**Лабораторная работа №3**

**Сжатие данных**

Написать программу, сжимающую данные из некоторого файла по алгоритму Хаффмана.

Программа должна выполнять следующие действия:

Сжатие данных

1. Открывать файл, подлежащий сжатию, как бинарный файл байтов.
2. Подсчитывать частоту вхождения каждого байта в файле.
3. По имеющимся частотам строить дерево Хаффмана.
4. Создавать новый файл-архив.
5. Записывать в файл-архив заголовок – байты с частотами их вхождения.
6. Последовательно считывать байты исходного файла, кодировать их и записывать в файл-архив.

Распаковка данных

1. Открывать файл-архив.
2. Считывать из файла-архива заголовок и строить по нему дерево Хаффмана.
3. Последовательно считывать байты файла-архива, анализировать байты побитно с помощью дерева Хаффмана и записывать найденные в дереве байты в новый файл.

**Текст программы**

#include <iostream>

#include <fstream>

#include <string>

#include "windows.h"

using namespace std;

// Структура для узла дерева Хаффмана

class HuffmanNode {

public:

unsigned char data; //символ

size\_t frequency; //частота

HuffmanNode\* left; //указатель на левое поддерево

HuffmanNode\* right; //указатель на правое поддерево

// Конструктор для создания узла дерева

HuffmanNode(unsigned char data, size\_t frequency) {

this->data = data;

this->frequency = frequency;

this->left = NULL;

this->right = NULL;

}

// Перегрузка оператора < для использования при сортировке узлов

bool operator<(const HuffmanNode& other) const {

return frequency > other.frequency;

}

~HuffmanNode() {

delete left;

delete right;

}

};

// Функция для построения дерева Хаффмана

HuffmanNode\* buildHuffmanTree(const size\_t\* frequency, const int size) {

// Создание массива указателей на узлы

HuffmanNode\*\* trees = new HuffmanNode \* [size];

// Заполнение массива узлами

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

trees[i] = new HuffmanNode(static\_cast<unsigned char>(i), frequency[i]);

}

// Построение дерева Хаффмана

while (true) {

int minIdx1 = -1, minIdx2 = -1;

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

if (trees[i] != NULL) {

if (minIdx1 == -1 || trees[i]->frequency < trees[minIdx1]->frequency) {

minIdx2 = minIdx1;

minIdx1 = i;

}

else if (minIdx2 == -1 || trees[i]->frequency < trees[minIdx2]->frequency) {

minIdx2 = i;

}

}

}

if (minIdx2 == -1) {

break;

}

HuffmanNode\* tmp = new HuffmanNode(0, trees[minIdx1]->frequency + trees[minIdx2]->frequency);

tmp->left = trees[minIdx1];

tmp->right = trees[minIdx2];

trees[minIdx1] = tmp;

trees[minIdx2] = NULL;

}

// Находим корень дерева

HuffmanNode\* root = NULL;

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

if (trees[i] != NULL) {

root = trees[i];

break;

}

}

delete[] trees;

return root;

}

// Функция для построения таблицы кодов Хаффмана

void buildHuffmanCodes(HuffmanNode\* root, const string& code, string codes[]) {

if (!root) return;

if (!root->left && !root->right) {

codes[root->data] = code;

}

buildHuffmanCodes(root->left, code + "0", codes);

buildHuffmanCodes(root->right, code + "1", codes);

}

// Функция для сжатия файла

void compressFile(const string& inputFile, const string& outputFile) {

// Открытие входного файла

ifstream inFile(inputFile, ios::binary);

if (!inFile.is\_open()) {

cerr << "Ошибка открытия входного файла" << endl;

return;

}

// Подсчет частоты вхождения каждого байта в файле

size\_t frequency[256] = { 0 };

unsigned char byte;

while (inFile.read(reinterpret\_cast<char\*>(&byte), sizeof(byte))) {

frequency[byte]++;

}

// Построение дерева Хаффмана

HuffmanNode\* root = buildHuffmanTree(frequency, 256);

// Построение таблицы кодов Хаффмана

string codes[256];

buildHuffmanCodes(root, "", codes);

// Закрытие входного файла

inFile.close();

// Открытие выходного файла для записи архива

ofstream outFile(outputFile, ios::binary);

if (!outFile.is\_open()) {

cerr << "Ошибка открытия выходного файла" << endl;

delete root;

return;

}

cout << "Идет архивация файла. Пожалуйста, подождите..." << endl;

// Запись заголовка в выходной файл

for (int i = 0; i < 256; ++i) {

outFile.write(reinterpret\_cast<char\*>(&frequency[i]), sizeof(size\_t));

}

// Переоткрытие входного файла для сжатия

inFile.open(inputFile, ios::binary);

// Сжатие данных и запись их в выходной файл

string buffer;

while (inFile.read(reinterpret\_cast<char\*>(&byte), sizeof(byte))) {

buffer += codes[byte];

while (buffer.size() >= 8) {

unsigned char compressedByte = 0;

for (int i = 0; i < 8; ++i) {

if (buffer[i] == '1') {

compressedByte += static\_cast<unsigned char>(pow(2, 7 - i));

}

}

outFile.write(reinterpret\_cast<const char\*>(&compressedByte), sizeof(compressedByte));

buffer = buffer.substr(8);

}

}

// Обработка оставшихся битов

if (!buffer.empty()) {

unsigned char compressedByte = 0;

for (size\_t i = 0; i < buffer.size(); ++i) {

if (buffer[i] == '1') {

compressedByte |= (1 << (7 - i));

}

}

outFile.write(reinterpret\_cast<const char\*>(&compressedByte), sizeof(compressedByte));

}

// Закрытие выходного файла

outFile.close();

// Освобождение памяти, выделенной для дерева Хаффмана

delete root;

cout << "Архив сохранен в " << outputFile << endl;

}

// Функция для распаковки файла

void decompressFile(const string& inputFile, const string& outputFile) {

// Открытие входного файла архива

ifstream inFile(inputFile, ios::binary);

if (!inFile.is\_open()) {

cerr << "Ошибка открытия входного файла" << endl;

return;

}

cout << "Идет разархивация файла. Пожалуйста, подождите..." << endl;

// Чтение частот из входного файла

size\_t frequency[256];

for (int i = 0; i < 256; ++i) {

inFile.read(reinterpret\_cast<char\*>(&frequency[i]), sizeof(size\_t));

}

// Построение дерева Хаффмана

HuffmanNode\* root = buildHuffmanTree(frequency, 256);

// Открытие выходного файла для распаковки

ofstream outFile(outputFile, ios::binary);

if (!outFile.is\_open()) {

cerr << "Ошибка открытия выходного файла" << endl;

delete root;

return;

}

// Чтение данных и их распаковка с использованием дерева Хаффмана

HuffmanNode\* currentNode = root;

unsigned char byte;

while (inFile.read(reinterpret\_cast<char\*>(&byte), sizeof(byte))) {

for (int i = 0; i < 8; ++i) {

if (byte & (1 << (7 - i))) {

currentNode = currentNode->right;

}

else {

currentNode = currentNode->left;

}

if (!currentNode->left && !currentNode->right) {

outFile.write(reinterpret\_cast<char\*>(&currentNode->data), sizeof(currentNode->data));

currentNode = root;

}

}

}

// Закрытие файлов

inFile.close();

outFile.close();

// Освобождение памяти, выделенной для дерева Хаффмана

delete root;

cout << "Файл сохранен в " << outputFile << endl;

}

int main() {

// Пример использования: сжатие и распаковка файла

setlocale(LC\_ALL, "Russian");

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

int n;

string input, output;

cout << "Какую операцию вы хотите выполнить?" << endl;

cout << "1. Архивация" << endl;

cout << "2. Разархивация" << endl;

cout << "Введите номер операции: ";

cin >> n;

switch (n) {

case 1:

cout << "Введите путь и имя исходного файла: ";

cin >> input;

cout << "Введите путь для сохранения архива и его имя: ";

cin >> output;

compressFile(input, output);

break;

case 2:

cout << "Введите путь и имя исходного файла: ";

cin >> input;

cout << "Введите путь для сохранения архива и его имя: ";

cin >> output;

decompressFile(input, output);

break;

default:

cout << "Такой операции нет!" << endl;

}

return 0;

}

**Теория**

**Дерево Хаффмана** – это бинарное дерево, используемое в алгоритме Хаффмана для сжатия данных путем присвоения кода переменной длины символам. В этом дереве каждый лист представляет собой символ, а каждый внутренний узел объединяет два узла-потомка. При построении дерева Хаффмана символы с наименьшей частотой встречаемости находятся ближе к корню, а символы с более высокой частотой - дальше от корня.

Каждый путь от корня до листа представляет собой уникальный двоичный код, где путь к левому потомку обозначается как 0, а к правому – как 1. Это обеспечивает эффективность сжатия, так как более часто встречающиеся символы имеют более короткие коды.

**Алгоритм Хаффмана** состоит из следующих шагов:

1. Подсчитываются частоты встречаемости каждого символа в сообщении.
2. На основе этих частот строится дерево Хаффмана. Для этого символы с наименьшей частотой объединяются в узлы дерева, которые затем объединяются в более крупные узлы, пока не будет построено полное дерево.
3. На пути от корня дерева до листа каждому символу присваиваются двоичные коды. При этом более часто встречающиеся символы имеют более короткие коды
4. Закодированные данные записываются с использованием полученных кодов

**Описание программы**

Класс **HuffmanNode** – узел дерева Хаффмана.

Содержит следующие поля:

* **data** – символ типа **char**, представляющий данный узел. Если быть точнее, данная переменная хранит число – код символа в кодировке ASCII
* **frequency** – переменная, которая хранит частоту встречаемости символа в исходных данных
* **left** и **right** – указатели на левое и правое поддеревья

Также содержит следующие функции:

* **HuffmanNode()** – конструктор для создания объекта класса **HuffmanNode** – узла дерева. В качестве параметров принимает символ **data** и частоту его встречаемости **frequency**. Они записываются в соответствующие поля, а указателям **left** и **right** присваивается начальное значение **NULL**
* **operator<** – перегрузка оператора **<** для использования при сортировке узлов. Узлы будут сравниваться по частоте **frequency**

**buildHuffmanTree()** – функция для построения дерева Хаффмана. В качестве параметров принимает массив частот **frequency** и его размер **size**

1. Создается массив указателей на узлы дерева **trees**
2. Массив **trees** заполняется указателями на узлы дерева, которые создаются по массиву **frequency**
3. Функция начинает строить дерево Хаффмана.
   1. Функция ищет в массиве **trees** 2 узла с наименьшими частотами и записывает номера символов в переменные **minIdx1** и **minIdx2**
   2. Создается узел **tmp**, который будет являться родительским для узлов **trees[minIdx1]** и **trees[minIdx2]**. Его частота будет равна сумме частот 2 этих узлов. Потомкам узла **tmp** – **left** и **right** – присваиваются указатели на узлы **trees[minIdx1]** и **trees[minIdx2]**.
   3. Узлу **trees[midIdx1]** присваивается значение узла **tmp**, а узлу **trees[midIdx2]** – значение **NULL**. В массиве **trees** стало на 1 узел меньше
   4. Цикл повторяется снова, и снова ищутся 2 минимальных значения
   5. И так до тех пор, пока в массиве **trees** не останется всего 1 узел – это будет корень дерева.
4. В оставшемся массиве **trees** из 1 элемента этот самый элемент находится и присваивается переменной **root**. Эта переменная и будет выходным значением функции

**buildHuffmanCodes()** – функция для построения таблицы кодов Хаффмана. В качестве параметров принимает указатель на текущий узел дерева **root**, строку с текущим кодом **code** и массив строк с кодами **codes**

1. Если узел **root** равен **NULL**, сразу же выходим из функции
2. Если узел **root** не имеет левого и правого потомков **left** и **right**, в элемент под номером **data** массива **codes** записывается текущий код **code**
3. Функция рекурсивно вызывается для левого поддерева **left**, при этом к текущему коду **code** добавляется **«0»**.
4. Функция рекурсивно вызывается для правого поддерева **right**, при этом к текущему коду **code** добавляется **«1»**.

**compressFile()** – функция для сжатия файла. В качестве параметров принимает имя входного файла для сжатия **inputFile** и имя выходного файла **outputFile**, в который будет записан сжатый архив

1. Файл **inputFile** открывается в бинарном режиме. Если при открытии файла возникла ошибка – выводится сообщение *«Ошибка открытия входного файла»*, выполнение функции прекращается.
2. Создается массив **frequency** размером 256 элементов (именно столько символов содержит таблица ASCII), который будет хранить частоты символов. Изначально все элементы равны **0**
3. Объявляется переменная **byte** типа **char**, которая будет хранить байты, считываемые из входного файла
4. Входной файл читается побайтово. Для каждого считанного байта увеличивается элемент **[byte]** массива **frequency**
5. Создается узел дерева **root**, вызывается функция **buildHuffmanTree()**, которая строит дерево Хаффмана по массиву **frequency**
6. Создается массив строк **codes** размером 256 элементов, который будет хранить коды Хаффмана для каждого символа
7. Вызывается функция **buildHuffmanCodes()**, которая строит таблицу кодов Хаффмана, начиная с узла **root**
8. Файл **inputFile** закрывается
9. Открывается выходной файл **outputFile** в бинарном режиме. Если при открытии файла возникла ошибка – выводится сообщение *«Ошибка открытия выходного файла»*, выполнение функции прекращается.
10. В файл **outputFile** записывается заголовок – для каждого символа записывается частота его вхождения в исходный файл
11. Файл **inputFile** переоткрывается для сжатия
12. Создается строка **buffer**, которая будет накапливать последовательности битов, представляющих сжатые данные
13. Программа побайтово читает файл, записывая каждый байт в переменную **byte**
    1. Для каждого байта добавляется соответствующий ему код Хаффмана из массива **codes** в строку **buffer**
    2. Как только в строке **buffer** накопилось хотя бы 8 бит (1 байт), его можно записывать в выходной файл
    3. Создается переменная **compressedByte** типа char, которая будет хранить сжатый байт
    4. Для первых 8 битов из строки **buffer** добавляем в **compressedByte** соответствующие степени 2-ки, если бит равен «1». Например, если 1 бит равен «1», добавляем 27 = 128, если 2 бит равен «1» - добавляем 26 = 64 и т. д.
    5. Значение переменной **compressedByte** записывается в файл **outputFile**
    6. Уже записанные первые 8 бит из строки **buffer** обрезаются, остается только не записанный в файл остаток (если он есть)
    7. Запись битов продолжается до тех пор, пока не будет считан весь файл **inputFile**
14. Выходной файл **outputFile** закрывается
15. Удаляется узел дерева **root**

**decompressFile()** – функция для распаковки файла. В качестве параметров принимает имя входного файла для распаковки **inputFile** и имя выходного файла **outputFile**, в который будет записан распакованный файл

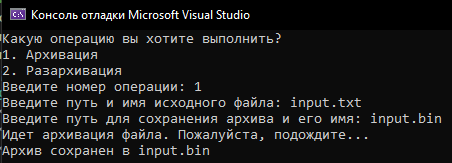
1. Файл **inputFile** открывается в бинарном режиме. Если при открытии файла возникла ошибка – выводится сообщение *«Ошибка открытия входного файла»*, выполнение функции прекращается.
2. Из файла **inputFile** считываются частоты появления каждого символа, которые записаны в заголовке (256 значений частот, по одному для каждого байта). Считанные частоты записываются в массив **frequency**
3. По массиву частот **frequency** с помощью функции **buildHuffmanTree()** строится дерево Хаффмана. Корень дерева сохраняется в переменную **root**
4. Открывается выходной файл **outputFile** в бинарном режиме. Если при открытии файла возникла ошибка – выводится сообщение *«Ошибка открытия выходного файла»*, выполнение функции прекращается.
5. Создается указатель **currentNode**, который указывает на текущий узел. В качестве начального значения присваивается корень дерева **root**
6. Создается переменная **byte** для прочитанных байтов
7. Программа побайтово читает файл, записывая каждый байт в переменную **byte**
   1. Каждый прочитанный байт обрабатывается побитово: если бит равен 1 – переходим в правый узел, если 0 – в левый
   2. Если у узла нет ни левого, ни правого потомков, значит достигнут конец дерева. Символ из данного узла записывается в выходной файл
   3. **currentNode** возвращается на корень **root**
   4. Далее считывается следующий байт, до тех пор, пока не будет достигнут конец файла
8. Файлы **inputFile** и **outputFile** закрываются
9. Удаляется узел дерева **root**

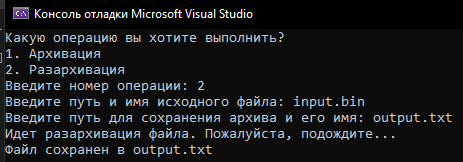
**main** – основная часть программы

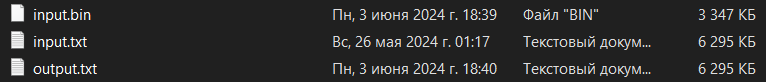
1. Программа предлагает ввести пользователю номер операции (1 – архивация, 2 - разархивация)
2. Если введен номер 1, программа запрашивает у пользователя имя исходного и сжатого файлов, затем вызывается функция для сжатия файла **compressFile**.
3. Если введен номер 2, программа запрашивает у пользователя имя исходного и распакованного файлов, затем вызывается функция для сжатия файла **decompressFile**.
4. Если введен какой-то другой номер, на экран выводится сообщение *«Такой операции нет!»*

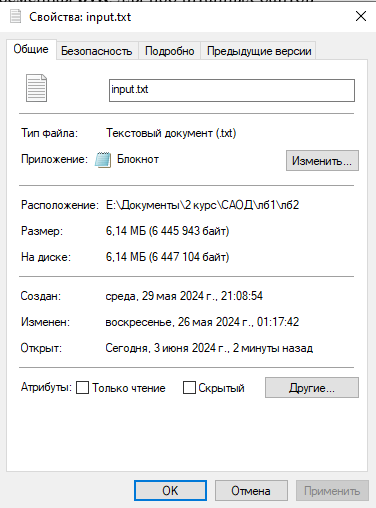
**Результаты тестов**

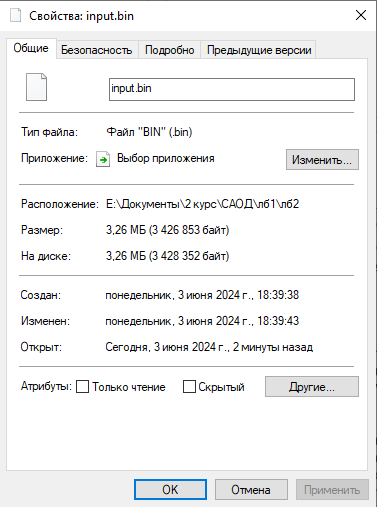
1. Текстовый файл формата .txt с произведением Л. Н. Толстого «Война и мир». Исходный вес файла составляет 6 445 943 байт (6295 Кб). После сжатия вес файла уменьшился до 3 426 853 байт (3347 Кб). После распаковки вес файла снова стал прежним – 6 445 943 байт (6295 Кб). Содержимое файла при сжатии и распаковке не пострадало.

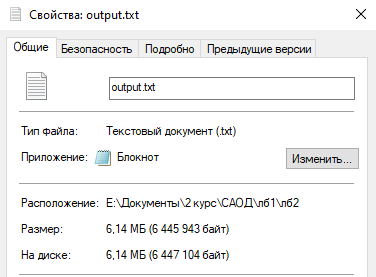


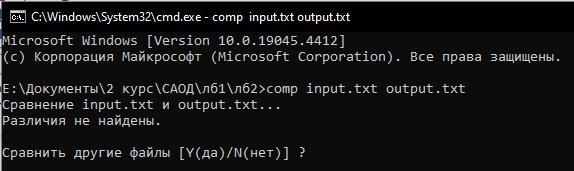




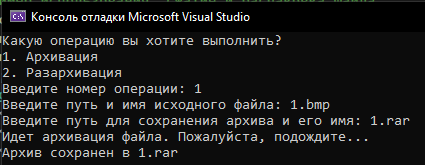


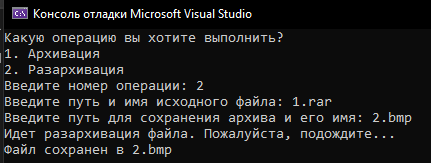


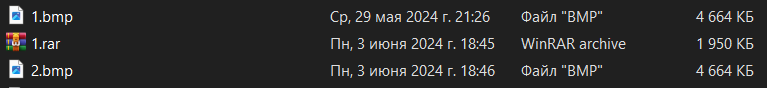


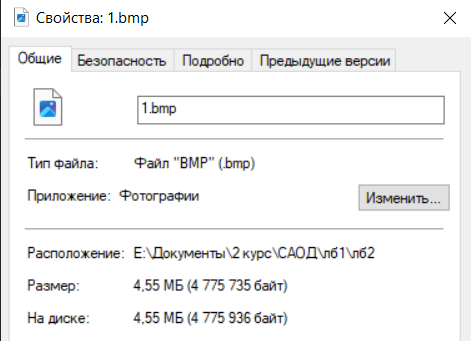


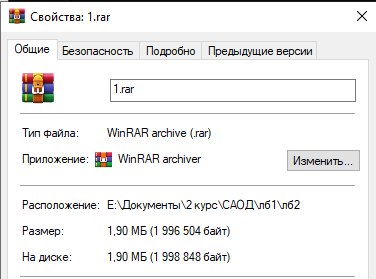
1. Картинка формата .bmp. Исходный вес файла составляет 4 755 735 байт (4664 Кб). После сжатия вес файла уменьшился до 1 996 504 байт (1950 Кб). После распаковки вес файла снова стал прежним – 4 755 735 байт (4664 Кб). Содержимое файла при сжатии и распаковке не пострадало.

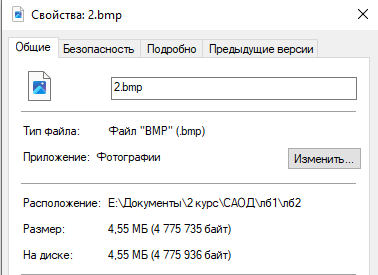


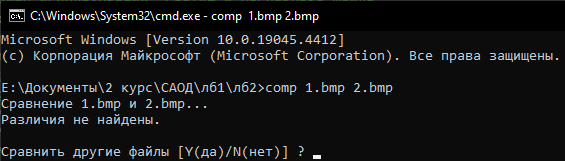




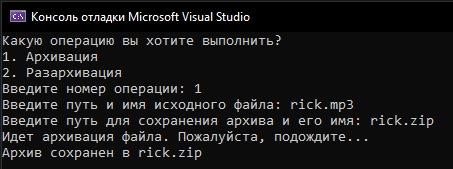


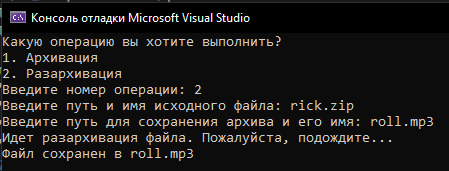


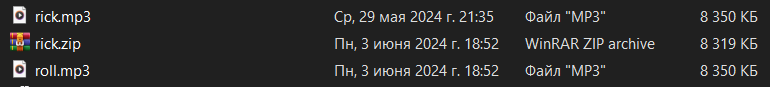


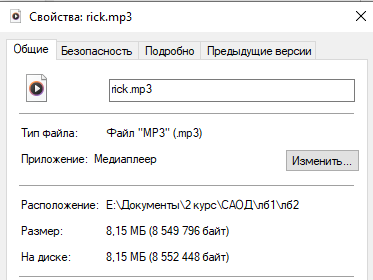


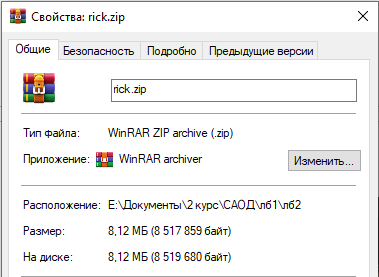
1. Музыкальный файл формата .mp3. Исходный вес файла составляет 8 549 796 байт (8350 Кб). После сжатия вес файла уменьшился до 8 517 859 байт (8319 Кб). После распаковки вес файла снова стал прежним – 8 549 796 байт (8350 Кб). Содержимое файла при сжатии и распаковке не пострадало.

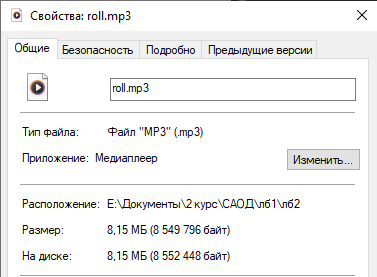


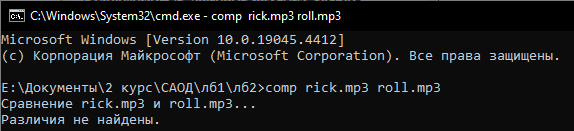




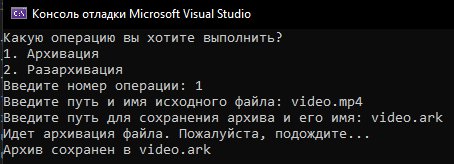


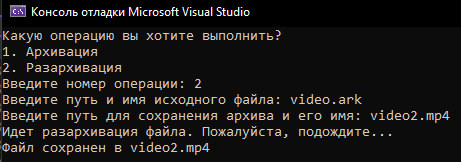


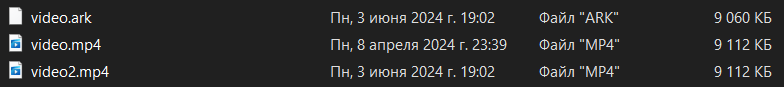


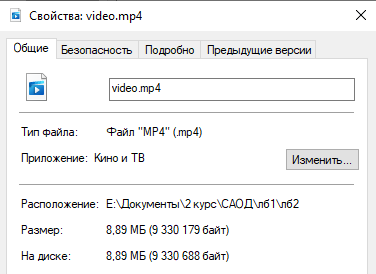


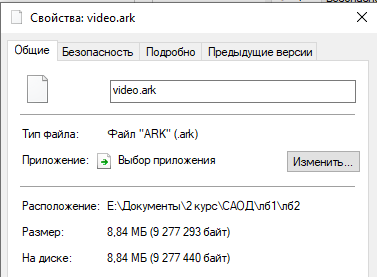
1. Видеофайл формата .mp4. Исходный вес файла составляет 9 330 179 байт (9112 Кб). После сжатия вес файла уменьшился до 9 277 293 байт (9060 Кб). После распаковки вес файла снова стал прежним – 9 330 179 байт (9112 Кб). Содержимое файла при сжатии и распаковке не пострадало.

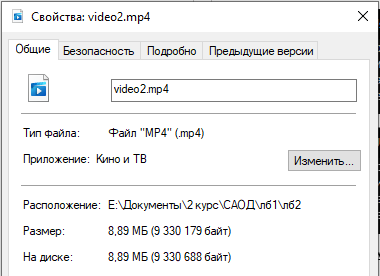


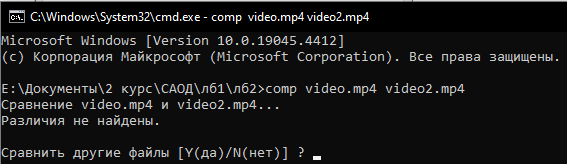




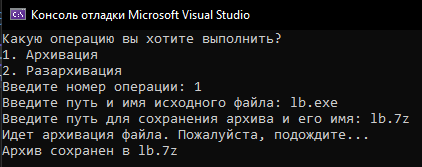


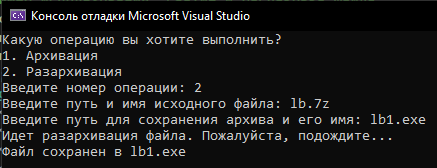


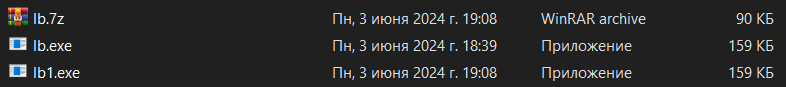


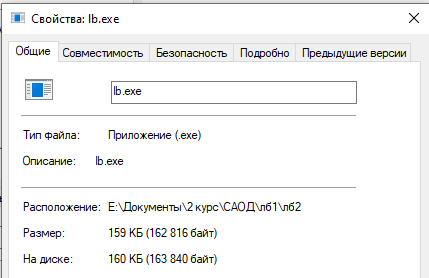


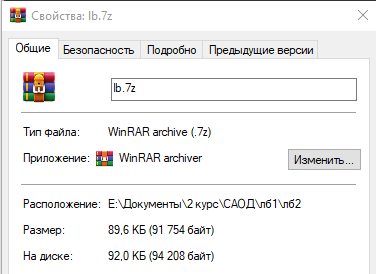
1. Программа формата .exe – сам архиватор, выполненный в рамках лабораторной работы. Исходный вес файла составляет 162 816 байт (159 Кб). После сжатия вес файла уменьшился до 91 754 байт (90 Кб). После распаковки вес файла снова стал прежним – 162 816 байт (159 Кб). Содержимое файла при сжатии и распаковке не пострадало.

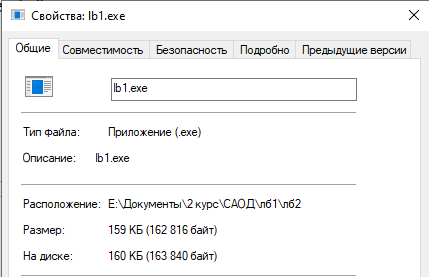


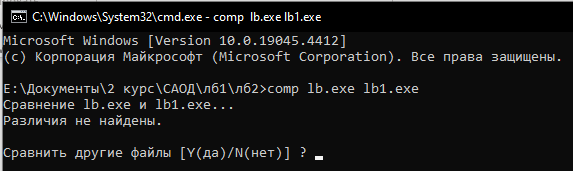




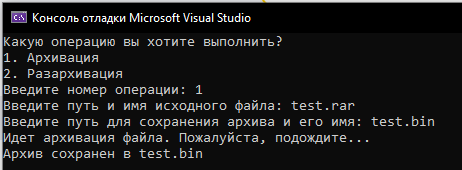


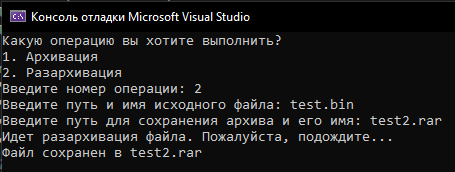


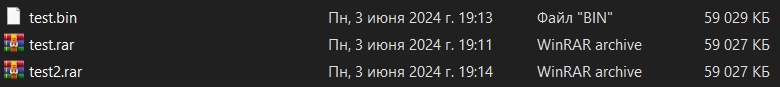


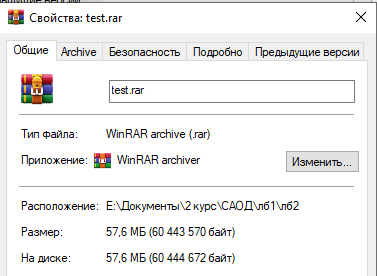


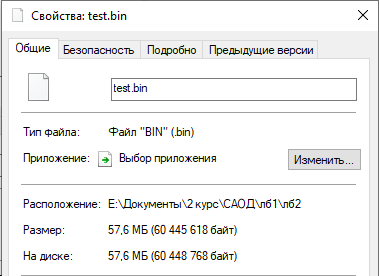
1. Архив формата .rar, содержащий все файлы, использованные в предыдущих тестах. Исходный вес файла составляет 60 443 570 байт (59027 Кб). После сжатия вес файла не уменьшился, а даже немного увеличился до 60 445 618 байт (59031 Кб). После распаковки вес файла снова стал прежним – 60 443 570 байт (59027 Кб). Содержимое файла при сжатии и распаковке не пострадало.

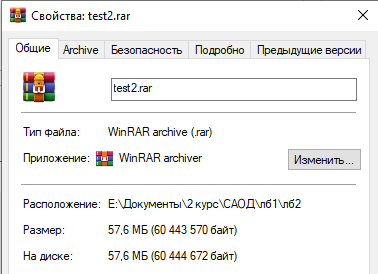


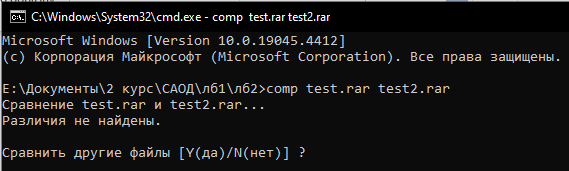




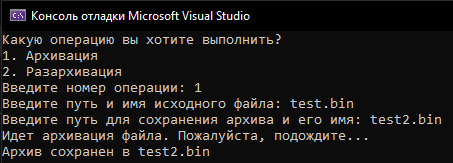


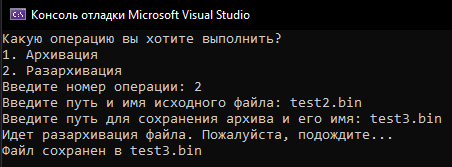


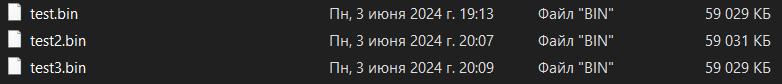


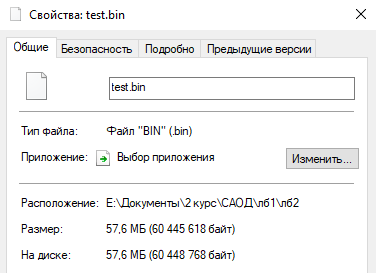


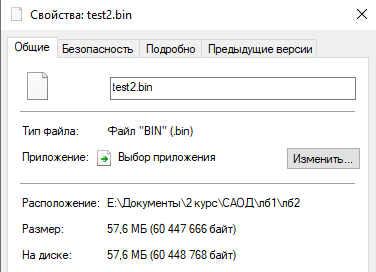
1. Бинарный файл формата .bin – один из файлов, полученных в ходе тестирования данного архиватора. Исходный вес файла составляет 60 445 618 байт (59029 Кб). После сжатия вес файла не уменьшился, а даже немного увеличился до 60 447 666 байт (59031 Кб). После распаковки вес файла снова стал прежним – 60 445 618 байт (59029 Кб). Содержимое файла при сжатии и распаковке не пострадало.

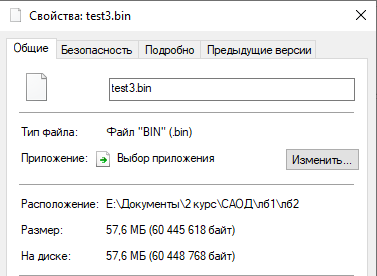


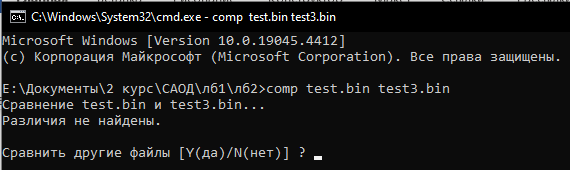












**Вывод**

В ходе выполнения лабораторной работы был написан архиватор, сжимающий данные по алгоритму Хаффмана, и распаковывающий ранее сжатые по такому методу файлы обратно.

В ходе тестирования архиватора были сжаты и распакованы следующие виды файлов:

* Текстовый файл .txt
* Картинка .bmp
* Музыкальный файл .mp3
* Видео .mp4
* Программа .exe
* Архив .rar
* Бинарный файл .bin

Во всех случаях, кроме архива и бинарного файла, размер файла уменьшился. После распаковки размер файла стал прежним, содержимое файлов не пострадало.

В случае с архивом и бинарным файлом размеры файлов увеличились на несколько килобайт. Это связано с тем, что алгоритм Хаффмана работает на основе поиска повторяющихся последовательностей в данных. Если данные уже сжаты, они содержат мало повторяющихся последовательностей, и попытка их повторного сжатия будет неэффективной, так как в файл будет только добавляться служебная информация (заголовок), тем самым лишь увеличивая размер файла, хоть и незначительно. Т. е., сжать архив еще сильнее путем многократного архивирования не получится. После распаковки размер файла стал прежним, содержимое файлов не пострадало.