-back   a-back to menu"<<endl;

cout<<"author"<<"\t\t"<<"title"<<"\t\t\t\t"<<"publisher"<<"\t"<<"year"<<"\t"<<"num_of_page
"<<"\t"<<endl;
        for ( int i = index ; i < index + 20; ++i) {
            if (tt[i].year != 0)

cout<<i+1<<")"<<tt[i].author<<"\t"<<tt[i].title<<"\t"<<tt[i].publisher<<"\t"<<tt[i].year<<"\t"<
<tt[i].num_of_page<<"\t"<<endl;
        }

        cin >> letter;
        switch (letter) {
            case 'a': // back to menu
                return;
            case 'w': // next
                if (index < size)
                    index += 20;
                break;
            case 's': // back
                if (index > 0)
                    index -= 20;
                break;
        }
    } while (true);
}
//----------------------------SORT---------------------////
void downHeap(record a[], int k, int n, int keyIndex) {
    record new_elem;
    int child;
```

```
   new_elem = a[k];

   while (k <= n / 2) {
      child = 2 * k;
      if (child < n &&
         strcmp(a[child].key(keyIndex).c_str(), a[child + 1].key(keyIndex).c_str()) < 0)
         child++;
      if (strcmp(new_elem.key(keyIndex).c_str(), a[child].key(keyIndex).c_str()) >= 0)
         break;
      a[k] = a[child];
      k = child;
   }
   a[k] = new_elem;
}
void heapSort(record a[], int size, int keyIndex) {
   int i;
   record temp;

   for (i = size / 2 - 1; i >= 0; i--) downHeap(a, i, size - 1, keyIndex);


   for (i = size - 1; i > 0; i--) {
      temp = a[i];
      a[i] = a[0];
      a[0] = temp;

      downHeap(a, 0, i - 1, keyIndex);
   }
}
//---------------------------------AVL-----------------------------------------//
bool rise;
string fall;
struct vertex {
        string fas;
        bool ch;
   int i;
   record data;
   int balance;
   vertex *left;
   vertex *right;
};

void LL1(vertex *(&p)) {

   vertex *q;
   q = p->left;
   if (q->balance == 0) {
```

```
        q->balance = 1;
        p->balance = -1;
    }
    else {
        q->balance = 0;
        p->balance = 0;
    }
    p->left = q->right;
    q->right = p;
    p = q;

}

void RR1(vertex *(&p)) {

    vertex *q;
    q = p->right;
    if (q->balance == 0) {
        q->balance = 1;
        p->balance = -1;
    }
    else {
        q->balance = 0;
        p->balance = 0;
    }
    p->right = q->left;
    q->left = p;
    p = q;
}

void LR(vertex *(&p)) {

    vertex *q = new vertex;
    vertex *r = new vertex;
    q = p->left;
    r = q->right;
    if (r == NULL) return;
    if (r->balance < 0)p->balance = 1;
    else p->balance = 0;

    if (r->balance > 0)q->balance = -1;
    else q->balance = 0;

    r->balance = 0;
    p->left = r->right;
    q->right = r->left;
    r->left = q;
```

```
    r->right = p;
    p = r;
}

void RL(vertex *(&p)) {

    vertex *q = new vertex;
    vertex *r = new vertex;
    q = p->right;
    r = q->left;
    if (r == NULL)return;
    if (r->balance > 0)p->balance = -1;
    else p->balance = 0;

    if (r->balance < 0)q->balance = 1;
    else q->balance = 0;

    r->balance = 0;
    p->right = r->left;
    q->left = r->right;
    r->left = p;
    r->right = q;
    p = r;
}

void add_AVL(vertex *(&p), record x,int index) {
    if (p == NULL) {

        p = new vertex;
        p->data = x;
        p->i = index;
        p->left = NULL;
        p->right = NULL;
        p->balance = 0;
        rise = true;
          p->ch=true;
    }else if (strcmp(p->data.keyTree().c_str(), x.keyTree().c_str())  < 0) {
        add_AVL(p->left, x,index);
        if (rise) {
            if (p->balance > 0) {
                p->balance = 0;
                rise = false;
            }
            else if (p->balance == 0) {
                p->balance = -1; rise = true;
            }
            else if (p->left->balance < 0) {
```

```
            LL1(p);
            rise = false;
         }
         else {
            LR(p);
            rise = false;
         }
      }
   }

   else if (strcmp(p->data.keyTree().c_str(), x.keyTree().c_str())  >0) {
      add_AVL(p->right, x,index);

      if (rise) {
         if (p->balance < 0) { p->balance = 0; rise = false; }
         else if (p->balance == 0) { p->balance = 1; rise = true; }
         else if (p->right->balance > 0) {
            RR1(p);
            rise = 0;
         }
         else {
            RL(p);
            rise = false;
         }
      }
   }else if (strcmp(p->data.keyTree().c_str(), x.keyTree().c_str())  ==0){
        p->ch=true;
        fall= p->data.toString()  +  x.toString();
        p->fas=p->fas+fall;

        }
}
//----------------------------------Search--------------------------------//
int sizeSearch = 0;
vertex *search(vertex *root, string key) {
   vertex *p = root;
   while (p != NULL) {
      if (strcmp(key.c_str(), p->data.keyTree().c_str()) > 0) p = p->left;
      else if (strcmp(key.c_str(), p->data.keyTree().c_str()) < 0) p = p->right;
      else break;
   }
   if (p != NULL) {
      search(p->left, key);
      sizeSearch++;
      if(p->ch==false){
      cout << (p->i + 1) << ") author: " << p->data.author << "\t"
          << "title: " << p->data.title << "   \t"
```

```
            << "publisher: " << p->data.publisher << "\t"
            << "year: " << p->data.year << "\t"
            << "num_of_page: " << p->data.num_of_page << endl;
         }else cout << (p->i + 1)<<") "<<p->fas;
        search(p->right, key);
                return p;
    } else {
        return p;
    }
}
// ----------------------------------SEARCH BINARY----------------///
int indexx = 0;
char let;
void Search_Binary(record arr[], int left, int right, string key, int keyIndex) {
    int midd = 0;
    while (true) {
        midd = (left + right) / 2;
        if (left > right)
            return;
        if (strcmp(key.c_str(), arr[midd].keySpis(keyIndex).c_str()) < 0)
            right = midd - 1;
        else if (strcmp(key.c_str(), arr[midd].keySpis(keyIndex).c_str()) > 0)
            left = midd + 1;
        else {
            while (true) {
                if (strcmp(arr[midd - 1].keySpis(keyIndex).c_str(),
arr[midd].keySpis(keyIndex).c_str()) == 0)
                    midd--;
                else
                    break;
            }
            temp[indexx++] = arr[midd];
            while (true) {
                if (strcmp(arr[midd].keySpis(keyIndex).c_str(), arr[midd +
1].keySpis(keyIndex).c_str()) == 0) {
                    temp[indexx++] = arr[midd + 1];
                    midd++;
                } else
                    break;
            }
            return;
        }
    }
}
//-----------------Haffman---------------------------------//
int Up(int n, vector<double> &p, double q) {
    int j;
```

```
    for (int i = n - 1; i >= 0; i--) {
        if (i > 0 && p[i - 1] < q) {
            p[i] = p[i - 1];
            j = i;
        } else {
            j = i;
            break;
        }
    }
    p[j] = q;
    return j;
}

void Down(int n, int j, vector<vector<int>> &c) {
    vector<int> s(c[j].size());
    s = c[j];
    for (int i = j; i < n - 1; i++) {
        c[i] = c[i + 1];
    }
    c[n - 1] = s;
    c[n] = s;
    c[n - 1].push_back(0);
    c[n].push_back(1);
}

void Huffman(int n, vector<double> &p, vector<vector<int>> &c) {
    double q;
    int j;
    if (n == 1) {
        c[0].push_back(0);
        c[1].push_back(1);
    } else {
        q = p[n - 1] + p[n];
        j = Up(n, p, q);
        Huffman(n - 1, p, c);
        Down(n, j, c);
    }
}

string printHuffman(string text) {
    map<char, int> mp;
    int count = 0;
    double huffman = 0;

    stringstream inp(text);
    while (1) {
        char c = inp.get();
```

```
    if (inp.eof()) {
        break;
    }
    count++;
    mp[c]++;

}

vector<pair<char, double>> p;

for (auto i : mp) {
    p.push_back({i.first, i.second / (double) count});
}

double h = 0;

for (auto i : p) {
    h += i.second * (-log2(i.second));
}
vector<vector<int>> c(p.size());
stringstream out;
int mx = 0;

sort(p.begin(), p.end(), [](pair<char, double> a, pair<char, double> b) {
    return a.second > b.second;
});
vector<double> r;
for (auto i : p) {
    r.push_back(i.second);
}
Huffman(p.size() - 1, r, c);
mx = 0;
for (auto i : c) {
    mx = max(mx, (int) i.size());
}

for (int i = 0; i < p.size(); i++) {
    out << setprecision(5) << p[i].first << " " << p[i].second << "\t";
    for (auto j : c[i]) {
        out << j;
    }
    for (int j = 0; j < mx - c[i].size() + 3; j++) {
        out << " ";
    }
    out << c[i].size() << endl;
}
for (int i = 0; i < p.size(); i++) {
```

```
        huffman += p[i].second * c[i].size();
    }

    out << "Huffman";
    out << "\t" << huffman << endl;
    return out.str();
}
//-----------------------MENU--------------------------//
void menu() {
    char letter;
    char latter;
    string year;
    string data;
    vertex *root_avl = NULL;
    do {
        clear();
        cout << "1) Show database1" << endl;
        cout << "2) heapsort and show database1" << endl;
        cout << "3) AVL tree search " << endl;
        cout << "4) Haffman" << endl;
        cout << " Press q to exit"<<endl;

        cin >> letter;

        switch (letter) {
            case '1':
                printData(ttNo, 3980);
                break;
            case '2':
                                heapSort(tt, 4000, KEY_SORT);
                printData(tt, 3980);
                                break;
            case '3':
                                indexx = 0;
                heapSort(tt, 4000, KEY_SPIS);
                do {
                    clear();
                    cout << "Enter the first 3 letters of the publisher ([q] exit):  ";
                    cin >> year;
                    for (int i = 0; i < 3; ++i)
                        data += year[i];
                    if (year == "q") break;
                    Search_Binary(tt, 0, 4000, data, KEY_SPIS);
                    if (indexx == 0) {
                        cout << "key not found!" << endl;
                        continue;
                    } else {
```

```
                printData(temp,indexx - 20);
                root_avl = NULL;

                for (int i = 0; i < indexx; ++i)
                   add_AVL(root_avl, temp[i],i);

                clear();
                cout << "Enter the first 3 letters of the Author:  ";
                for (int i = 0; i < 3; ++i)
                   data += year[i];
                cin >> year;
                search(root_avl, year);
                if (sizeSearch == 0) {
                   cout << "key not found!" << endl;
                   cin >> letter;
                   break;
                }
                sizeSearch = 0;

                for (int i = 0; i < indexx; i++)
                   temp[i] = temp[4001];
                indexx = 0;

                cin >> letter;
                break;
                }
            } while (true);
            break;
         case '4':
            clear();
            string data;
            for (int i = 0; i < 4000; ++i) {
               data+= tt[i].toString();
            }
            cout << printHuffman(data)<<endl;
            cout<<"Enter any letter to get into the menu"<<endl;
cin >> latter;
            break;

      }
      if (letter == 'q') break;
   } while (true);
}
int main() {
   FILE *fp;
   fp = fopen("testBase1.dat", "rb");
   fread(&ttNo, sizeof(record), 4000, fp);
```

```
        for (int i = 0; i < 4000; i++)
      tt[i] = ttNo[i];
   menu();

   return 0;
}
```

6. РЕЗУЛЬТАТЫ

Рисунок 1. Неотсортированная база данных.

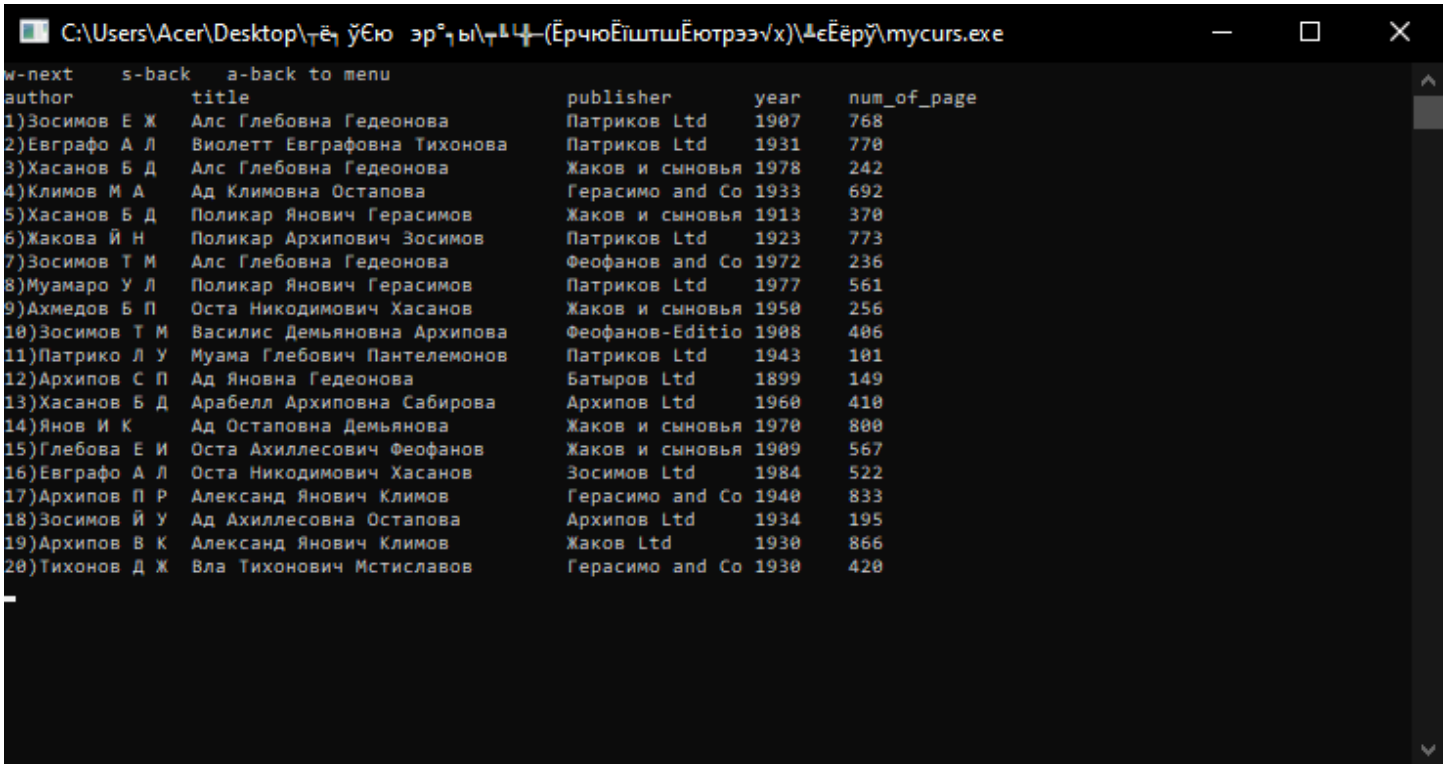


Рисунок 2. Отсортированная по издателю и автору база данных.

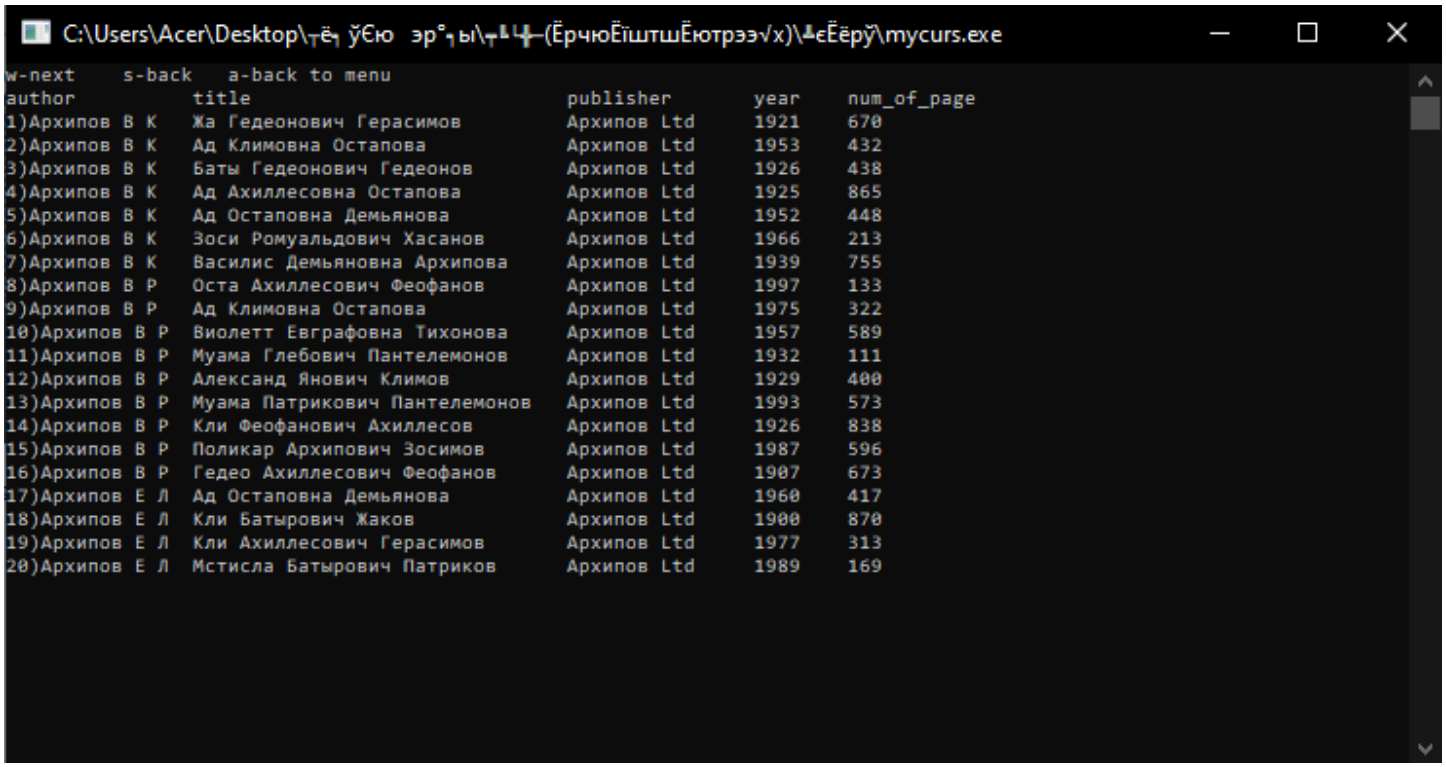


Рисунок 3. Поиск по издательству.

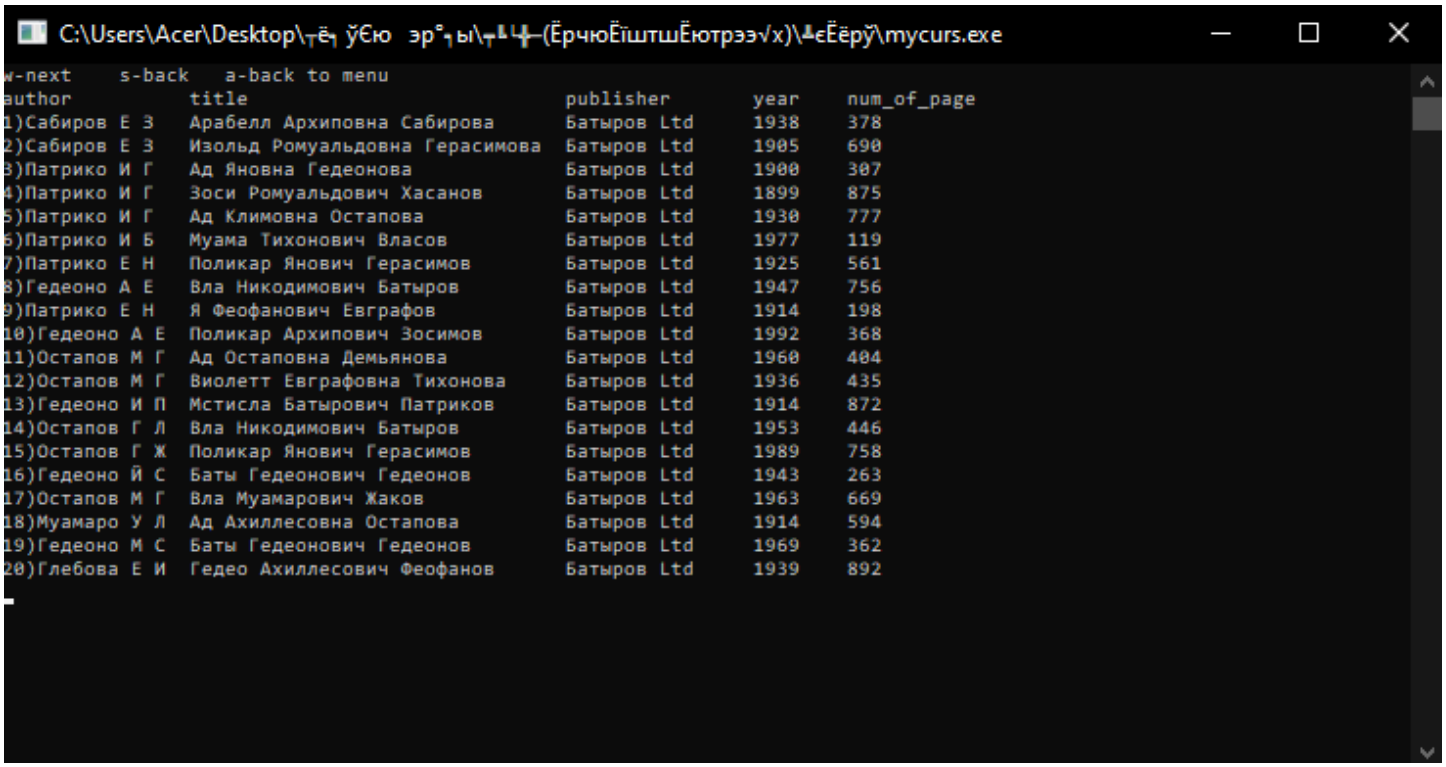


Рисунок 4. Поиск по дереву .

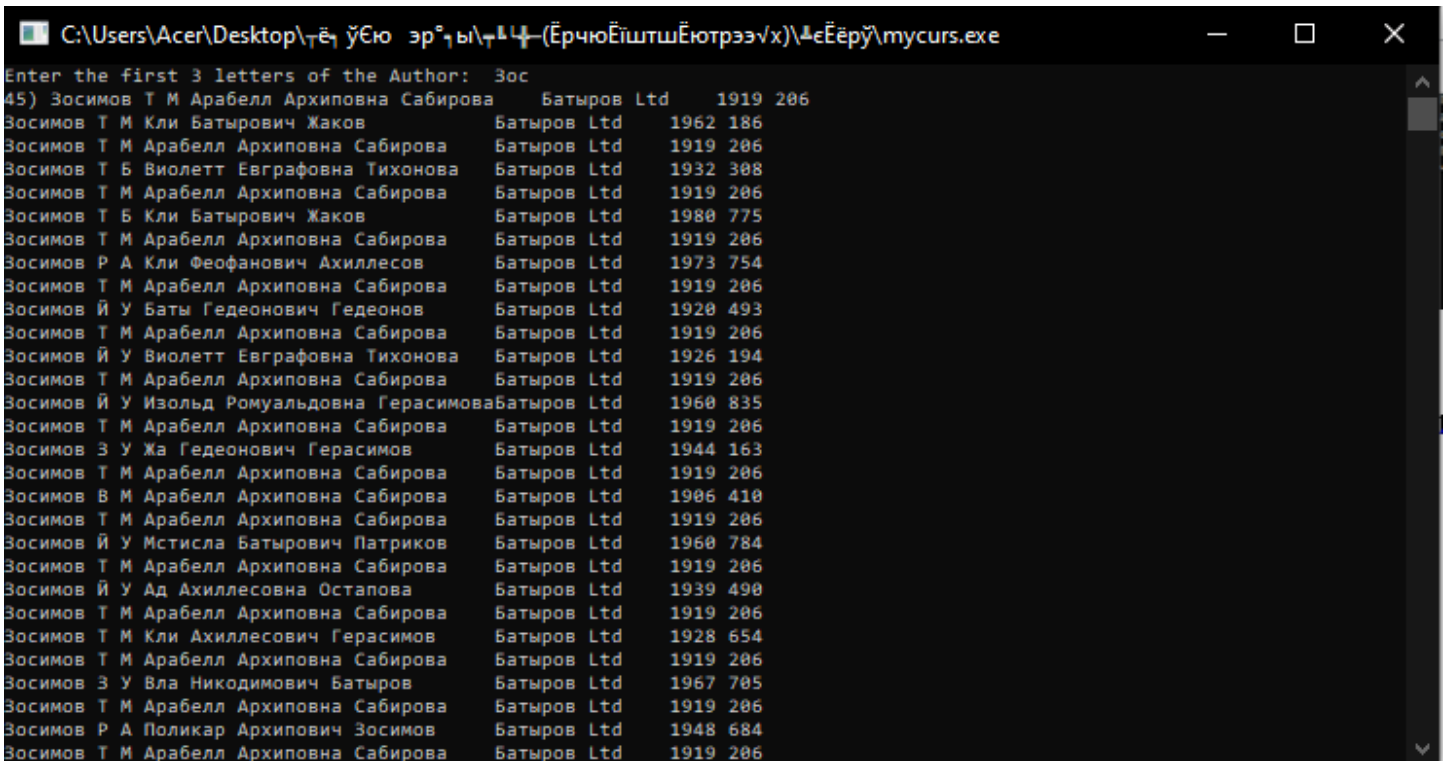
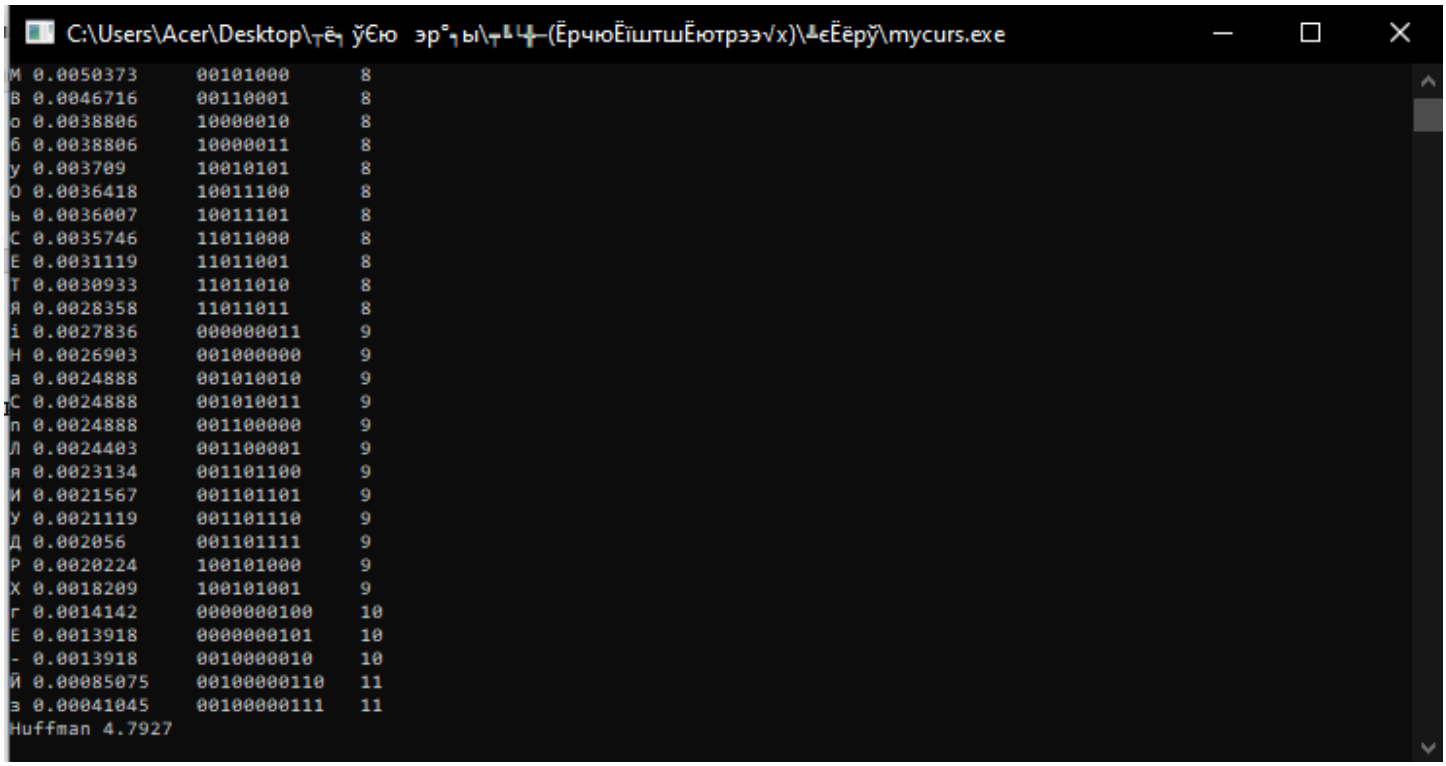


Рисунок 6. Примеры кодовых слов, энтропия.



7. ВЫВОДЫ

В ходе выполнения курсовой работы были выполнены все поставленные задачи и реализованы необходимые алгоритмы: сортировки, поиска, построения Авл-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 ($Lcp \leq H + 2$).

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.