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: Статическое кодирование и декодирование текстового файла методами Хаффмана и Фано-Шеннона

Студент гр. 7383 _____ Александров Р.А.

Преподаватель _____ Размочаева Н.В.

Санкт-Петербург

2018

ЗАДАНИЕ
НА КУРСОВУЮ РАБОТУ

Студент Александров Р.А.

Группа 7383

Тема работы: Статическое кодирование и декодирование текстового файла методами Хаффмана и Фано-Шеннона

Исходные данные:
Реализовать программу на язык C++ с демонстрацией алгоритмов

Содержание пояснительной записки:

- Содержание
- Введение
- Формулировка задачи и теоретические данные
- Решение задачи
- Тестирование программы
- Заключение
- Приложение А. Исходный код программы

Предполагаемый объем пояснительной записки:
Не менее 10 страниц.

Дата выдачи задания:

Дата сдачи реферата:

Дата защиты реферата:

Студент _____ Александров Р.А.

Преподаватель _____ Размочаева Н.В.

АННОТАЦИЯ

В ходе работы была разработана программа, которая осуществляет статическое кодирование и декодирование текстового файла методами Фано-Шеннона и Хафммана. Для наглядности была создана демонстрация работы алгоритмов.

SUMMARY

In the work a program, performed static coding and decoding a textfile by Shannon-Fano and Huffman algorithms, was developed. For illustrative purposes work a work demonstration was implemented.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель работы: освоение алгоритмов префиксного кодирования и декодирования.

Основные задачи: программная реализация статических методов Фано-Шеннона и Хаффмана.

Тестирование работы программы будет проводится в OS Windows 10.

1. ФОРМУЛИРОВКА ЗАДАЧИ И ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Необходимо реализовать демонстрацию работы статического кодирования и декодирования текстового файла методами Хаффмана и Фано-Шеннона.

Кодирование Шеннона — Фано – алгоритм префиксного неоднородного кодирования. Относится к вероятностным методам сжатия (точнее, методам контекстного моделирования нулевого порядка).

Основные этапы:

1. Символы первичного алфавита выписывают по убыванию вероятностей.
2. Символы полученного алфавита делят на две части, суммарные вероятности символов которых максимально близки друг другу.
3. В префиксном коде для первой части алфавита присваивается двоичная цифра «0», второй части — «1».
4. Полученные части рекурсивно делятся и их частям назначаются соответствующие двоичные цифры в префиксном коде.

Кодирование Хаффмана – жадный алгоритм оптимального префиксного кодирования алфавита с минимальной избыточностью.

Основные этапы:

1. Символы входного алфавита образуют список свободных узлов. Каждый лист имеет вес, который может быть равен либо вероятности, либо количеству вхождений символа в сжимаемое сообщение.
2. Выбираются два свободных узла дерева с наименьшими весами.
3. Создается их родитель с весом, равным их суммарному весу.
4. Родитель добавляется в список свободных узлов, а два его потомка удаляются из этого списка.

5. Одной дуге, выходящей из родителя, ставится в соответствие бит 1, другой — бит 0. Битовые значения ветвей, исходящих от корня, не зависят от весов потомков.
6. Шаги, начиная со второго, повторяются до тех пор, пока в списке свободных узлов не останется только один свободный узел. Он и будет считаться корнем дерева.

2. РЕШЕНИЕ ЗАДАЧИ

Для решения данной задачи были реализован ряд методов, вызываемых из главного класса Main.

Класс Main предлагает пользователю выбор алгоритма и просит указать имя текстового файла, для этого используются следующие методы:

- void getHuffman();
- void getFanoShannon();
- string fileRead().

Данные символов хранит структура struct Symbol, где содержится char – считанный символ, и weight – его частота встречаемости в полученном тексте.

Структура struct CodeTree содержит struct Symbol и указатели на родителя и на правого и левого потомков.

Методы CodeTree *fanoShanon(const std::string &message) и CodeTree *huffmanCode(const std::string &message) преобразуют текст в бинарное дерево, а также через вспомогательные функции отображают информацию для каждого символа и способ построения дерева (дерево Хаффмана строится от потомков к родителям, а дерево Фано-Шеннона – от родителей к потомкам).

Метод char *encode(const CodeTree *tree, const std::string &message) преобразует полученное дерево и исходный текст в последовательность бинарного кода.

Метод char *decode(const CodeTree *tree, const char *code, vector< std::string> &rows) преобразует бинарную последовательность в итоговое сообщение и передает в вектор код для каждого символа.

Для метода Хаффмана используется очередь с приоритетом, которая помогает выбирать узлы с наименьшими весами. Структура struct PriorityQueueItem содержит как и struct Symbol содержит символ и его вес. А метод void siftUp(PriorityQueue<T> *pq, int index) производит сортировку в порядке убывания по этому весу.

Печать бинарного дерева осуществляется в уступчатом виде при помощи методов void displayTree(const CodeTree *b, int n) и вспомогательного void helpToDisplay(const CodeTree *b).

Полноценный исходный код содержится в приложении А.

3. ТЕСТИРОВАНИЕ ПРОГРАММЫ

Тестирование программы проводилось в OS Windows 10 с использованием компилятора MinGW. Файлы для тестирования на 21, 35000 и 357000 символов расположены в папке tests.

I. Алгоритм Фано-Шеннона.

На рис. 1 представлен основной этап – упорядочивание символов строки по убыванию вероятностей и процесс деления полученного алфавита на две части, суммарные вероятности символов которых близки друг к другу.

На рис. 2 представлен этап построения бинарного дерева в уступчатом виде. Дерево строится от родителя к потомкам, “сверху вниз”. Для левой части дерева, которая имеет большую сумму, присваивается 1, для правой – 0. Затем по дереву и исходному тексту формируется итоговая кодовая последовательность.

На рис. 3 происходит декодирование исходного сообщения при помощи кодовой последовательности и бинарного дерева, также выводится код каждого символа.

II. Алгоритм Хаффмана.

На рис. 4 представлен основной этап – упорядочивание символов строки по убыванию вероятностей, процесс выбора двух свободных узлов с наименьшими весами и преобразование их в единого родителя с суммированным весом.

На рис. 5 представлен этап построения бинарного дерева в уступчатом виде. Дерево строится от потомков к родителям, “снизу вверх”. Для левой части дерева, которая имеет большую сумму, присваивается 1, для правой – 0. Затем по дереву и исходному тексту формируется итоговая кодовая последовательность.

На рис. 6 происходит декодирование исходного сообщения при помощи кодовой последовательности и бинарного дерева, также выводится код каждого символа.

```
-------------
f4shannon3
1. Count and sort by ascending numbers of each characters in the entered text
f4shannon3[10]
n[3] h[1] o[1] f[1] a[1] 4[1] 3[1] s[1]
2. Divide the list into two parts, with the total frequency counts of the left part being as close to the total of the right as possible
------------------------
s34af[5] and ohn[5] | Full weight =  10
s34[3] and af[2] | Full weight =  5
s3[2] and 4[1] | Full weight =  3
s[1] and 3[1] | Full weight =  2
a[1] and f[1] | Full weight =  2
oh[2] and n[3] | Full weight =  5
o[1] and h[1] | Full weight =  2
------------------------
3. We`ve just got tree`s leafs
```

Рисунок 1 – Основной этап метода Фано-Шеннона

```
4. Now we should build the Shannon-Fano tree
 Remember: assign 1 to leafs with smaller weights (left)
    and assign 0 to leafs with larger weights (right)
------------------------
's34afohn' / {10}
    [1] 's34af' / {5}
       [1] 's34' / {3}
          [1] 's3' / {2}
             [1] 's' / {1}
             [0] '3' / {1}
          [0] '4' / {1}
       [0] 'af' / {2}
          [1] 'a' / {1}
          [0] 'f' / {1}
    [0] 'ohn' / {5}
       [1] 'oh' / {2}
          [1] 'o' / {1}
          [0] 'h' / {1}
       [0] 'n' / {3}
------------------------
5. As we can see it`s very simple to create a binary code to the each character in our text
6. Finally go to write it
10011011110101010000011001110
```

Рисунок 2 – Построение бинарного дерева и кодовой последовательности

```
___________ Decoding Shannon-Fano algorithm ___________
We received the prefix code that can be decoded using the binary tree very easy
[0] f = 100
[1] 4 = 110
[2] s = 1111
[3] h = 010
[4] a = 101
[5] n = 00
[6] n = 00
[7] o = 011
[8] n = 00
[9] 3 = 1110
Result decoding message = f4shannon3
```

Рисунок 3 – Декодирование метода Фано-Шеннона

```
--------------
h4ffma2n
1. Count and sort by ascending numbers of each characters in the entered text
h4ffma2n[8]
f[2] m[1] n[1] h[1] a[1] 4[1] 2[1]
2. Join 2 nodes with each other according to their low priority of weights
------------------------
2[1] and m[1] | Full weight =  2
h[1] and 4[1] | Full weight =  2
n[1] and a[1] | Full weight =  2
na[2] and f[2] | Full weight =  4
2m[2] and h4[2] | Full weight =  4
2mh4[4] and naf[4] | Full weight =  8
------------------------
3. We`ve just got tree`s leafs
```

Рисунок 4 – Основной этап метода Хаффмана

```
4. Now we should build the huffman tree
 Remember: assign 1 to leafs with smaller weights (left)
    and assign 0 to leafs with larger weights (right)
------------------------
'2mh4naf' / {8}
    [1] '2mh4' / {4}
       [1] '2m' / {2}
          [1] '2' / {1}
          [0] 'm' / {1}
       [0] 'h4' / {2}
          [1] 'h' / {1}
          [0] '4' / {1}
    [0] 'naf' / {4}
       [1] 'na' / {2}
          [1] 'n' / {1}
          [0] 'a' / {1}
       [0] 'f' / {2}
------------------------
5. As we can see it`s very simple to create a binary code to the each character in our text
6. Finally go to write it
101100000011001011011
```

Рисунок 5 – Построение бинарного дерева и кодовой последовательности

```
___________ Decoding Huffman algorithm ___________
We received the prefix code that can be decoded very easy using the binary tree
[0] h = 101
[1] 4 = 100
[2] f = 00
[3] f = 00
[4] m = 110
[5] a = 010
[6] 2 = 111
[7] n = 011
Result decoding message = h4ffma2n
```

Рисунок 6 – Декодирование метода Хаффмана

ЗАКЛЮЧЕНИЕ

В ходе работы была реализована консольная программа на языке C++ для статического префиксного кодирования и декодирования текста методами Фано-Шеннона и Хаффмана.

Для наглядного представления работы алгоритмов была создана демонстрация отдельных шагов каждого из методов с их пояснением.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Код Хаффмана // Википедия. URL: https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%B4_%D0%A5%D0%B0%D1%84%D1%84%D0%BC%D0%B0%D0%BD%D0%B0

2. Алгоритм Шеннона — Фано // Википедия. URL: https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%A8%D0%B5%D0%BD%D0%BD%D0%BE%D0%BD%D0%B0_%E2%80%94_%D0%A4%D0%B0%D0%BD%D0%BE

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

main.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstring>
#include <vector>
#include <algorithm>
#include "main.h"
#include "codetree.h"

using namespace std;

void Main::menu() {
    cout << "1. Shannon-Fano coding/decoding" << endl;
    cout << "2. Huffman coding/decoding" << endl;
    cout << "0. Exit" << endl;
}

string Main::fileRead() {
    string fileName;
    string message;
    cout << "What`s the file name?" << endl;
    cin >> fileName;
    cout << "--------------" << endl;
    cout << "Reading from " << fileName << endl;
    cout << "--------------" << endl;
    ifstream inFile;
    inFile.open(fileName);
    if (!inFile) {
        cout << "Cannot find this file" << endl;
        cout << endl;
        return "";
    }
    string result;
    while (!inFile.eof()) {
        getline(inFile, message);
        if (message.empty()) continue;
        result += message;
    }
    cout << result << endl;
    inFile.close();
    return result;
}

void Main::getHuffman() {
    string message;
    message = fileRead();
```

```
    if (message.empty()) {
        return;
    }
    CodeTree *huffman = huffmanCode(message);
    char *huffCode = encode(huffman, message);
    int decodingLength = message.size();
    vector<string> stringRows;
    char *decodeFano = decode(huffman, huffCode, stringRows);
    cout << "------------------------" << endl;
    cout << "3. We`ve just got tree`s leafs" << endl;
    cout << "4. Now we should build the huffman tree" << endl;
    cout << " Remember: assign 1 to leafs with smaller weights (left)" <<
endl;
    cout << "     and assign 0 to leafs with larger weights (right)" <<
endl;
    cout << "------------------------" << endl;
    displayTree(huffman, 1);
    cout << "------------------------" << endl;
    cout << "5. As we can see it`s very simple to create a binary code to
the each character in our text" << endl;
    cout << "6. Finally go to write it" << endl;
    cout << huffCode << endl;
    cout << "___________ Decoding Huffman algorithm ___________" << endl;
    cout << "We received the prefix code that can be decoded very easy
using the binary tree" << endl;
    for (int i = 0; i < decodingLength; i++) {
        cout << "[" << i << "] " << stringRows[i] << endl;
    }
    cout << "Result decoding message = " << decodeFano << endl;
    cout << endl;
}

void Main::getFanoShannon() {
    string message;
    message = fileRead();
    if (message.empty()) {
        return;
    }
    CodeTree *fsTree = fanoShanon(message);
    int decodingLength = message.size();
    vector<string> stringRows;
    char *fsCode = encode(fsTree, message);
    char *decodeFano = decode(fsTree, fsCode, stringRows);
    cout << "3. We`ve just got tree`s leafs" << endl;
    cout << "4. Now we should build the Shannon-Fano tree" << endl;
    cout << " Remember: assign 1 to leafs with smaller weights (left)" <<
endl;
    cout << "     and assign 0 to leafs with larger weights (right)" <<
endl;
    cout << "------------------------" << endl;
    displayTree(fsTree, 1);
```

```
    cout << "------------------------" << endl;
    cout << "5. As we can see it`s very simple to create a binary code to
the each character in our text" << endl;
    cout << "6. Finally go to write it" << endl;
    cout << fsCode << endl;
    cout << "___________ Decoding Shannon-Fano algorithm ___________" <<
endl;
    cout << "We received the prefix code that can be decoded using the
binary tree very easy" << endl;
    for (int i = 0; i < decodingLength; i++) {
        cout << "[" << i << "] " << stringRows[i] << endl;
    }
    cout << "Result decoding message = " << decodeFano << endl;
    cout << endl;
}


int main() {
    Main main;
    while (true) {
        main.menu();
        cin >> main.choice;
        switch (main.choice) {
            case 1:
                main.getFanoShannon();
                break;
            case 2:
                main.getHuffman();
                break;
            case 0:
                exit(1);
        }
    }
}
```

main.h

```
#ifndef COURSEWORK_MAIN_H
#define COURSEWORK_MAIN_H
#include "fs.h"
#include "huffman.h"

class Main {
public:
    Main() {}

    unsigned int choice;

    string fileRead();

    void getHuffman();
```

```
    void getFanoShannon();

    void menu();
};

#endif //COURSEWORK_MAIN_H
```

codetree.cpp

```
#include "codetree.h"
#include <climits>
#include <cstring>
#include <iostream>

bool greaterSymbol(const Symbol &l, const Symbol &r) {
    return l.weight > r.weight;
}

CodeTree *makeLeaf(const Symbol &s) {
    return new CodeTree{s, nullptr, nullptr, nullptr};
}

CodeTree *makeNode(int weight, CodeTree *left, CodeTree *right) {
    Symbol s{0, weight};
    return new CodeTree{s, nullptr, left, right};
}

bool isLeaf(const CodeTree *node) {
    return node->left == nullptr && node->right == nullptr;
}

bool isRoot(const CodeTree *node) {
    return node->parent == nullptr;
}

static void fillSymbolsMap(const CodeTree *node, const CodeTree
**symbols_map);

bool checkString(const string &message) {
    for (int i = 1; i < message.size(); i++) {
        if (message[0] != message[i]) return false;
    }
    return true;
}

char *encode(const CodeTree *tree, const string &message) {
    unsigned int firstLength = 1000;
    char *code = new char[firstLength];
    const CodeTree **symbols_map = new const CodeTree *[UCHAR_MAX];
    for (int i = 0; i < UCHAR_MAX; ++i) {
```

```
        symbols_map[i] = nullptr;
    }
    fillSymbolsMap(tree, symbols_map);
    int len = message.size();
    unsigned int index = 0;
    char path[UCHAR_MAX];
    if (!checkString(message)) {
        for (int i = 0; i < len; ++i) {
            const CodeTree *node = symbols_map[message[i] - CHAR_MIN];
            int j = 0;
            while (!isRoot(node)) {
                if (node->parent->left == node)
                    path[j++] = '0';
                else
                    path[j++] = '1';
                node = node->parent;
            }
            while (j > 0) {
                if (index >= firstLength) {
                    code = resize(code, firstLength);
                }
                code[index++] = path[--j];
            }
        }
    } else {
        while (index < message.size()) {
            if (index >= firstLength) {
                code = resize(code, firstLength);
            }
            code[index++] = '0';
        }
    }
    code[index] = 0;
    delete[] symbols_map;
    return code;
}

char *resize(char *prevArr, unsigned int &sizeOfCode) {
    unsigned int newSize = sizeOfCode * 2;
    char *newArr = new char[newSize];
    for (int i = 0; i < sizeOfCode; i++) {
        newArr[i] = prevArr[i];
    }
    sizeOfCode = newSize;
    return newArr;
}

char *decode(const CodeTree *tree, const char *code, vector<string>
&rows) {
    unsigned int firstLength = 1000;
    char *message = new char[firstLength];
```

```
    int index = 0;
    int len = strlen(code);
    const CodeTree *v = tree;
    string binaryCodes;
    string rowString;
    int rowCounter = 0;
    // check if we have only root
    if (tree->parent || tree->left || tree->right) {
        for (int i = 0; i < len; ++i) {
            if (code[i] == '0') {
                v = v->left;
                binaryCodes.push_back(code[i]);
            } else {
                v = v->right;
                binaryCodes.push_back(code[i]);
            }
            if (isLeaf(v)) {
                if (index >= firstLength) {
                    message = resize(message, firstLength);
                }
                message[index++] = v->s.c;
                rowString += (v->s.c);
                rowString += " = " + binaryCodes;
                rows.push_back(rowString);
                rowString.clear();
                binaryCodes.clear();
                v = tree;
            }
        }
    } else {
        for (int i = 0; i < len; ++i) {
            binaryCodes.push_back(code[i]);
            if (index >= firstLength) {
                message = resize(message, firstLength);
            }
            message[index++] = v->s.c;
            rowString += (v->s.c);
            rowString += " = " + binaryCodes;
            rows.push_back(rowString);
            rowString.clear();
            rowCounter++;
            binaryCodes.clear();
        }
    }
    message[index] = 0;
    return message;
}


void destroy(CodeTree *tree) {
    if (tree == nullptr) return;
```

```
    destroy(tree->left);
    destroy(tree->right);
    delete tree;
    tree = nullptr;
}

void fillSymbolsMap(const CodeTree *node, const CodeTree **symbols_map) {
    if (isLeaf(node))
        symbols_map[node->s.c - CHAR_MIN] = node;
    else {
        fillSymbolsMap(node->left, symbols_map);
        fillSymbolsMap(node->right, symbols_map);
    }
}
```

codetree.h

```
#ifndef COURSEWORK_CODETREE_H
#define COURSEWORK_CODETREE_H

#include <string>
#include <vector>

using namespace std;
struct Symbol {
    char c;
    int weight;
};

bool greaterSymbol(const Symbol &l, const Symbol &r);

struct CodeTree {
    Symbol s;
    CodeTree *parent;
    CodeTree *left;
    CodeTree *right;
};

CodeTree *makeLeaf(const Symbol &s);

CodeTree *makeNode(int weight, CodeTree *left, CodeTree *right);

bool isLeaf(const CodeTree *node);

bool isRoot(const CodeTree *node);

char *encode(const CodeTree *tree, const string &message);
char *decode(const CodeTree *tree, const char *code, vector<string>
&rows);

void destroy(CodeTree *tree);
```

```
char *resize(char *prevArr, unsigned int &sizeOfCode);


#endif //COURSEWORK_CODETREE_H
```

helper.cpp

```
#include "helper.h"

void printChWeights(CodeTree *ltree, CodeTree *rtree, int weight, string
&testRows) {
    string left;
    string right;
    if (ltree->s.c != 0 && rtree->s.c != 0) {
        testRows.push_back(rtree->s.c);
        testRows.push_back('[');
        testRows += to_string(rtree->s.weight);
        testRows.push_back(']');
        testRows += " and ";
        testRows.push_back(ltree->s.c);
        testRows.push_back('[');
        testRows += to_string(ltree->s.weight);
        testRows.push_back(']');
    } else if (ltree->s.c == 0 && rtree->s.c == 0) {
        helpToPrintChW(ltree, left);
        helpToPrintChW(rtree, right);
        testRows += right;
        testRows.push_back('[');
        testRows += to_string(rtree->s.weight);
        testRows.push_back(']');
        testRows += " and ";
        testRows += left;
        testRows.push_back('[');
        testRows += to_string(ltree->s.weight);
        testRows.push_back(']');
    } else if (ltree->s.c == 0) {
        helpToPrintChW(ltree, left);
        testRows.push_back(rtree->s.c);
        testRows.push_back('[');
        testRows += to_string(rtree->s.weight);
        testRows.push_back(']');
        testRows += " and ";
        testRows += left;
        testRows.push_back('[');
        testRows += to_string(ltree->s.weight);
        testRows.push_back(']');
    } else if (rtree->s.c == 0) {
        helpToPrintChW(rtree, right);
        testRows += right;
        testRows.push_back('[');
        testRows += to_string(rtree->s.weight);
```

```
        testRows.push_back(']');
        testRows += " and ";
        testRows.push_back(ltree->s.c);
        testRows.push_back('[');
        testRows += to_string(ltree->s.weight);
        testRows.push_back(']');
    }
    testRows += " | Full weight =  ";
    testRows += to_string(weight);
}

void helpToPrintChW(CodeTree *b, string &side) {
    if (b->s.c != 0) {
        side += b->s.c;
    }
    if (b->right) {
        helpToPrintChW(b->right, side);
    }
    if (b->left) {
        helpToPrintChW(b->left, side);
    }
}

void throughRecursive(CodeTree *b, string &side) {
    if (b != nullptr) {
        CodeTree *newT = b;
        if (b->s.c == 0) {
            helpToPrintChW(newT, side);
        }
        if (b->right) {
            throughRecursive(b->right, side);
        }
        if (b->left) {
            throughRecursive(b->left, side);
        }
        delete newT;
    }
}

void displayTree(const CodeTree *b, int n) {
    if (b != nullptr) {
        const CodeTree *newT = b;
        if (b->s.c != 0) {
            cout << "'" << b->s.c << "' / {";
            cout << b->s.weight << "}";
        } else {
            cout << "'";
            helpToDisplay(newT);
            cout << "' / {";
            cout << b->s.weight << "}";
        }
```

```
        cout << endl;
        if (b->right) {
            for (int i = 1; i <= n; i++) cout << "   ";
            cout << " [1] ";
            displayTree(b->right, n + 1);
        }
        if (b->left) {
            for (int i = 1; i <= n; i++) cout << "   ";
            cout << " [0] ";
            displayTree(b->left, n + 1);
        }
        delete newT;
    }
}

void helpToDisplay(const CodeTree *b) {
    if (b->s.c != 0) {
        cout << b->s.c;
    }
    if (b->right) {
        helpToDisplay(b->right);
    }
    if (b->left) {
        helpToDisplay(b->left);
    }
}
```

helper.h

```
#ifndef COURSEWORK_HELPER_H
#define COURSEWORK_HELPER_H

#include "codetree.h"
#include <iostream>
#include <string>

#pragma once

void printChWeights(CodeTree *ltree, CodeTree *rtree, int weight, string
&testRows);

void helpToPrintChW(CodeTree *b, string &side);

void throughRecursive(CodeTree *b, string &side);

void displayTree(const CodeTree *b, int n);

void helpToDisplay(const CodeTree *b);
```

```
#endif //COURSEWORK_HELPER_H
```

fs.cpp

```
#include "fs.h"
#include <algorithm>
#include <climits>
#include <string>
#include <cstring>
#include <iostream>
#include <vector>
using namespace std;
static int middle(const Symbol *symbols, int l, int sum, int &lsum, int
&rsum);

CodeTree *fanno_shannon(const Symbol *symbols, int l, int r, int sum,
vector<string> &vectorFanoShanon) {
    if (l >= r) return nullptr;
    if (r - l == 1) return makeLeaf(symbols[l]);
    int lsum, rsum;
    int m = middle(symbols, l, sum, lsum, rsum);
    CodeTree *ltree = fanno_shannon(symbols, l, m + 1, lsum,
vectorFanoShanon);
    CodeTree *rtree = fanno_shannon(symbols, m + 1, r, rsum,
vectorFanoShanon);
    CodeTree *node = makeNode(sum, ltree, rtree);
    string testRow;
    printChWeights(ltree, rtree, sum, testRow);
    vectorFanoShanon.push_back(testRow);
    ltree->parent = node;
    rtree->parent = node;
    return node;
}

CodeTree *fanoShanon(const Symbol *symbols, int len) {
    int sum = 0;
    for (int i = 0; i < len; ++i) {
        cout << symbols[i].c << "[" << symbols[i].weight << "] ";
        sum += symbols[i].weight;
    }
    cout << endl;
    vector<string> vectorFanoShanon;
    CodeTree *returnValue = fanno_shannon(symbols, 0, len, sum,
vectorFanoShanon);
    cout << "2. Divide the list into two parts, with the total frequency
counts of the left part being as close to the total of the right as
possible" << endl;
    cout << "------------------------" << endl;
    if (vectorFanoShanon.size() > 0) {
        for (int i = vectorFanoShanon.size() - 1; i >= 0; i--) {
            cout << vectorFanoShanon[i] << endl;
```

```
        }
    }
    cout << "------------------------" << endl;
    return returnValue;
}

CodeTree *fanoShanon(const string &message) {
    Symbol symbols[UCHAR_MAX];
    for (int i = 0; i < UCHAR_MAX; ++i) {
        symbols[i].c = i + CHAR_MIN;
        symbols[i].weight = 0;
    }
    int size = message.size();
    for (int i = 0; i < size; ++i) {
        symbols[message[i] - CHAR_MIN].weight++;
    }
    std::sort(symbols, symbols + UCHAR_MAX, greaterSymbol);
    int len = 0;
    while (symbols[len].weight > 0 && len < UCHAR_MAX) len++;
    cout << "1. Count and sort by ascending numbers of each characters in
the entered text" << endl;
    cout << message << "[" << message.size() << "]" << endl;
    return fanoShanon(symbols, len);
}

int middle(const Symbol *symbols, int l, int sum, int &lsum, int &rsum) {
    int m = l;
    lsum = symbols[m].weight;
    rsum = sum - lsum;
    int delta = lsum - rsum;
    while (delta + symbols[m + 1].weight < 0) {
        m++;
        lsum += symbols[m].weight;
        rsum -= symbols[m].weight;
        delta = lsum - rsum;
    }
    return m;
}
```

fs.h

```
#ifndef COURSEWORK_FS_H
#define COURSEWORK_FS_H

#include "codetree.h"
#include "helper.h"

#include <string>

CodeTree *fanoShanon(const std::string &message);
```

```
CodeTree *fanoShanon(const Symbol *symbols, int len);



#endif //COURSEWORK_FS_H
```

huffman.cpp

```
#include "huffman.h"
#include "priority_queue.h"
#include <functional>
#include <algorithm>
#include <climits>
#include <cstring>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

CodeTree *huffmanCode(const Symbol *symbols, int len) {
    PriorityQueue<CodeTree *> *queue = createPQ<CodeTree *>(len);
    for (int i = 0; i < len; ++i) {
        cout << symbols[i].c << "[" << symbols[i].weight << "] ";
        push(queue, symbols[i].weight, makeLeaf(symbols[i]));
    }
    cout << endl;
    cout << "2. Join 2 nodes with each other according to their low
priority of weights" << endl;
    cout << "------------------------" << endl;
    vector<string> rows;
    while (size(queue) > 1) {
        CodeTree *ltree = pop(queue);
        CodeTree *rtree = pop(queue);
        int weight = ltree->s.weight + rtree->s.weight;
        CodeTree *node = makeNode(weight, ltree, rtree);
        ltree->parent = node;
        rtree->parent = node;
        push(queue, weight, node);
        string testRow;
        printChWeights(ltree, rtree, weight, testRow);
        rows.push_back(testRow);
    }
    for (int i = 0; i < rows.size(); i++) {
        cout << rows[i] << endl;
    }
    CodeTree *result = pop(queue);
    destroyPq(queue);
    return result;
}

CodeTree *huffmanCode(const string &message) {
```

```
    Symbol symbols[UCHAR_MAX];
    for (int i = 0; i < UCHAR_MAX; ++i) {
        symbols[i].c = i + CHAR_MIN;
        symbols[i].weight = 0;
    }
    int size = message.size();
    for (int i = 0; i < size; ++i)
        symbols[message[i] - CHAR_MIN].weight++;
    std::sort(symbols, symbols + UCHAR_MAX, greaterSymbol);
    int len = 0;
    while (symbols[len].weight > 0 && len < UCHAR_MAX) len++;
    cout << "1. Count and sort by ascending numbers of each characters in
the entered text" << endl;
    cout << message << "[" << message.size() << "]" << endl;
    return huffmanCode(symbols, len);
}
```

huffman.h

```
#ifndef COURSEWORK_HUFFMAN_H
#define COURSEWORK_HUFFMAN_H
#include "codetree.h"
#include "helper.h"
#include <iostream>

using namespace std;

CodeTree *huffmanCode(const string &message);

CodeTree *huffmanCode(const Symbol *symbols, int len);

#endif //COURSEWORK_HUFFMAN_H
```

priority_queue.h

```
#ifndef COURSEWORK_PRIORITY_QUEUE_H
#define COURSEWORK_PRIORITY_QUEUE_H

#include <utility>

template<typename T>
struct PriorityQueueItem {
    int key;
    T data;
};
template<typename T>
struct PriorityQueue {
    int size_;
    int capacity_;
    PriorityQueueItem<T> *heap_;
};
```

```
template<typename T>
PriorityQueue<T> *createPQ(int capacity) {
    PriorityQueue<T> *pq = new PriorityQueue<T>;
    pq->heap_ = new PriorityQueueItem<T>[capacity];
    pq->capacity_ = capacity;
    pq->size_ = 0;
    return pq;
}

template<typename T>
int size(PriorityQueue<T> *pq) {
    return pq->size_;
}

template<typename T>
void siftUp(PriorityQueue<T> *pq, int index) {
    int parent = (index - 1) / 2;
    while (parent >= 0 && pq->heap_[index].key < pq->heap_[parent].key) {
        std::swap(pq->heap_[index], pq->heap_[parent]);
        index = parent;
        parent = (index - 1) / 2;
    }
}

template<typename T>
bool push(PriorityQueue<T> *pq, int key, const T &data) {
    if (pq->size_ >= pq->capacity_) return false;
    pq->heap_[pq->size_].key = key;
    pq->heap_[pq->size_].data = data;
    pq->size_++;
    siftUp(pq, pq->size_ - 1);
    return true;
}

template<typename T>
void siftDown(PriorityQueue<T> *pq, int index) {
    int l = 2 * index + 1;
    int r = 2 * index + 2;
    int min = index;
    if (l < pq->size_ && pq->heap_[l].key < pq->heap_[min].key)
        min = l;
    if (r < pq->size_ && pq->heap_[r].key < pq->heap_[min].key)
        min = r;
    if (min != index) {
        std::swap(pq->heap_[index], pq->heap_[min]);
        siftDown(pq, min);
    }
}

template<typename T>
```

```
T pop(PriorityQueue<T> *pq) {
    std::swap(pq->heap_[0], pq->heap_[pq->size_ - 1]);
    pq->size_--;
    siftDown(pq, 0);
    return pq->heap_[pq->size_].data;
}

template<typename T>
void destroyPq(PriorityQueue<T> *pq) {
    delete[] pq->heap_;
    delete pq;
}

#endif //COURSEWORK_PRIORITY_QUEUE_H
```

Makefile

```
CXX=g++
RM=rm -f
LDFLAGS=-g -Wall

SRCS=main.cpp codetree.cpp fs.cpp helper.cpp huffman.cpp priority_queue.h
OBJS=$(subst .cpp,.o,$(SRCS))

all: main

main: $(OBJS)
	$(CXX) $(LDFLAGS) -o main $(OBJS)

main.o: main.cpp main.h

codetree.o: codetree.cpp codetree.h

helper.o: helper.cpp helper.h

fs.o: fs.cpp fs.h

huffman.o: huffman.cpp huffman.h

priority_queue.h.o: priority_queue.h.h

clean:
	$(RM) $(OBJS)

distclean: clean
	$(RM) main
```