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

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: Динамическое кодирование и декодирование по Хаффману – сравнительное исследование со “статическим” методом.

Студент гр. 9381 Фоминенко А.Н.

Преподаватель Фирсов М.А.

Санкт-Петербург

2020

ЗАДАНИЕ

НА КУРСОВУЮ РАБОТУ

Студент Фоминенко А.Н.

Группа 9381

Тема работы : Динамическое кодирование и декодирование по Хаффману – сравнительное исследование со “статическим” методом.

Исходные данные:

Содержание пояснительной записки:

- титульный лист, лист задания, аннотация, содержание;
- формальная постановка задачи;
- описание алгоритма;
- описание структур данных и функций;
- тестирование;
- исследование;
- программный код (в приложении) с комментариями;
- выводы.

Предполагаемый объем пояснительной записки:

Не менее 15 страниц.

Дата выдачи задания: 31.10.2020

Дата сдачи реферата: 28.12.2020

Дата защиты реферата: 28.12.2020

Студент		Фоминенко А.Н.
Преподаватель		Фирсов М.А.

АННОТАЦИЯ

В данной курсовой работе производится исследование статического и динамического(адаптивного) кодирования Хаффмана. Результатом исследования являются данные на основе, которых выявляются слабые и сильные стороны двух алгоритмов.

SUMMARY

This paper is supposed to bring brief investigation on Huffman static and Adaptive algorithms. The result of investigation are data that show good and poor sides of each algorithm.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Алгоритм Хаффмана — алгоритм оптимального префиксного кодирования алфавита с минимальной избыточностью. Был разработан в 1952 году аспирантом Массачусетского технологического института Дэвидом Хаффманом при написании им курсовой работы. В настоящее время используется во многих программах сжатия данных.

Этот метод кодирования состоит из двух основных этапов:

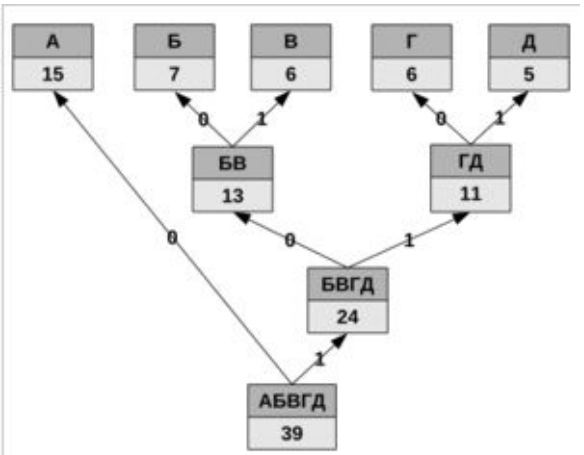
1. Построение оптимального кодового дерева.
2. Построение отображения код-символ на основе построенного дерева.

2. ХОД ВЫПОЛНЕНИЯ РАБОТЫ

2.1 Статический метод

Классический алгоритм Хаффмана на входе получает таблицу частот встречаемости символов в сообщении. Далее на основании этой таблицы строится дерево кодирования Хаффмана

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.



Итого:

А	Б	В	Г	Д
0	100	101	110	111

2.2 Динамический метод

Адаптивное сжатие позволяет не передавать модель сообщения вместе с ним самим и ограничиться одним проходом по сообщению как при кодировании, так и при декодировании.

Наиболее известными алгоритмами перестроения являются алгоритм Фоллера-Галлагера-Кнута (FGK) и алгоритм Виттера.

Все алгоритмы перестроения дерева при считывании очередного символа, включают в себя две операции:

Первая — увеличение веса узлов дерева. Вначале увеличиваем вес листа, соответствующего считанному символу, на единицу. Затем увеличиваем вес родителя, чтобы привести его в соответствие с новыми значениями веса потомков. Этот процесс продолжается до тех пор, пока мы не доберемся до корня дерева. Среднее число операций увеличения веса равно среднему количеству битов, необходимых для того, чтобы закодировать символ.

Вторая операция — перестановка узлов дерева — требуется тогда, когда увеличение веса узла приводит к нарушению свойства упорядоченности, то есть тогда, когда увеличенный вес узла стал больше, чем вес следующего по порядку узла. Если и дальше продолжать обрабатывать увеличение веса, двигаясь к корню дерева, то дерево перестанет быть деревом Хаффмана. Чтобы сохранить упорядоченность дерева кодирования, алгоритм работает следующим образом. Пусть новый увеличенный вес узла равен W+1. Тогда начинаем двигаться по списку в сторону увеличения веса, пока не найдем последний узел с весом W. Переставим текущий и найденный узлы между собой в списке, восстанавливая таким образом порядок в дереве (при этом родители каждого из узлов тоже изменятся). На этом операция перестановки заканчивается.

После перестановки операция увеличения веса узлов продолжается дальше. Следующий узел, вес которого будет увеличен алгоритмом, — это новый родитель узла, увеличение веса которого вызвало перестановку.

2.3. Описание функций

```
/* структура узла дерева */
struct Node {
    int isZero;
    int isRoot;
    int isLeaf;

    struct node *parent;
    struct node *leftChild;
    struct node *rightChild;

    unsigned char symbol;
    int value;
    int order;
};

-----------------------------------------------------------------

/* функция создание дерева
 * returns: указатель на корень дерева
 */
Node* createTree()
-----------------------------------------------------------------
/* функция возвращания вершины соответств. данному символу(если такой вершины нет то возвращаем nullptr)
symbol - символ
symbols  - массив символов
 */
Node* getTreeFromSymbol(unsigned char symbol, Symbol **symbols)
-----------------------------------------------------------------
* фуннкция разворота массива чисел
 * code: массив
 * codeSize: длина массива
 */
void reverseCode(int *code, int codeSize)

/* функция возврата пути (массива из 0 и 1) соотв. данной вершине в дереве
 * node: вершина
 * n: макс количество символов в массиве
 */
```

```
int* codeOfNode(Node *node, int *n)
-----------------------------------------------------------------
/* функция создания ребенка от данной вершины
 * parent:вершина
 * isRoot: 1 если это корень, иначе 0
 * symbol: символ новой вершины(ребенка)
 * value: значение в вершине
 * returns: ребенка.
 */
Node* addChild(Node *parent, int isZero, int isRoot, unsigned char symbol, int value, int order)
-----------------------------------------------------------------
/* добавляет новый символ в дерево и в массив символов
 * symbol: символ
 * symbols: массив символов
 * returns: указатель на родителя вершины которую мы создали
 */
Node* addSymbol(unsigned char symbol, Node **zeroNode, Symbol **symbols)
-----------------------------------------------------------------
/* поиск вершины с таким же значением
 * currMax: наша вершина.
 * root: корень.
 * returns: вершину с таким же значением как у данной, иначе nullptr.
 */
Node* findReplaceNode(Node *cur, Node *root)
-----------------------------------------------------------------
/* функция
 поменять местами вершины в дереве
n1 - первая вершина
n2 - вторая
 */
void swapNodes(Node *n1, Node *n2)
```

```
-----------------------------------------------------------------
/* функция обновления дерева
 * currNode: начальная вершина с которой происходит обновление
 * root: корень дерева.
 */
void updateTree(Node *currNode, Node *root) {


-----------------------------------------------------------------
/* функция кодировки
 * fp_in: вход файл
 * fp_out: выход файл
 */
void encode(FILE *fp_in, FILE *fp_out)
-----------------------------------------------------------------
/*функция декодирования
 * fp_in: вход файл
 * fp_out: выход файл
 */
void decode(FILE *fp_in, FILE *fp_out)
```

3. Тестирование

Основной тест №1:

Входные данные: закодировать строку

this is course work

input	output
this is course work	0000001001101101010001000011000010011000000111010000111011100000
zxcqwe	0000001000100
mama mila ramy	0101001000000001001100001011000
help me !!!	0000001000100100010000011010
huffman encode	0000010000000010011001000101001000110001000111

4. Исследование

Так как алгоритм Хаффмана сжимает данные за счёт вероятностей появления символов в источнике, следовательно для того чтоб успешно что-то сжать и разжать нам потребуется знать эти самые вероятности для каждого символа. Статические алгоритмы пробегаются по самому файлу и подсчитывает какой символ сколько раз встречается. Затем, в соответствии с вероятностями появлений символов, строится двоичное дерево Хаффмана - откуда извлекаются соответствующие каждому символу коды разной длины. И на третьем этапе снова осуществляется проход по исходному файлу, когда каждый символ заменяется на свой древесный код. Таким образом статическому алгоритму требуется два прохода по файлу источнику, чтоб закодировать данные.

Динамический алгоритм позволяет реализовать однопроходную модель сжатия. Не зная реальных вероятностей появлений символов в исходном файле - программа постепенно изменяет двоичное дерево с каждым встречаемым символом увеличивая частоту его появления в дереве и перестраивая в связи с этим само дерево. Однако становится очевидным, что выиграв в количестве проходов по исходному файлу - мы начинаем терять в качестве сжатия, так как в статическом алгоритме частоты встречаемости символов были известны с самого начала и длины кодов этих символов были более близки к оптимальным, в то время как динамическая модель изучая источник постепенно доходит до его реальных частотных характеристик и узнаёт их лишь полностью пройдя исходный файл. Так как динамическое двоичное дерево постоянно модифицируется новыми символами - нам нет необходимости запоминать их частоты заранее - при разархивировании, программа, получив из архива код символа, точно так же восстановит дерево, как она это делала при сжатии и увеличит на 1 частоту его встречаемости. Более того, нам не требуется запоминать какие символы в двоичном дереве не встречаются. Все символы которые будут добавляться в дерево и есть те, которые нам потребуются для восстановления первоначальных данных. В статическом алгоритме с этим делом немного сложнее - в самом начале сжатого файла требуется хранить информацию о встречаемых в файле источнике символах и их вероятностных частотах. Это обусловлено тем, что ещё до начала разархивирования нам необходимо знать какие символы будут встречаться и каков будет их код.

Исходя из выше написанного можно понять что адаптивный алгоритм будет сжимать хуже чем статический, давайте это проверим:

a = количество символов после сжатия в адаптивном делить на количество символов после сжатия в статическом

строка	адаптивный	статический	a
this is course work	00000010011011010101 00010000110000100110 00000111010000111011 10000 количество символов :: 66	10101100011111011111 00001010010011111101 10000100011011 количество символов :: 54	1.22
mnogo bykav tyt napisano	10010101110101010101 01101011010101111010 11010111010101011010 10111110111111011110 10101101011011010101 11101101110101 количество символов :: 114	00111011000110100011 11110010011000001111 11101111101110001001 010001110101100 количество символов ::76	1.5
mama mila ramy	01010010000000010011 000010110000 количество символов ::32	10111011100110101100 11110000 количество символов ::28	1.14
help me !!!	00000010001001000100 0001101010 количество символов ::30	10100100010011001111 11 количество символов ::22	1.36
huffman encode	00000100000000100110 01000101001000110001 0000111 количество символов ::47	11111100100100111001 00111010110011101000 101 количество символов ::42	1.2
qwertyuiop[asdf ghjklwertyuiops dfvgbhjkl;dcvfb nm,./ertyuiop	00000010001000000110 01000011000100000100 00000111001100010100 10000011100011000010 10011100010101001010	11001010000011000010 00011110111110100001 10010110111110110100 10011101011011110110 10101101001001110000	1.18

	01010110010000110001 10010000110001100010 00001000011110001111 00101101001011111011 01001001010000001101 11000010001001010000 11000111110000010001 11010001111100001101 01000001010111101111 01101 количество символов ::285	01100001000011110111 11010000110010100100 11101011000110111110 00101101010110100100 11111000011111010110 00101011100011001111 01001110111111010111 количество символов ::240	

среднее значение a ::

$$\alpha = \frac{\sum \alpha_i}{n} = \frac{1.22 + 1.5 + 1.14 + 1.36 + 1.2 + 1.18}{6} = 1.26$$

Это означает, что статический алгоритм в среднем сжимает в 1.26 раз лучше чем адаптивный.

5. Заключение

В ходе данной работы были изучены алгоритмы статического и динамического кодирования Хаффмана. И было проведено исследование на основании которого было выявлено, что статический метод в 1.26 раз лучше сжимает данный чем динамический.

ПРИЛОЖЕНИЕ А

Исходный код программы

Файл main.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define INVALID -1
#define HOW_MANY_POSSIBLE_SYMBOLS 256 /* количество допустимых
символов */

enum { ENCODE, DECODE };

/* структура узла дерева */
struct node {
    int isZero;
    int isRoot;
    int isLeaf;

    struct node *parent;
    struct node *leftChild;
    struct node *rightChild;

    unsigned char symbol;
    int value;
    int order;
};
typedef struct node Node;

/* символ и укзатель на его вершину в дереве */
struct symbol {
    char symbol;
    Node *tree;
};
```

```
typedef struct symbol Symbol;

/* создание дерева
 * returns: указатель на корень дерева
 */
Node* createTree() {
    Node *tree = malloc(sizeof(Node));
    tree->isZero = 1;
    tree->isRoot = 1;
    tree->isLeaf = 1;

    tree->parent = NULL;
    tree->leftChild = NULL;
    tree->rightChild = NULL;

    tree->symbol = INVALID;
    tree->value = 0;
    tree->order = HOW_MANY_POSSIBLE_SYMBOLS * 2;

    return tree;
}

/* возвращаем вершину соответств. данному символу(если такой
вершины нет то возвращаем nullptr)
 */
Node* getTreeFromSymbol(unsigned char symbol, Symbol **symbols) {
    Symbol *symbolPtr = symbols[(unsigned int)symbol];

    if (!symbolPtr) {
        return NULL;
    }

    return symbolPtr->tree;
}
```

```
/* разворот массива чисел
 * code: массив
 * codeSize: длина массива
 */
void reverseCode(int *code, int codeSize) {
    if (code == NULL) {
        printf("reverseCode: tried to reverse a NULL code.\n");
        return;
    }

    int *start = code;
    int *end = code+(codeSize-1);

    while (start < end) {
        int temp = *start;
        *start = *end;
        *end = temp;
        start++;
        end--;
    }
}

/* возвращает массив 0 и 1 соотв. данной вершине в дереве
 * node: вершина
 * n: макс количество символов в массиве
 */
int* codeOfNode(Node *node, int *n) {
    Node *current = node;
    int *code = malloc(sizeof(int) * HOW_MANY_POSSIBLE_SYMBOLS *
2);

    int i = 0;
    while (!current->isRoot) {
        Node *parent = current->parent;
        code[i] = (parent->leftChild == current) ? 0 : 1;
```

```
        current = current->parent;
        i++;
    }
    reverseCode(code, i);

    *n = i;
    return code;
}

/* создает ребенка от данной вершины
 * parent:вершина
 * isRoot: 1 если это корень, иначе 0
 * symbol: символ ноой вершины(ребенка)
 * value: значение в венршине
 * returns: ребенка.
 */
Node* addChild(Node *parent, int isZero, int isRoot, unsigned char
symbol, int value, int order) {
    Node *node = malloc(sizeof(Node));
    node->isZero = isZero;
    node->isRoot = isRoot;
    node->isLeaf = 1;
    node->parent = parent;
    node->leftChild = NULL;
    node->rightChild = NULL;
    node->symbol = symbol;
    node->value = value;
    node->order = order;
    return node;
}

/* добавляет новый сивол в дерево и в массив символов
 * symbol: символ
 * symbols: массив символов
 * returns: указатель на родителя вершины которую мы создали
```

```
 */
Node* addSymbol(unsigned char symbol, Node **zeroNode, Symbol
**symbols) {
    Node *leftNode = addChild(*zeroNode, 1, 0, INVALID, 0,
(*zeroNode)->order - 2);
    Node *rightNode = addChild(*zeroNode, 0, 0, symbol, 1,
(*zeroNode)->order - 1);

    Node *previousZeroNode = *zeroNode;
    (*zeroNode)->isZero = 0;
    (*zeroNode)->isLeaf = 0;
    (*zeroNode)->leftChild = leftNode;
    (*zeroNode)->rightChild = rightNode;

    unsigned int symbolIndex = (unsigned int)symbol;
    symbols[symbolIndex] = malloc(sizeof(Symbol));
    symbols[symbolIndex]->symbol = symbol;
    symbols[symbolIndex]->tree = rightNode;

    *zeroNode = leftNode;

    return previousZeroNode;
}

/* поиск вершины с таким же значением
 * currMax: наша вершина.
 * root: корень.
 * returns: вершину с таким же значением как у данной, иначе
nullptr.
 */
Node* findReplaceNode(Node *cur, Node *root) {
    Node *result = cur;

    if (root->value > result->value && !root->isLeaf) {
```

```
        Node *greatestLeft = findReplaceNode(result,
root->leftChild);
        if (greatestLeft) result = greatestLeft;

        Node *greatestRight = findReplaceNode(result,
root->rightChild);
        if (greatestRight) result = greatestRight;
    } else if (root->value == result->value && root->order >
result->order) {
        result = root;
    }

    return (result != cur) ? result : NULL;
}

/* поменять местами вершины в дереве
 */
void swapNodes(Node *n1, Node *n2) {
    int tempOrder = n1->order;
    n1->order = n2->order;
    n2->order = tempOrder;

    if (n1->parent->leftChild == n1) {
        n1->parent->leftChild = n2;
    } else {
        n1->parent->rightChild = n2;
    }

    if (n2->parent->leftChild == n2) {
        n2->parent->leftChild = n1;
    } else {
        n2->parent->rightChild = n1;
    }

    Node *temp = n1->parent;
```

```
    n1->parent = n2->parent;
    n2->parent = temp;
}

/* обновление дерева
 * currNode: начальная вершина с которой происходит обновление
 * root: корень дерева.
 */
void updateTree(Node *currNode, Node *root) {
    while (!currNode->isRoot) {
        Node *replaceNode = findReplaceNode(currNode, root);

        if (replaceNode && currNode->parent != replaceNode) {
            swapNodes(currNode, replaceNode);
        }

        (currNode->value)++;
        currNode = currNode->parent;
    }

    (currNode->value)++;
}

unsigned char addCodeToBuffer(int *code, int codeSize, FILE *fp,
unsigned char buffer, int *bufferSize) {
    unsigned char currBuffer = buffer;

    int i;
    for (i = 0; i < codeSize; i++) {
        unsigned char bit = ((unsigned char) code[i]) <<
(*bufferSize - 1);
        currBuffer = currBuffer | bit;

        (*bufferSize)--;
```

```
        if (*bufferSize == 0) {
            fwrite(&currBuffer, sizeof(unsigned char), 1, fp);
            currBuffer = 0;
            *bufferSize = 8;
        }
    }

    return currBuffer;
}

unsigned char addByteToBuffer(char byte, FILE *fp, unsigned char
buffer, int *bufferSize) {
    unsigned char currBuffer = buffer;

    int howManyBitsWillBeWritten = *bufferSize;
    int shiftSize = 8 - howManyBitsWillBeWritten;
    unsigned char tempByte = ((unsigned char) byte) >> shiftSize;
    currBuffer = currBuffer | tempByte;

    fwrite(&currBuffer, sizeof(unsigned char), 1, fp);

    currBuffer = byte << howManyBitsWillBeWritten;

    return (*bufferSize == 8) ? 0 : currBuffer;
}

/* запись буфера в конечный файл
 * в последний бит так же записывается количество бит, для
декодера
 *
 * fp: файл вывода.
 * buffer: буфер.
 * bufferSize: размер буфера.
 */
```

```
void writeRemainingBits(FILE *fp, unsigned char buffer, int
bufferSize) {
    if (bufferSize < 8) {
        fwrite(&buffer, sizeof(unsigned char), 1, fp);
    }

    /* записывает сколько битов нужно прочитать(для декодера) */
    buffer = (bufferSize == 8) ? 8 : 8 - bufferSize;
    fwrite(&buffer, sizeof(unsigned char), 1, fp);
}

/* кодировка
 * fp_in: вход файл
 * fp_out: выход файл
 */
void encode(FILE *fp_in, FILE *fp_out) {
    Node *root = createTree();
    Node *zeroNode = root;

    unsigned char buffer = 0;
    int bufferSize = 8;

    Symbol **symbols = calloc(HOW_MANY_POSSIBLE_SYMBOLS,
sizeof(Symbol*)); /* initializing with 0s */

    unsigned char currByte;
    while (fread(&currByte, sizeof(unsigned char), 1, fp_in) > 0)
{
        Node *symbolTree = getTreeFromSymbol(currByte, symbols);

        if (symbolTree) {
            int codeSize;
            int *symbolCode = codeOfNode(symbolTree, &codeSize);
            buffer = addCodeToBuffer(symbolCode, codeSize, fp_out,
buffer, &bufferSize);
```

```
            updateTree(symbolTree, root);
            free(symbolCode);
        } else {
            int codeSize;

            int *zeroCode = codeOfNode(zeroNode, &codeSize);
            buffer = addCodeToBuffer(zeroCode, codeSize, fp_out,
buffer, &bufferSize);
            buffer = addByteToBuffer(currByte, fp_out, buffer,
&bufferSize);

            Node *newNode = addSymbol(currByte, &zeroNode,
symbols);
            updateTree(newNode, root);
            free(zeroCode);
        }
    }

    writeRemainingBits(fp_out, buffer, bufferSize);
}

/* читает след бит из буфера и возвращает его
 * fp: вход файл
 * buffer: буфер
 * bufferSize: сколько осталось бит в буфере
 * fileSize: размер файла
 * returns: 0 или 1.
 */
int readBit(FILE *fp, unsigned char *buffer, int *bufferSize, long
int fileSize, int readHowManyBitsAtTheEnd) {
    if (*bufferSize == 0) {
        *bufferSize = 8;
        fread(buffer, sizeof(unsigned char), 1, fp);
    }
```

```
    if (readHowManyBitsAtTheEnd != 8) {
        if (ftell(fp) == fileSize && readHowManyBitsAtTheEnd <= (8
- *bufferSize)) return -1;
    }

    if (ftell(fp) > fileSize || feof(fp)) return -1;

    (*bufferSize)--;

    return (*buffer >> *bufferSize) & 1;
}

char readByte(FILE *fp, unsigned char *buffer, int *bufferSize,
long int fileSize, int readHowManyBitsAtTheEnd) {
    char result = 0;

    int i, bit;
    for (i = 0; i < 8; i++) {
        bit = readBit(fp, buffer, bufferSize, fileSize,
readHowManyBitsAtTheEnd);
        bit = bit << (7-i);
        result |= bit;
    }

    return result;
}

/* декодирование
 * fp_in: вход файл
 * fp_out: выход файл
 */
void decode(FILE *fp_in, FILE *fp_out) {
    Node *root = createTree();
    Node *zeroNode = root;
```

```
    unsigned char buffer = 0;
    int bufferSize = 0;

    Symbol **symbols = calloc(HOW_MANY_POSSIBLE_SYMBOLS,
sizeof(Symbol*));

    fseek(fp_in, -1, SEEK_END);
    long int fileSize = ftell(fp_in);

    unsigned char readHowManyBitsAtTheEnd;
    fread(&readHowManyBitsAtTheEnd, sizeof(unsigned char), 1,
fp_in);
    rewind(fp_in);

    while (!feof(fp_in)) {
        Node *currNode = root;

        int endOfFile = 0;
        while (!currNode->isLeaf && !endOfFile) {
            int bit = readBit(fp_in, &buffer, &bufferSize,
fileSize, readHowManyBitsAtTheEnd);
            if (bit == 0) {
                currNode = currNode->leftChild;
            } else if (bit == 1) {
                currNode = currNode->rightChild;
            } else {
                endOfFile = 1;
            }
        }

        if (endOfFile) break;

        unsigned char c;
        if (currNode->isZero) {
```

```
            c = readByte(fp_in, &buffer, &bufferSize, fileSize,
readHowManyBitsAtTheEnd);
            currNode = addSymbol(c, &zeroNode, symbols);
        } else {
            c = currNode->symbol;
        }

        fwrite(&c, sizeof(unsigned char), 1, fp_out);
        updateTree(currNode, root);
    }
}

int main(int argc, char *argv[]) {
     if (argc != 4) {
        printf("Usage:\n");
          printf("\t./fgk input_file output_file -c (to
encode)\n");
          printf("\t./fgk input_file output_file -d (to
decode)\n");
          exit(1);
     }

    FILE *fp_in;
    FILE *fp_out;
    int option;

    fp_in = fopen(argv[1], "rb");
    while (fp_in == NULL) {
        char in_file[100];
        printf("The file %s could not be opened. Try again: ",
argv[1]);
        scanf("%s", in_file);
        fp_in = fopen(in_file, "rb");
    }
```

```
    fp_out = fopen(argv[2], "wb");

    if (strcmp(argv[3], "-e") == 0 || strcmp(argv[3], "-E") == 0)
{
        option = ENCODE;
    } else if (strcmp(argv[3], "-d") == 0 || strcmp(argv[3], "-D")
== 0) {
        option = DECODE;
    } else {
        char opt;
        do {
            printf("type 'e' to encode or 'd' to decode: ");
            scanf("%c", &opt);
            getchar();
        } while (opt != 'e' && opt != 'E' && opt != 'd' && opt !=
'D');
        option = (opt == 'e' || opt == 'E') ? ENCODE : DECODE;
    }

    if (option == ENCODE) {
        encode(fp_in, fp_out);
        printf("The file was encoded.\n");
    } else {
        decode(fp_in, fp_out);
        printf("The file was decoded.\n");
    }

    fclose(fp_in);
    fclose(fp_out);

    return 0;
}
```