МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: КОДИРОВАНИЕ И ДЕКОДИРОВАНИЕ, БДП, ХЕШ-ТАБЛИЦЫ, СОРТИРОВКИ

Студент гр. 9382 _____ Кодуков А.В.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

ЗАДАНИЕ
НА КУРСОВУЮ РАБОТУ

Студент Кодуков А.В.

Группа 9382

Тема работы (проекта):
Индивидуальное задание 1:
Статическое кодирование и декодирование текстового
файла методами Хаффмана и Фано-Шеннона – демонстрация

Исходные данные:
Пользователь задает программе файл, который будет закодирован и декодирован выбранным алгоритмом

Содержание пояснительной записки:
«Содержание», «Введение», «Ход выполнения работы», «Заключение», «Список использованных источников»

Предполагаемый объем пояснительной записки:
Не менее 00 страниц.

Дата выдачи задания: 31.10.2020

Дата сдачи реферата: 06.12.2020

Дата защиты реферата:

Студент _____ Кодуков А.В.

Преподаватель _____ Фирсов М.А.

АННОТАЦИЯ

Курсовая работа представляет собой демонстрационную реализацию кодировок Фано-Шеннона и Хаффмана. Пользователь вводит название файла и выбирает алгоритм кодировки. После этого выбранный файл кодируется и декодируется с подробным выводом работы алгоритма. Кодированный и декодированный файл, а также дополнительный вывод сохраняются в отдельные файлы.

SUMMARY

The course work is a demo implementation of Fano-Shannon and Huffman encodings. The user enters the name of the file and selects the encoding algorithm. After that, the selected file is encoded and decoded with detailed output of the algorithm. The encoded and decoded file and additional output are saved as separate files.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью работы является изучение и реализация алгоритмов кодирования Хаффмана и Фано-Шеннона, а также реализация программного комплекса для работы с данными методами сжатия: работа с деревьями кодирования, сжатие, расшифровка. Так как алгоритмы имеют сходства в своей работе, то часть реализации была выделена в независимый модуль. Помимо исполнения самого алгоритма программа предоставляет пользователю подробное описание работы алгоритма на представленных ей данных с выводом всех промежуточных результатов и действий.

1. ОПИСАНИЕ АЛГОРИТМА

1.1. Общая часть

Алгоритмы используют избыточность сообщения, заключенную в неоднородном распределении частот символов. Также кодирования являются префиксными, то есть никакой код не может быть префиксом другого.

Для кодирования файла необходимо два прохода. Первый необходим для подсчета частот всех символов. Затем символы сортируются по убыванию частоты встречаемости. Далее необходимо построить дерево кодирования (эта часть будет описана для каждого алгоритма отдельно). Теперь если обозначить шаг влево по дереву как ‘0’, а вправо как ‘1’, то можно составить коды всех символов как путь до них по дереву. Так как символы хранятся исключительно в листьях, то никакой код не может быть префиксом другого кода. Далее закодированный текст разбивается по 8 бит и записывается в файл.

Для расшифровки закодированного файла необходимо считать статистические данные и восстановить по ним дерево кодирования. Затем, получая из зашифрованного файла по 1 биту, алгоритм совершает шаг по дереву, в зависимости от значения бита, начиная с корня дерева. В какой-то момент следующий шаг приведет в концевую вершину дерева, которая соответствует символу. Символ записывается в файл, а алгоритм продолжает работу, вернувшись в корень дерева. Чтобы избежать проблем с интерпретацией последнего байта, дешифровщик работает в цикле, пока не запишет нужное количество символов, число которых известно из статистических данных в зашифрованном файле.

1.2. Алгоритм Фано-Шеннона

Для построения дерева массив символов делится пополам таким образом, чтобы обе части имели примерно одинаковую частоту. Эти части будут левым и правым поддеревом текущего узла. Такое разбиение повторяется

до тех пор, пока не дойдет до отдельных символов. Таким образом, символ всегда является конечным листом дерева

1.3. Алгоритм Хаффмана

Из отсортированного по частоте массива символов создается лес, в котором каждое дерево состоит из одного узла, включающего в себя символ и его частоту встречаемости. Далее на каждом этапе листья с двумя минимальными суммарными частотами удаляются из списка и становятся дочерними для нового элемента леса, статистика которого считается как сумма дочерних. Этот узел вставляется в лес так, чтобы сохранить упорядоченность по суммарной частоте элементов леса. Данная итерация повторяется пока в списке не останется один элемент. Таким образом, единственный оставшийся элемент является бинарным деревом, содержащим в себе все символы из файла.

2. ОПИСАНИЕ СТРУКТУР ДАННЫХ И ФУНКЦИЙ

2.1. Структуры данных

Элемент дерева:

```
typedef std::pair<std::string, long> Elem;
```

Бинарное дерево:

```
class Tree {
 public:
  // Tree node structure
  struct node {
    Elem info;      // Node data
    Tree *lt, *rt;  // Node childs

    …

  }
 private:
  node *Node = nullptr;  // Tree root

 …
}
```

Символы и частоты:

`typedef std::map<unsigned char, long> ElemMap;` - map для быстрого поиска элемента при сборе статистики

`typedef std::vector<std::pair<unsigned char, long>> ElemArr;` - вектор для сортировки по частоте

Код символа:

```
struct CODE {
  bool bits[50];
  int len = 0;
};
```

Лес:

```
typedef std::vector<Tree *> Forest;
```

2.2. Функции

Построение новых кодов символов:

Сигнатура: `void buildCodes(Tree *T)`

Аргументы:

- T –дерево кодирования

Алгоритм:

- Левое и правое поддерево пусты – алгоритм дошел до символа. Записать накопившийся код в список.
- Левое поддерево не пусто – увеличить текущую длину кода, записать в текущйи код ‘0’, запустить функцию от левого поддерева.

- Правое поддерево – аналог. с записью ‘1’.

Построение дерева кодирования Фано-Шеннона:

Сигнатура: `Tree *buildCodeTreeFano(ElemArr CurFreq, bool output)`

Аргументы:

- CurFreq – текущий набор символов и их частот
- output – флаг дополнительного вывода

Алгоритм:

- Массив пуст – вернуть пустой узел
- Массив содержит один символ – вернуть узел, построенный из этого элемента
- Посчитать среднюю частоту символов
- Разбить элементы массивы на две примерно равных по частоте части
- Найти левое и правое поддерево как результат работы функции для первой и второй части получившегося разбиения.
- Заполнить и вернуть узел

Итерация алгоритма Хаффмана:

Сигнатура: `void HuffmanIter(Forest &forest, bool output)`

Аргументы:

- forest – текущий лес для построения дерева кодирования
- output – флаг дополнительного вывода

Алгоритм:

- Создать новый узел с частотой равной сумме двух последних элементов в отсортированном лесу
- Удалить два последних элемента из леса и присвоить их в левое и правое поддерево нового узла
- Вставить новый элемент в лес так, чтобы не нарушить его отсортированность

Построение дерева кодирования Хаффмана:

Сигнатура: `Tree *buildCodeTreeHuffman(ElemArr CurFreq, bool output)`

Аргументы:

- CurFreq – текущий набор символов и их частот
- output – флаг дополнительного вывода

Алгоритм:

- Создать лес из отсортированного по частоте массива элементов
- Пока размер леса не станет равным одному дереву, запускать итерацию алгоритма (см. функцию HuffmanIter)
- Вернуть первый элемент леса

Сжатие файла

Сигнатура:

```
bool press(const char *filename , Tree *(*buildCodeTree)(ElemArr curFreq, bool output))
```

Возвращаемое значение:

(bool) – Успешно ли кодирование

Аргументы:

- filename – имя файла
- buildCodeTree – указатель на функцию построения дерева

Алгоритм:

- Посчитать частоты символов
- Отсортировать символы по убыванию частоты
- Построить дерево кодирования
- Построить коды
- Вернуть файл в начало
- Записать метку
- Записать частоты
- Кодировать данные файлы, накапливая биты в специальном аккумуляторе

Разжатие файла

Сигнатура: `bool depress(Tree *(*buildCodeTree)(ElemArr curFreq, bool output))`

Возвращаемое значение:

(bool) – Успешно ли декодирование

Аргументы:

- buildCodeTree – указатель на функцию построения дерева

Алгоритм:

- Проверить метку
- Считать частоты
- Построить дерево кодирования
- Побитово считывать закодированный файл и восстанавливать данные по дереву кодирования

3. ОПИСАНИЕ РАБОТЫ ПРОГРАММЫ

Программа запрашивает у пользователя имя файла и алгоритм кодирования. Затем по ходу работы программы выводится на экран и в файл промежуточная информация о работе алгоритма, а именно:

- Статистическую информацию, которая была посчитана за первый проход по файлу (частоты встречаемости символов)
- Шаги по построению кодового дерева выбранным алгоритмом
- Итоговое дерево кодирования. Так как во время работы алгоритма не происходит какого-либо перемещения узлов внутри дерева, то итогового дерева достаточно, чтобы однозначно отследить ход работы алгоритма построения дерева
- Новые двоичные коды символов, полученные обходом дерева (‘0’ – лево, ‘1’ – право)
- Закодированные данные. Отдельные символы исходного текста разделены пробелами.
- Размер исходного файла
- Размер закодированного текста без учета статистических данных, которые будут записаны в файл
- Восстановленное при декодировании дерево (для сравнения с оригинальным)
- Шаги декодирования

4. ТЕСТИРОВАНИЕ

Тестирование:

№	Входные данные	Результат (коды и декодированный файл)				
		Частоты	Коды		Результат кодирования	
			Фано	Хафф.	Фано	Хафф.
1	a	a - 1	a: 1	a: 1	1	1
2	aaaaaaaaa	a - 9	a:1	a:1	1 1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1 1
3	aa bbb cccc ddddd	d - 5; c - 4; - 3; b - 3; a - 2;	d:00 c:01 :10 b:111 a:111	d:01 c:10 :11 a:001 b:000	111 111 10 110 110 110 10 01 01 01 01 10 00 00 00 00 00	001 001 11 000 000 000 11 10 10 10 10 11 01 01 01 01 01
4	happy new year	-2; a-2; e-2; p-2; y-2; h-1; n-1; r-1; w-1;	:00 a:010 e:011 p:100 y:101 h:1100 n:1101 r:1110 w:1111	:001 a:010 e:011 p:100 y:101 h:110 n:111 r:0000 w:0001	1100 010 100 100 101 00 1101 011 1111 00 101 011 010 1110	110 010 100 100 101 001 111 011 0001 001 101 011 010 0000
5	abcdefghijklmnop qrstuvwxyz	a-1; b-1; c-1; d-1; e-1; f-1; g-1; h-1; i-1; j-1; k-1; l-1; m-1; n-1; o-1; p-1; q-1; r-1; s-1; t-1; u-1; v-1; w-1; x-1;	a:0000 b:00010 c:00011 d:0010 e:00110 f:00111 g:0100 h:01010 i:01011 j:01100 k:01101 l:01110 m:01111 n:1000 o:10010 p:10011 q:1010 r:10110 s:10111 t:1100 u:11010 v:11011 w:11100 x:11101	a:1010 b:1011 c:1100 d:1101 e:1110 f:1111 g:00000 h:00001 i:00010 j:00011 k:00100 l:00101 m:00110 n:00111 o:01000 p:01001 q:01010 r:01011 s:01100 t:01101 u:01110 v:01111 w:10000 x:10001	0000 00010 00011 0010 00110 00111 0100 01010 01011 01100 01101 01110 01111 1000 10010 10011 1010 10110 10111 1100 11010 11011 11100 11101 11110 11111	1010 1011 1100 1101 1110 1111 00000 00001 00010 00011 00100 00101 00110 00111 01000 01001 01010 01011 01100 01101 01110 01111 10000 10001 10010 10011

		y-1; z-1;	y:11110 z:11111	y:10010 z:10011		
6	WAR IS PEACE FREEDOM IS SLAVERY IGNORANCE IS STRENGTH	-9; E-7; R-5; S-5; A-4; I-4; N-3; C-2; G-2; O-2; T-2; D-1; F-1; H-1; L-1; M-1; P-1; V-1; W-1; Y-1;	:000 E:001 R:010 S:011 A:100 I:1010 N:10110 C:10111 G:1100 O:11010 T:110110 D:110111 F:111000 H:111001 L:111010 M:111011 P:111100 V:111101 W:111110 Y:111111	:010 E:100 R:110 S:111 A:0011 I:0110 N:1010 C:00011 G:00100 O:00101 T:01110 D:01111 F:10110 H:10111 L:000000 M:000001 P:000010 V:000011 W:000100 Y:000101	111110 100 010 000 1010 011 000 111100 001 100 10111 001 000 111000 010 001 001 110111 11010 111011 000 1010 011 000 011 111010 100 111101 001 010 111111 000 1010 1100 10110 11010 010 100 10110 10111 001 000 1010 011 000 011 110110 010 001 10110 1100 110110 111001 000	000100 0011 110 010 0110 111 010 000010 100 0011 00011 100 010 10110 110 100 100 01111 00101 000001 010 0110 111 010 111 000000 0011 000011 100 110 000101 010 0110 00100 1010 00101 110 0011 1010 00011 100 010 0110 111 010 111 01110 110 100 1010 00100 01110 10111
7	Следование единому стилю форматирования исходного кода на протяжении всего исходного кода программы	о-15; -11; и-9; а-7; н-7; д-6; е-5; р-5; г-4; м-4; с-4; в-3; т-3; к-2; л-2; п-2; х-2; я-2; С-1; ж-1; у-1; ф-1; ы-1; ю-1;	о:000 :001 и:010 а:0110 н:0111 д:1000 е:1001 р:1010 г:10110 м:10111 с:11000 в:11001 т:11010 к:11011 л:11100 п:111010 х:111011 я:111100 С:1111010 ж:1111011 у:1111100 ф:1111101 ы:1111110 ю:1111111	о:010 :101 и:111 а:0011 н:0110 д:1000 е:1001 р:1100 г:00011 м:00100 с:00101 в:01110 т:01111 к:000001 л:000010 п:000011 х:000100 я:000101 С:110100 ж:110101 у:110110 ф:110111 ы:0000000 ю:0000001	1111010 11100 1001 1000 000 11001 0110 0111 010 1001 001 1001 1000 010 0111 000 10111 1111100 001 11000 11010 010 11100 1111111 001 1111101 000 1010 10111 0110 11010 010 1010 000 11001 0110 0111 010 111100 001 010 11000 111011 000 1000 0111 000 10110 000 001 11011 000 1000 0110 001 0111 0110 001 111010 1010 000 11010 111100 1111011 1001 0111 010 010 001 11001 11000 1001 10110 000 001 010 11000 111011 000 1000 0111 000 10110	110100 000010 1001 1000 010 01110 0011 0110 111 1001 101 1001 1000 111 0110 010 00100 110110 101 00101 01111 111 000010 0000001 101 110111 010 1100 00100 0011 01111 111 1100 010 01110 0011 0110 111 000101 101 111 00101 000100 010 1000 0110 010 00011 010 101 000001 010 1000 0011 101 0110 0011 101 000011 1100 010 01111 000101 110101 1001 0110 111 111 101 01110 00101 1001 00011 010 101 111 00101 000100 010 1000 0110 010 00011

					000 001 11011 000 1000 0110 001 111010 1010 000 10110 1010 0110 10111 10111 1111110	010 101 000001 010 1000 0011 101 000011 1100 010 00011 1100 0011 00100 00100 0000000

ЗАКЛЮЧЕНИЕ

В ходе выполнения работы были изучены и реализованы на практике алгоритмы кодирования Фано-Шеннона и Хаффмана. Разработана программа, позволяющая сжимать и расшифровывать файлы, предоставляющая пользователю полное описание работы алгоритма. В ходе работы были выявлены особенности данных алгоритмов:

1) неэффективность на небольших файлах из-за дополнительных данных, записываемых в сжатый файл
2) необходимо два прохода по файлу
3) метод Хаффмана в большинстве случаев не выигрывает по длине сжатого файла у метода Фано-Шеннона. Дерево Хаффмана является одним из деревьев Фано-Шеннона

ПРИЛОЖЕНИЕ А
КОД ПРОГРАММЫ

def.h

```
#ifndef DEF_H_
#define DEF_H_

#include <iostream>
#include <fstream>
#include <vector>
#include <map>
#include <cstring>
#include <set>
#include <cstdlib>
#include <queue>

typedef std::pair<std::string, long> Elem;

class Tree {
 public:
  // Tree node structure
  struct node {
    Elem info; // Node data
    Tree *lt, *rt;  // Node childs

    node() {
      lt = nullptr;
      rt = nullptr;
    }

    node(const Elem &x, Tree *lst, Tree *rst) {
      info = x;
      lt = lst;
      rt = rst;
    }
  };

 private:
  node *Node = nullptr;  // Tree root

 public:
  Tree() {}

  Tree(node *N) { Node = N; }

  // Tree memory clear function
  void clear() {
    if (Node != nullptr) {
      if (Node->lt != nullptr) Node->lt->clear();
      if (Node->rt != nullptr) Node->rt->clear();
      delete Node;
      Node = nullptr;
    }
  }

  ~Tree() { clear(); }

  // left child getting function
  Tree *left() {
```

```
    if (Node == nullptr) {  // No node
      std::cout << "Error: left(null) \n";
      exit(1);
    } else
      return Node->lt;
  }

  // right child getting function
  Tree *right() {
    if (Node == nullptr) {
      std::cout << "Error: right(null) \n";
      exit(1);
    } else
      return Node->rt;
  }
  // Node pointer getting function
  node *nodePtr() {
    if (Node == nullptr) {
      std::cout << "Error: RootBT(null) \n";
      exit(1);
    } else
      return Node;
  }
  // Node data getting function
  Elem getNode() {
    if (Node == nullptr) {  // No node
      std::cout << "Error: RootBT(null) \n";
      exit(1);
    } else
      return Node->info;
  }
};

// Types
typedef std::map<unsigned char, long> ElemMap;
typedef std::vector<std::pair<unsigned char, long>> ElemArr;

#endif // DEF_H_
```

code.h

```
      #ifndef CODE_H_
      #define CODE_H_

      #include "def.h"

      bool press(const char *filename,
                 Tree    *(*buildCodeTree)(ElemArr    curFreq,    bool    output,
std::ofstream &info));
      bool   depress(Tree   *(*buildCodeTree)(ElemArr   curFreq,   bool   output,
std::ofstream &info));

#endif // CODE_H_
```

code.cpp

```
      #include <map>
```

```
#include "code.h"

// Tree printing function
void print(Tree *q, long n, std::ofstream &f) {
  long i;
  if (q != nullptr) {
    print(q->right(), n + q->getNode().first.size() + 2, f);
    for (i = 0; i < n; i++) {
      std::cout << " ";
      f << " ";
    }
    std::cout << "\\" << q->getNode().first << "\\\n";
    f << "\\" << q->getNode().first << "\\\n";
    print(q->left(), n + q->getNode().first.size() + 2, f);
  }
}

// Code building definitions
struct Code {
  bool bits[50];
  int len = 0;
};
Code curCode;
std::map<unsigned char, Code> newCodes;

void buildCodes(Tree *T) {
  // No subtrees -> symbol found
  if (T->left() == nullptr && T->right() == nullptr) {
    unsigned char ch = T->getNode().first[0];
    if (curCode.len == 0) {
      curCode.bits[0] = 1;
      curCode.len = 1;
    }
    newCodes.insert({ch, curCode});
    return;
  }
  // left subtree (0 to code)
  if (T->left() != NULL) {
    curCode.bits[curCode.len] = 0;
    curCode.len++;
    buildCodes(T->left());
    curCode.len--;
  }
  // right subtree (1 to code)
  if (T->right() != NULL) {
    curCode.bits[curCode.len] = 1;
    curCode.len++;
    buildCodes(T->right());
```

```
      curCode.len--;
    }
  }

  int comp(const std::pair<unsigned char, long> *i,
           const std::pair<unsigned char, long> *j) {
    if (i->second == j->second) return i->first - j->first;
    return j->second - i->second;
  }

  bool press(const char *filename, Tree * (*buildCodeTree)(ElemArr curFreq,
bool output, std::ofstream &info)) {
    int ch;
    long long size1 = 0, size2 = 0;
    ElemMap freq;
    bool output = false;

    std::cout << "Pressing file " << filename << "\n";
    setlocale(LC_CTYPE, ".1251");
    std::ifstream infile(filename, std::ios::in);
    if (!infile.is_open()) {
      std::cout << "Impossible to open file\n";
      return false;
    }
    std::ofstream outfile, info;
    // Counting Frequensies
    while ((ch = infile.get()) != EOF) {
      size1++;
      auto iter = freq.find(ch);
      if (iter != freq.end())
        (*iter).second++;
      else
        freq.insert({ch, 1});
    }
    // Sorting frequencies
    ElemArr curFreq(freq.begin(), freq.end());
    std::qsort(curFreq.data(), curFreq.size(),
               sizeof(std::pair<unsigned char, long>),
               (int (*)(const void *, const void *))comp);

    outfile.open("Files/pressed.txt", std::ios::binary);

    info.open("Files/info.txt");
    output = size1 < 100;

    // Write label
    outfile << "CD!";
    outfile << (int)freq.size();
    if (output) {
```

```
  std::cout << "Frequencies:\n";
  info << "Frequencies:\n";
}
// Write frequencies
for (auto &i : curFreq) {
  if (output) {
    std::cout << (unsigned char)i.first << "-" << i.second << ";";
    info << (unsigned char)i.first << "-" << i.second << ";";
  }
  outfile << i.first;
  outfile.write(reinterpret_cast<char *>(&i.second), sizeof(long));
}
if (output) {
  std::cout << "\n\n";
  info << "\n\n";
}
// Building code tree
if (output) {
  std::cout << "Building code tree: \n";
  info << "Building code tree: \n";
}
Tree *T = buildCodeTree(curFreq, output, info);
if (output) {
  std::cout << "\n";
  info << "\n";
}
// Building new codes
buildCodes(T);

// Printing tree

if (output) {
  std::cout << "Tree:\n";
  info << "Tree:\n";
  print(T, 0, info);
  std::cout << "\n";
  info << "\n";
}
T->clear();
// Printing codes
std::cout << "Codes:\n";
info << "Codes:\n";
for (auto &c : newCodes) {
  std::cout << c.first << ":";
  info << c.first << ":";
  for (int i = 0; i < c.second.len; i++) {
    std::cout << (int)c.second.bits[i];
    info << (int)c.second.bits[i];
  }
```

```
  std::cout << "\n";
  info << "\n";
}
std::cout << "\n";
info << "\n";

/*** Passing through the file second time ***/
// Coding input data to output file
infile.seekg(infile.beg);
unsigned char bitAccum = 0;
int bitPos = 7;
infile.close();
infile.open(filename);
if (output) {
  std::cout << "Coded data: \n";
  info << "Coded data: \n";
}
while ((ch = infile.get()) != EOF) {
  for (int k = 0; k < newCodes[ch].len; k++) {
    int bit = newCodes[ch].bits[k] << bitPos--;
    if (output) {
      std::cout << (int)newCodes[ch].bits[k];
      info << (int)newCodes[ch].bits[k];
    }
    bitAccum |= bit;
    // Writing byte
    if (bitPos < 0) {
      size2++;
      outfile << bitAccum;
      bitAccum = 0;
      bitPos = 7;
    }
  }
  if (output) {
    std::cout << " ";
    info << " ";
  }
}
if (output) {
  std::cout << "\n";
  info << "\n";
}
if (bitPos < 7) outfile << bitAccum, size2++;
std::cout << "Input size: " << size1
          << "\nPressed size(pure input data): " << size2 << "\n";
info << "Input size: " << size1
          << "\nPressed size(pure input data): " << size2 << "\n";

info.close();
```

```
    infile.close();
    outfile.close();
    return true;
  }

  // Decompress function
  bool depress(Tree *(*buildCodeTree)(ElemArr curFreq, bool output,
std::ofstream &info)) {
    std::ifstream infile("Files/pressed.txt", std::ios::binary);
    std::ofstream outfile, info("Files/info.txt", std::ios::app);
    if (!infile.is_open()) {
      std::cout << "Impossible to open file\n";
      return false;
    }
    std::cout << "Decompressing file pressed.txt\n";
    info << "Decompressing file pressed.txt\n";
    setlocale(LC_ALL, "Russian");
    // Check label
    char label[4];
    infile.read(label, 3);
    label[3] = '\0';
    if (strcmp(label, "CD!") != 0) {
      infile.close();
      std::cout << "Wrong pressed file\n";
      return false;
    }
    // Read frequencies
    int cnt;
    long long size = 0;
    infile >> cnt;
    ElemMap freq;
    for (int i = 0; i < cnt; i++) {
      unsigned char ch;
      unsigned long num = 0;
      ch = infile.get();
      unsigned char numstr[4];
      for (int i = 0; i < 4; i++) numstr[3 - i] = infile.get();
      for (int i = 0; i < 4; i++) {
        num <<= 8;
        num |= numstr[i];
      }
      freq.insert({ch, num});
      size += num;
    }
    bool output = size < 100;
    // Sort frequencies
    Tree *tree, *start;
    ElemArr curFreq(freq.begin(), freq.end());
    std::qsort(curFreq.data(), curFreq.size(), sizeof(curFreq[0]),
```

```
            (int (*)(const void *, const void *))comp);
// Building code tree
tree = buildCodeTree(curFreq, false, info);

if (output) {
  std::cout << "Code tree:\n";
  info << "Code tree:\n";
  print(tree, 0, info);
  std::cout << "\n";
  info << "\n";
}
start = tree;
// Reading coded data
int ch, bitPos = -1, res = 0;
unsigned char bitAccum;
bool isfirst = true, isstart = true;
long num = 0;
outfile.open("Files/decompressed.txt");
if (output) {
  std::cout << "Decoding data: (0 - left, 1 - right)\n";
  info << "Decoding data: (0 - left, 1 - right)\n";
}
while (1) {
  // Leaf -> symbol
  if (!isfirst && tree->left() == NULL) {
    if (isstart && !res) break;
    unsigned char ch = (unsigned char)tree->getNode().first[0];
    if (output) {
      std::cout << "Symbol: " << ch << "\n";
      info << "Symbol: " << ch << "\n";
    }
    outfile << ch;
    num++;
    if (num == size) break;
    tree = start;
    isstart = true;
  }
  if (isfirst) isfirst = false;
  // Get new byte
  if (bitPos < 0) {
    ch = infile.get();
    if (ch == EOF) break;
    bitAccum = ch;
    bitPos = 7;
  }
  // 0 - go left, 1 - go right
  res = (bitAccum >> bitPos--) & 1;
  if (res && (tree->right() != nullptr)) {
    tree = tree->right();
```

```
            if (output) {
              std::cout << "1 -> /" << tree->getNode().first << "/; ";
              info << "1 -> /" << tree->getNode().first << "/; ";
            }
            isstart = false;
          } else if (tree->left() != nullptr) {
            isstart = false;
            tree = tree->left();
            if (output) {
              std::cout << "0 -> /" << tree->getNode().first << "/; ";
              info << "0 -> /" << tree->getNode().first << "/; ";
            }
          }
        }
        info.close();
        infile.close();
        outfile.close();

        return true;
}
```

fanohuffman.h

```
#ifndef FANOHUFFMAN_H_
#define FANOHUFFMAN_H_

#include "def.h"

Tree *buildCodeTreeFano(ElemArr curFreq, bool output, std::ofstream &info);
Tree *buildCodeTreeHuffman(ElemArr curFreq, bool output, std::ofstream &info);

#endif //FANOHUFFMAN_H_
```

fanohuffman.cpp

```
#include <cmath>
#include "fanohuffman.h"


Tree *buildCodeTreeFano(ElemArr curFreq, bool output, std::ofstream &info) {
        if (output) {
          std::cout << "Symbols: ";
          info << "Symbols: ";
        }
        // Quit recursion
        if (curFreq.size() == 0) {
          if (output) {
            std::cout << "No symbol -> empty node\n";
            info << "No symbol -> empty node\n";
          }
          return nullptr;
        }
        // Symbol leaf case
        if (curFreq.size() == 1) {
          std::string s;
```

```
          s.push_back(curFreq.begin()->first);
          if (output) {
            std::cout << s << " One symbol -> node {" << s << ", nullptr,
nullptr}" << "\n";
            info << s << " One symbol -> node {" << s << ", nullptr, nullptr}"
<< "\n";
          }
          return new Tree(new Tree::node({s, curFreq[0].second}, nullptr,
nullptr));
        }
        // Count average frequency
        long sum = 0;
        std::string nodestr;
        for (auto &f : curFreq) {
          nodestr.push_back(f.first);
          sum += f.second;
        }
        long avg = sum / 2;
        if (output) {
          std::cout << "/" << nodestr << "/ Sum: " << sum << " Average: " << avg
<< "\n";
          info << "/" << nodestr << "/ Sum: " << sum << " Average: " << avg <<
"\n";
        }
        // Splitting current array by frequency
        long cursum = 0, last = sum;
        auto iter = curFreq.begin();
        int strcnt = 0;
        while (last > std::abs(cursum + (iter->second) - avg)) {
          cursum += iter->second;
          last = std::abs(cursum - avg);
          iter++;
          strcnt++;
        }
        if (output) {
          std::cout << "  left: " << nodestr.substr(0, strcnt) << "(" << cursum
<< ")\n"
                    << "   right: " << nodestr.substr(strcnt) << "(" << sum -
cursum << ")\n";
          info << "  left: " << nodestr.substr(0, strcnt) << "(" << cursum <<
")\n"
               << "  right: " << nodestr.substr(strcnt) << "(" << sum - cursum
<< ")\n";
        }
        // Building left and right subtree
        ElemArr left(curFreq.begin(), iter);
        ElemArr right(iter, curFreq.end());
        return new Tree(new Tree::node({nodestr, cursum},
buildCodeTreeFano(left, output, info),
```

```
                                        buildCodeTreeFano(right, output, info)));
    }

    typedef std::vector<Tree *> Forest;

    void HuffmanIter(Forest &forest, bool output, std::ofstream &info) {
      // Printing forest
      if (output) {
        std::cout << "Current forest:\n  ";
        info << "Current forest:\n  ";
        for (auto &e : forest) {
          std::cout << e->getNode().first << "(" << e->getNode().second << ")
";
          info << e->getNode().first << "(" << e->getNode().second << ") ";
        }
        info << "\n";
        std::cout << "\n";
      }

      long size = forest.size();

      // Initialize new node data
      Tree *last = forest[size - 1], *prelast = forest[size - 2];
      long sum = last->getNode().second + prelast->getNode().second;
      std::string nodestr = last->getNode().first + prelast->getNode().first;

      // Printing new node
      if (output) {
        std::cout << "  Sum nodes: " << prelast->getNode().first << "("
                  <<  prelast->getNode().second  <<  ")  and  "  <<  last-
>getNode().first
                  << "(" << last->getNode().second << ")\n";
        std::cout << "  Result: " << nodestr << "(" << sum << ")\n";
        info << "  Sum nodes: " << prelast->getNode().first << "("
                  <<  prelast->getNode().second  <<  ")  and  "  <<  last-
>getNode().first
                  << "(" << last->getNode().second << ")\n";
        info << "  Result: " << nodestr << "(" << sum << ")\n";
      }

      // Creating new node
      Tree::node *newNode = new Tree::node({nodestr, sum}, prelast, last);
      forest.pop_back(), forest.pop_back();
      if (forest.size() == 0) {
        forest.push_back(new Tree(newNode));
        return;
      }
      auto iter = forest.end();
      iter--;
```

```
        while ((*iter)->getNode().second <= newNode->info.second) {
          if (iter == forest.begin()) break;
          iter--;
        }
        if    ((*(forest.begin()))->getNode().second    >    newNode->info.second)
iter++;

        // Inserting new node in forest
        forest.insert(iter, new Tree(newNode));
      }

      // Building code tree function
      Tree  *buildCodeTreeHuffman(ElemArr  curFreq,  bool  output,  std::ofstream
&info) {
        Forest forest;

        for (auto &e : curFreq) {
          std::string nodestr;
          char c = e.first;
          nodestr.push_back(c);
          Tree  *t  =  new  Tree(new  Tree::node({nodestr,  e.second},  nullptr,
nullptr));
          forest.push_back(t);
        }
        while (forest.size() > 1)
          HuffmanIter(forest, output, info);
        return forest[0];
}
```

main.cpp

```
      #include <limits>

      #include "code.h"
      #include "fanohuffman.h"

      int main() {
        std::string fname;
        auto buildFunc = buildCodeTreeFano;

        std::cout << "Input filename: ";
        std::cin >> fname;
        while (1) {
          char mode;
          std::cin.clear();
          std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
          system("cls");
          std::cout << "Choose mode:\n1 - fano\n2 - huffman\n";
          mode = getchar();
```

```
            if (mode == '1') {
              buildFunc = buildCodeTreeFano;
              break;
            } else if (mode == '2') {
              buildFunc = buildCodeTreeHuffman;
              break;
            } else {
              std::cout << "Wrong option";
              getchar();
            }
          }
          if (press(fname.data(), buildFunc)) {
            std::cout << "Compression complete!\n";
            if (depress(buildFunc)) std::cout << "Decompression complete!\n";
            else
              std::cout << "Decompression error\n";
          } else
            std::cout << "Compression error\n";
          system("pause");
}
```