

СОДЕРЖАНИЕ

| | |
|---|-----------|
| ВВЕДЕНИЕ | 2 |
| 1. Обзор литературы..... | 4 |
| 2. Структурное проектирование | 6 |
| 3. Функциональное проектирование | 8 |
| 4. Разработка программных модулей | 14 |
| 5. Тестирование программы и руководство пользователя | 24 |
| ЗАКЛЮЧЕНИЕ | 35 |
| Список использованных источников..... | 35 |
| ПРИЛОЖЕНИЕ А..... | 36 |
| ПРИЛОЖЕНИЕ Б | 37 |

ВВЕДЕНИЕ

В современное время архивы играют важную роль в хранении данных. Они позволяют хранить наборы разнообразных файлов. Это достаточно удобно, например, для передачи данных – легче передавать один файл, чем несколько. Архивы также позволяют хранить информацию в удобной форме.

Запакованные (сжатые) файлы занимают меньше дискового пространства и быстрее передаются на другие компьютеры, чем обычные файлы. Работа со сжатыми файлами и папками в Windows аналогична работе с обычными файлами и папками. Объединив несколько файлов в одну сжатую папку, вы сможете с легкостью поделиться ими.

Что представляет собой файл – это последовательность байт, которая оперирует процесс, открывший файл (ресурс).

На сегодняшний день существует три группы архиваторов:

1. Файловые позволяют упаковывать один или несколько файлов в единый архив. Размер архива, как правило, меньше чем суммарный размер исходных файлов. Воспользоваться архивными данными и программами пока они находятся в архиве нельзя. Для распаковки архива требуется разархиватор, который совмещен с архиватором в единой программе.
2. Программные позволяют упаковывать за один прием один единственный файл - выполняемую программу типа *.exe, которая при запуске самораспаковывается в оперативной памяти и начинает работу. Программа становится в два раза меньше и при этом сохраняет работоспособность.
3. Дисковые представляет собой резидентный драйвер, который незаметно для пользователя архивирует любую записываемую на диск информацию и распаковывает ее обратно при чтении. При этом на диске создается огромный архив, который отображается как еще один логический раздел винчестера.

С момента возникновения первой программы (архиватор) данного типа, было написано очень много различных архиваторов, поддерживающих различные форматы архивов. Самым распространенным был ARJ, на втором месте почти сразу за ним ZIP, затем, следовали такие архиваторы, как ARC, ACE, LZH. Теперь ситуация изменилась. Первое место среди форматов архиваторов занимает ZIP, забрав его у ARJ, который отошел теперь на задний план, на втором месте RAR и потом следуют ACE, ARJ и другие менее популярные форматы.

ZIP – формат был разработан компанией PKWARE.

RAR – формат был разработан Евгением Рошалем, автором одноименного архиватора и благодаря удобному интерфейсу одновременно с хорошим сжатием завоевал популярность.

Теперь немного о сжатии. Сжатие файлов зависит от используемого алгоритма и от содержания в файлах данных. Текстовые файлы, как правило, сжимаются очень хорошо, бинарные файлы – хуже, а файлы, содержимое которых уплотнено максимально (аудио-, видео-, а также программы – инсталляторы) – сжимаются хуже всех.

1. Обзор литературы

Начнём с главного определения – это **Архиватор**.

Архиватор — программа, предназначенная для упаковки без потерь файлов в единый файл-архив. Распаковка архивов выполняется с помощью того же архиватора. Большинство современных архиваторов также выполняют сжатие упаковываемых в архив данных.

Таким образом мы подошли к следующему термину – сжатие (сжатие данных).

Сжатие данных (*data compression*) – это алгоритм преобразования данных, который производится с целью уменьшения занимаемого ими объёма. Обратная процедура называется восстановлением данных (распаковкой или декомпрессией).

Основное применение сжатия – это устранение избыточности, которая содержится в исходных данных. Примером избыточности является повторение фрагментов в тексте. Такого рода избыточность устраняется заменой повторяющейся последовательности ссылкой на уже закодированный фрагмент с указанием его длины.

Существующие алгоритмы сжатия данных можно разделить на два больших класса – без потерь, и с потерями данных:

- Алгоритмы сжатия без потерь применяются для уменьшения размера данных, и работают таким образом, что возможно восстановить данные в точности такими, какие они были до сжатия. Они применяются в коммуникациях, архиваторах и некоторых алгоритмах сжатия аудио и графической информации.
- Алгоритмы с потерями обычно применяются для сжатия изображений и аудио. Эти алгоритмы позволяют достичь больших степеней сжатия благодаря избирательной потере качества. Однако, по определению, восстановить первоначальные данные из сжатого результата невозможно.

Основной принцип алгоритмов сжатия базируется на том, что в любом файле, содержащем неслучайные данные, информация которых полностью (приведённый выше пример) или частично повторяется.

Для частичных повторений, используя статистические математические модели можно определить вероятность повторения определённой комбинации символов. После этого можно создать коды, обозначающие выбранные фразы,

и назначить самым часто повторяющимся фразам самые короткие коды. Для этого используются разные техники, например, *энтропийное кодирование*, кодирование повторов, и сжатие при помощи словаря. С их помощью 8-битный символ, или целая строка, могут быть заменены всего лишь несколькими битами, устраняя таким образом излишнюю информацию.

В данной работе будет применяться два алгоритма сжатия на выбор пользователя:

1. **Алгоритм RLE (run-length encoding) кодирование длин серий** – алгоритм сжатия данных, заменяющий повторяющиеся символы (серии) на один символ и число его повторов. *Серией* называется последовательность, которая состоит из нескольких одинаковых символов. При кодировании (сжатии) строка одинаковых символов, составляющих серию, заменяется строкой, содержащей символ и количество его повторов.
2. **Алгоритм сжатия без потерь. Метод Хаффмана** – это метод сжатия информации на основе двоичных кодирующих деревьев был предложен Д. А. Хаффманом в 1952 году задолго до появления современного цифрового компьютера. Обладая высокой эффективностью, он и его многочисленные адаптивные версии лежат в основе многих методов, используемых в алгоритмах кодирования.

Таким образом, теперь мы знаем об архиваторах достаточно информации. По ходу всей ПЗ мы будем узнавать больше и больше информации на данную тему.

2. Структурное проектирование

В данном разделе описана структурное проектирование приложения, т.е. структурная схема работы приложения. Также сама структурная схема представлена в приложении А.

Структурное проектирование понимается как методология построения алгоритмов, программ и систем, в том числе информационных, в основе которой лежит выявление структуры задачи, определение составляющих компонент и выделение связей между ними. Процесс разделения сложных задач (объектов, систем) на относительно независимые друг от друга подзадачи (части, подсистемы) называется декомпозицией.

Следовательно, любая программа состоит из трёх базовых управляющих конструкций: *последовательность, ветвление, цикл*; кроме того, используются *подпрограммы*. При этом разработка программы ведётся пошагово, методом «сверху вниз».

Работа данной программы разбита на четыре основных блока: программный интерфейс приложения (API), контроллеры управления (программное управление), файловая система хранения информации и представление информации. Все блоки между собой связаны.

API определяет функциональность, которую предоставляет программа, при этом API позволяет абстрагироваться от того, как именно эта функциональность реализована. Этот блок служит для распознавания программой нужного исполняемого функционала. Если программу рассматривать как чёрный ящик, то API — это множество «ручек», которые доступны пользователю данного ящика и которые он может вертеть и дёргать. Программные компоненты взаимодействуют друг с другом посредством API. Когда интерфейс определяет, что именно должно быть использовано, то он обращается в блок контроллеров управления. Который в дальнейшем запоминает информацию/данные или забирает ее/их и предоставляет ее/их. Данный блок создан для удобства работы программиста, чтобы он мог работать с функционалом встраиваемым модулем и не думать о реализации этого функционала.

Блок контроллеров управления служит для исполнения команд, обработанных в блоке API. Таким образом организована последовательность выполнения программы, где один блок получает команду, обрабатывает, а другой блок выполняет и передаёт следующему блоку.

Блок файловой системы хранения информации служит для хранения данных. В данной работе требуется сжатие файлов, поэтому используются сжатые файлы и файлы, которые необходимо сжать (определяются пользователем), также распакованные (декомпрессированные) файлы.

Блок представления информации служит для отображения хранимой информации. Этот блок связан с контроллером управления. Когда блок управления сообщает, что поступил запрос о получении какой-либо информации, то этот блок отображает её.

Таким образом данный модуль можно представить, как пять основных блоков, которые связаны между. Каждый блок представляет маленькие подпрограммы. Такой способ разделения и взаимодействия обеспечивает понимание работы программы как одно целое.

3. Функциональное проектирование

Так как в данной работе используется объектно-ориентированный подход программирования, то ниже приведены некоторые из классов, использованных в программе (диаграмма классов представлена в приложении Б):

1. Класс `MainWindow` – это главное окно графического интерфейса архиватора, с помощью которого пользователь может удобно взаимодействовать с программой.

- Поля (private):

`-ui:Ui::MainWindow*` – экземпляр главного класса (`MainWindow`).

`-m_mapPages:QMap<int,QDialog*>` – карта, которая хранит пару (ключ, значение) и обеспечивает быстрый поиск значения. Необходима генерации показа новых окон – “AddWindow” или “ExtractWindow”.

`-model: QFileSystemModel*` – доступ к локальной файловой системе.

`-toolBarLineEdit:QLineEdit*` – панель инструментов для показа текущего пути.

`-spacer: QWidget*` – фиксатор для прокрутки страницы.

`-toolBarTextBeforeChanged:QString` – строка, которая запоминает конечный текущий путь.

- Методы (private):

`-init():void` – метод, который инициализирует окна – “AddWindow” или “ExtractWindow”.

- Методы (public):

`-MainWindow(parent: QWidget* = nullptr):explicit` – конструктор.

`-synchronizeListFromEdit():void` – синхронизация текущего пути в панели инструментов и “ListView” – отображение данных.

`-createFileSystemModel():void` – создаёт модель файловой системы.

`-setQLineEditToToolBar():void` – устанавливает текст “QLineEdit” в панель инструментов “QToolBar” для показа текущего пути.

`-MainWindow(): ~` – деструктор.

- Методы (signals):

`-emitText(text: const QString&):void` – сигнал, который нужен для передачи нужного текста и соединения с окнами “AddWindow” и “ExtractWindow”.

- Методы (private slots):


`-on_actionAdd_files_to_archive_triggered():void` – слот, который обрабатывает нажатие на кнопку добавить файл. И показывает окно “AddWindow”.

`-on_actionExtract_to_triggered():void` – слот, который обрабатывает нажатие на кнопку добавить файл. И показывает окно “ExtractWindow”.

`-on_listView_clicked(index: const QModelIndex&):void` – слот, который обрабатывает одинарное нажатие на элемент в отображаемых данных “listView” и выводит результат в панель инструментов “QToolBar”.

`-on_listView_doubleClicked(index:const QModelIndex&):void` – слот, который обрабатывает двойное нажатие на элемент в отображаемых данных “listView” и открывает нужную директорию.

`-EnterPressed():void` – слот, который обрабатывает одинарное нажатие на клавишу “Enter”.

`-upOnLevel():void` – слот, который обрабатывает одинарное нажатие на кнопку “на шаг назад” .

`-on_actionDelete_file_triggered():void` – слот, который удаляет определённый файл.

Перечисление:

-Page{ADD_WINDOW, EXTRACT_WINDOW}:enum – константы для генерации окон – “AddWindow” или “ExtractWindow”.

2. Класс AddWindow – это окно графического интерфейса архиватора, с помощью которого пользователь может добавить определённый файл и выбрать способ сжатия (компрессии).

- Поля (private):

-ui:Ui::Add_window* – экземпляр окна “AddWindow”.

-filesToCompress:QStringList – список строк, который содержит путь к файлу.

-direct:QString – строка, которая содержит директорию.

-pathForQLineEdit:QString – строка, которую мы получаем из списка строк.

-compressorRLE: CompressorRLE – экземпляр класса “CompressorRLE”.

-compressorHuf: CompressorHuffman – экземпляр класса “CompressorHuffman”.

- Методы (public):

-Add_window(parent:QWidget* = nullptr):explicit – конструктор.

-Add_window():~ – деструктор.

- Методы (private slots)

-on_BrowseButton_clicked():void – слот, который помещает путь в поле “Browse”.

-on_AppendButton_clicked():void – слот, который помещает путь в поле “Append”.

-on_Ok_clicked():void – слот, обрабатывает нажатие на кнопку “Ok”.

-recvText(text:const QString&):void – слот, который соединяется с “MainWindow” для получения пути для выбранного файла.

3. Класс `ExtractWindow` – это окно графического интерфейса архиватора, с помощью которого пользователь может распаковать (декомпрессия) ранее запакованный файл.

- Поля (private):

`-ui:Ui::Extract_window*` – экземпляр окна “`ExtractWindow`”.

`-filesToCompress:QStringList` – список строк, который содержит путь к файлу.

`-direct:QString` – строка, которая содержит директорию.

`-pathForQLineEdit:QString` – строка, которую мы получаем из списка строк.

`-compressorRLE:CompressorRLE` – экземпляр класса “`CompressorRLE`”.

`-compressorHuf: CompressorHuffman` – экземпляр класса “`CompressorHuffman`”.

- Методы (public):

`-Extract_window(parent:QWidget*nullptr):explicit`
– конструктор.

`-Extract_window():~` – деструктор.

- Методы (private slots)

`-on_BrowseButton_clicked():void` – слот, который помещает путь в поле “`Browse`”.

`-on_AppendButton_clicked():void` – слот, который помещает путь в поле “`Append`”.

`-on_Ok_clicked():void` – слот, обрабатывает нажатие на кнопку “`Ok`”.

`-recvText(text:const QString&):void` – слот, который соединяется с “`MainWindow`” для получения пути для выбранного файла.

4. Класс `CompressorRLE` – алгоритм сжатия RLE (*Run-Length Encoding*).

- Методы (public):

`-compressRLE (compressFileName:QString&, rleFileName: QString&):void` – метод, который реализует алгоритм сжатия файла “RLE”.

`-decompressRLE (compressedRLE:QString&, decompressedFileName:QString&):void` – метод, который реализует алгоритм декодирования сжатого файла “RLE”.

5. Класс `CompressorHuffman` – алгоритм сжатия методом Хаффмана.

- Поля (private):

`-code:std::vector<bool>` – создаём вектор типа `bool` для проверки.

`-table:std::map<char, std::vector<bool>>` – создаём таблицу(map) типа `char` и пару ключ-значение для нее типа `bool` (для представления в виде бинарного дерева).

`-root:static Node*` – создаём указатель узел.

- Методы (public):

`-readAllBytesFile (filename:QString&, info:std::vector<char>&):void` – метод, который считывает все байты из файла.

`-compressHuffman (compressFileName:QString&, hufFileName:QString&):void` – метод, который будет реализовать сжатие данных.

`-decompressHuffman (hufFileName:QString&, decompressFileName:QString&):void` – метод, который будет реализовывать декодирование сжатого файла (распаковка).

`-buildTable (root:Node*):void` – метод, который создаёт таблицу символов для кодов Хаффмана.

`-Print (root:Node*, depth:int = 0):void` – метод, который выводит символы в консоль.

6. Класс `Node` – класс, который представляет собой “узел”, для реализации алгоритма Хаффмана.

- Поля (private):

-left:Node* - левый указатель на узел.

-right:Node* - правый указатель на узел.

-count:int - переменная, которая будет считать кол-во совпавших символов.

-symbol:char - переменная, которая будет хранить сам символ.

- Методы (public):

-Node() - конструктор без параметров, создаёт узел без параметров т.е. пустой (новый).

-Node(symbol:char, count:int) - конструктор, который устанавливает символ и кол-во совпадений и обнуляет левый и правый узел.

-Node(left:Node*, right:Node*) - конструктор, который создаёт узел где-то в середине и поэтому уже известны его ветви левая и правая.

4. Разработка программных модулей

В данном разделе будет рассматриваться реализация функционала, который используется программой для работы, а также алгоритм работы программы.

Для сжатия данных придумано множество техник. Большинство из них комбинируют несколько принципов сжатия для создания полноценного алгоритма. Даже хорошие принципы, будучи скомбинированы вместе, дают лучший результат.

В данной работе применяются два алгоритма сжатия:

1. Алгоритм сжатия RLE (*Кодирование длин серии*);
2. Алгоритм сжатия методом Хаффмана.

Коротко рассмотрим основной принцип этих двух алгоритмов:

Кодирование длин серии (RLE).

Это алгоритм заменяет серии из двух или более одинаковых символов числом, обозначающим длину серии, за которым идет сам символ. Полезен для сильно избыточных данных, типа картинок с большим количеством одинаковых пикселей, или в комбинации с алгоритмами типа BWT.

Пример:

На входе: AAABVCCCCDEEEEEEEAAAAAAAAAAAAAAAAAAAAA

На выходе: 3A2B4C1D6E21A

Алгоритм Хаффмана

Идея алгоритма: зная вероятность вхождения символов в сообщение, можно описать процедуру построения кодов переменной длины состоящих из целого количества битов. Символам с большей вероятностью присваиваются более короткие коды.

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

Рассмотрим алгоритм по шагам:

1. Символы входного алфавита образуют список свободных узлов. Каждый узел имеет кол-во вхождений символа в ожидаемое сообщение, сам символ, и два указателя – на левый и правый узел.

2. Для удобства можно отсортировать узлы (сортировать можно по кол-ву вхождений или по ascii-кодам символов).
3. Выбираются 2 наименьших свободных узла дерева.
4. Создается родительский узел, имеющий указатели на созданные 2 узла (левый и правый).
5. Родитель добавляется в список свободных узлов, а двое его потомков удаляются из этого списка.
6. Одной дуге выходящей из родителя ставится в соответствие бит 1, другой – бит 0.
7. Далее пункты повторяются, начиная со второго, до тех пор, пока в списке свободных узлов не останется только один свободный узел. Он и будет считаться корнем дерева.

Пусть входная последовательность имеет следующий вид - «It is my striiiiiing!!!!».

Прodelав все действия по алгоритму получим выходную последовательность в следующем виде (смотреть рисунок ниже):

I t ' ' i s ' ' m y ' ' s t r i i i

«11010100111011100101010110100101101001111111111

i i n g ! ! ! !

1111011101010000000000».

Изначальная, входная строка, состояла из 23 символов (23 байта), что показывает корень дерева (смотреть рисунок ниже).

А на выходе получаем строку, состоящую из 9 байт, (надо разбить всю последовательность по 8 начиная с конца).

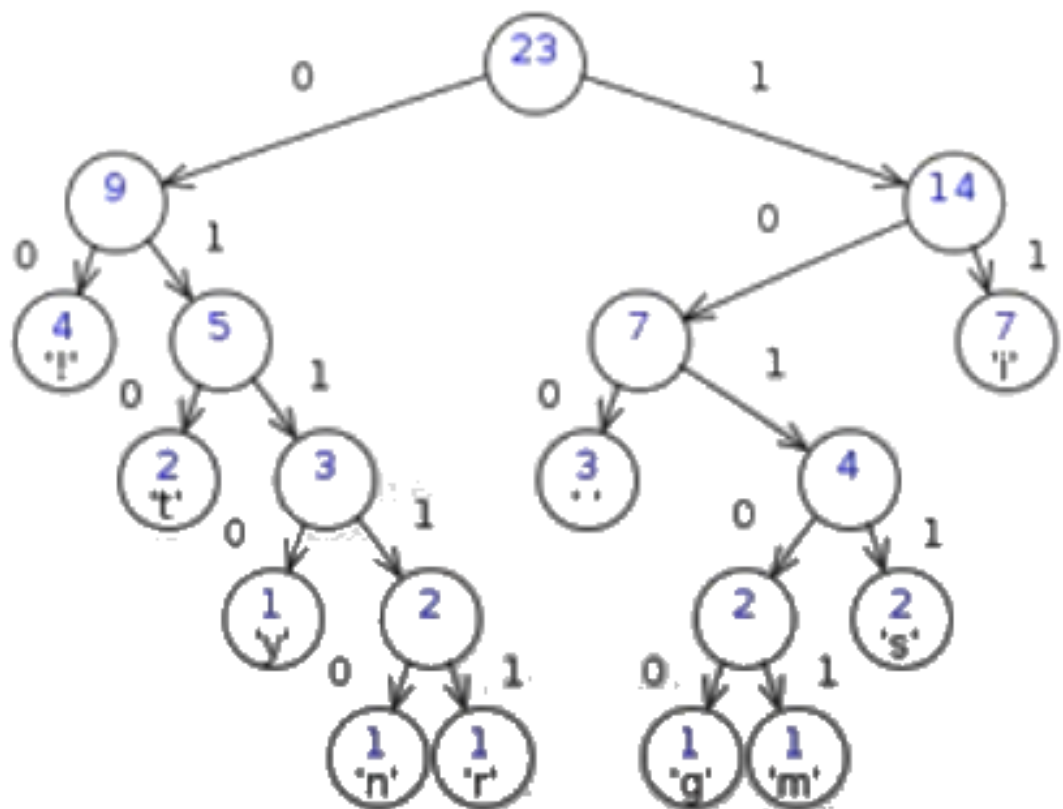
Фрагмент графического изображения алгоритма:

Input: it is my strlllling!!!!

| | |
|---|-------|
| | 100 |
| i | 00 |
| g | 10100 |
| i | 11 |
| m | 10101 |
| n | 01110 |
| r | 01111 |
| s | 1011 |
| t | 010 |
| y | 0110 |

Encoded Output

110101001110111001010101101001011



Рассмотрим основной функционал

Главное окно пользовательского интерфейса и основной функционал:

Основные элементы (методы или слоты) функционала - это конструктор главного окна, инициализация и создание окон “AddWindow” или “ExtractWindow”. Остальные элементы (методы или слоты) предназначены для внешнего вида и удобного взаимодействия (описания каждого из них см. выше).

Конструктор главного окна:

```
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this); //для построения дерева виджетов на родительском
    виджете

    init();           //инициализация окон ADD_WINDOW и EXTRACT_WINDOW
    createFileSystemModel(); //создание фаловой системы
    setQLineEditToToolBar(); //взаимодействие с панелью инструментов
    QLineEditToToolBar

    //Связываем между собой объекты используя сигналы и слоты
    connect(toolBarLineEdit, SIGNAL(returnPressed()), this,
    SLOT(EnterPressed()));
    connect(this->ui->actionUp_on_level, SIGNAL(triggered()), this,
    SLOT(upOnLevel())); //связываем upOnLevel() с actionUp_on_level
    connect(this, SIGNAL(emitText(const QString)),
    m_mapPages.value(ADD_WINDOW), SLOT(recvText(const QString)));
    //связываем ADD_WINDOW и MainWindow для установки пути
    connect(this, SIGNAL(emitText(const QString)),
    m_mapPages.value(EXTRACT_WINDOW), SLOT(recvText(const QString)));
    //связываем ADD_WINDOW и MainWindow для установки пути
}
```

Инициализация и создание окон “AddWindow” или “ExtractWindow”:

```
void MainWindow::init()
{
    m_mapPages.insert(ADD_WINDOW, new Add_window); //добавляем элементы
    Add_window
    m_mapPages.insert(EXTRACT_WINDOW, new Extract_window); //добавляем
    элементы Extract_window

    for(auto it = m_mapPages.begin(); it != m_mapPages.end(); ++it)
    {
        it->value()->setModal(true); // инициализируем
    }
}
```

Методы создания окон реализованы одинаково, поэтому рассмотрим на примере один из них:

```

void MainWindow::on_actionExtract_to_triggered()
{
    QString path = toolBarLineEdit->text();
    QStringList pathList = path.split(".");

    //создаём экземпляры классов для проверки и получения нужного нам
    файла и пути к нему
    QItemSelectionModel *selectionModel = ui->listView-
    >selectionModel();
    QModelIndexList selectedIndexes = selectionModel->selectedIndexes();
    //проверяем в какой мы директории
    if(selectedIndexes.size() <= 0 ||
        toolBarLineEdit->text() == "C:/" ||
        toolBarLineEdit->text() == "D:/" ||
        toolBarLineEdit->text() == "E:/") {
        //если мы не заходили не в какую директорию, то путь указываем
        сами в новом окне
        m_mapPages.value(ADD_WINDOW)->exec();
    } //иначе проверяем правильный ли файл выбран
    else if(pathList.back() == "txt" || pathList.back() == "cpp" ||
    pathList.back() == "c")
    {
        m_mapPages.value(ADD_WINDOW)->show();
        qDebug()<<toolBarLineEdit->text();
        emit emitText(toolBarLineEdit->text());
    } else { //если файл не выбран, то выводим оу этом сообщение
        QMessageBox msgBox(QMessageBox::Warning, "Warning", "Choose
        .txt/.cpp/.c file");
        msgBox.setWindowFlags(Qt::Dialog | Qt::CustomizeWindowHint |
        Qt::WindowTitleHint | Qt::WindowCloseButtonHint);
        msgBox.exec();
    }
}

```

Теперь рассмотрим отдельно каждый алгоритм сжатия данных

Реализация алгоритма сжатия RLE:

```

void CompressorRLE::compressRLE(QString &compressFileName, QString
&rleFileName)
{
    //создаём объекты для чтения и записи переданные в параметрах
    ifstream fileInput(compressFileName.toStdString());
    ofstream fileCompressed(rleFileName.toStdString());
    char sym1; //первый символ (байт) в файле

    while(fileInput.good())
    {
        int count=1; //кол-во совпавших символов
        fileInput.get(sym1); //считываем первый символ из файла fileInput
        в sym1
        while (sym1 == fileInput.peek() && count < 9) //пока первый символ
        равен следующему за ним и кол-во меньше 9 - повторяем
        {
            //увеличиваем кол-во совпадений и считываем второй символ в
            sym1
            count++;
        }
    }
}

```

```

        fileInput.get(sym1);
    }
    //записываем результат в файл для записи
    fileCompressed << sym1 << count;
}
//закрываем оба файла
fileInput.close();
fileCompressed.close();
}

void CompressorRLE::decompressRLE(QString &compressedRLEfileName,
QString &decompressedFileName)
{
    //создаём объекты для чтения и записи переданные в параметрах
    ifstream fileToDescompress(compressedRLEfileName.toStdString());
    ofstream fileDecompressed(decompressedFileName.toStdString());
    char ch1, ch2; //первый(сам символ) и второй(кол-во символов, т.е
число) символ в файле
    fileToDescompress.get(ch1); //считываем первый символ
    while(fileToDescompress.good()) //проверяем не дошли ли мы до конца
файла
    {
        fileToDescompress.get(ch2); //считываем второй
        for (int i = 0; i < ch2 - '0'; i++){ //пока не запишем все
повторяющиеся символы не выйдем
            fileDecompressed << ch1;
        }
        fileToDescompress.get(ch1); //считываем следующий символ
    }
    //закрываем оба файла
    fileToDescompress.close();
    fileDecompressed.close();
}

```

Надо сказать, что данный алгоритм считается самым простым, как говорилось ранее, и является хорошим алгоритмом для повторяющейся последовательности байтов.

Реализация алгоритма сжатия методом Хаффмана:

В данном методе необходимо реализовать класс узел, который будет формировать дерево Хаффмана.

```

class Node
{
public:
    Node *left;
    Node *right;
    int count;
    char symbol;
    Node() {left = right = 0;} // создаёт узел без параметров то есть пустой
(новый)

    Node(char symbol, int count) //конструктор, для инициализации // key
= symbol; value = count

```

```

    {
        this->count = count;
        this->symbol = symbol;
        //обнуляем левый и правый узел
        this->left = 0;
        this->right = 0;
    }

    Node(Node *left, Node *right)// создают узел где-то в середине и
    поэтому уже известны его ветви левая и правая
    {
        this->symbol = 0;
        this->left = left;
        this->right = right;
        count = left->count + right->count;// переменная кол-во элементов
на ветве
    }
};

```

Теперь реализация класса CompressorHuffman:

```

Node *CompressorHuffman::root = 0;//обнуляем первый узел

struct Compare //структура перегрузки оператора () и сравнения, для
сортировки указателей листа
{
    bool operator()(const Node* left, const Node* right)
    {
        return left->count < right->count;
    }
};

//функция чтения всех байтов из файла
void CompressorHuffman::readAllBytesFile(QString &fileName,
std::vector<char> &info){
    ifstream file(fileName.toStdString(), ios::binary |
ios::ate);//открываем файл
    if (file.is_open())//если все хорошо и файл открыт
    {
        ifstream::pos_type pos = file.tellg();//устанавливаем текущую
позицию
        info.resize(pos);//изменяем размер строки, чтобы она могла
содержать pos символов
        file.seekg(0, ios::beg);//ставим на начало
        file.read(&info[0], pos);//читаем весь файл в info[0]
    }
    else //иначе ошибка открытия файла
    {
        QMessageBox msgBox(QMessageBox::Warning, "Error", "Unable to open
file");
        msgBox.setWindowFlags(Qt::Dialog | Qt::CustomizeWindowHint |
Qt::WindowTitleHint | Qt::WindowCloseButtonHint);
        msgBox.exec();
    }
    file.close();
}

```

```

//функция сжатия данных
void CompressorHuffman::compressHuffman(QString &compressFileName,
QString &hufFileName)
{
    vector<char> info;//вектор ,который будет содержать символы
    readAllBytesFile(compressFileName, info);//читаем все символы в
вектор
    map<char, int> symbols;//карта, которая содержит символ и кол-во
вхождений этого символа
    for(auto it : info)
    {
        symbols[it]++;//заполняем карту <символом, кол-во вхождений
символа>: 1 - 'a':5, 2 - 'c':1, ...
    }

    list<Node*> tree;//создаём лист указателей на узел - дерево
    map<char, int>::iterator it;// итератор для прохода
    for (it = symbols.begin(); it != symbols.end(); it++)// заполняем
дерево
    {
        Node *p = new Node(it->first, it->second); // key = symbol; value
= count
        tree.push_back(p);
    }

    //Формируем закодированное бинарное дерево - цикл, до тех пор, пока
не останется один корень
    while(tree.size() != 1)
    {
        tree.sort(Compare());//сортируем лист по кол-ву вхождений: 1 -
'c':1, 2 - 'a':5, ...

        //выбираем первые два узла, т.к. они минимальные и запоминаем их
        Node *tmp1 = tree.front();
        tree.pop_front();//удаляем первый
        Node *tmp2 = tree.front();
        tree.pop_front();//удаляем второй

        Node *parent = new Node(tmp1, tmp2);// формируем один узел с
указателями на те, что удалили, но запомнили в tmp1 и tmp2
        tree.push_back(parent);//добавляем сформированный узел и
повторяем опять цикл
    }
    this->root = tree.front();//помещаем последний корень (вершину) в root
    Print(this->root);//выводим символы в консоль

    buildTable(this->root);//формируем таблицу
    ifstream fileCodes(compressFileName.toStdString());//создаём объект,
для чтения и кодировки символа
    ofstream fileCompressed(hufFileName.toStdString(), ios::out |
ios::binary);//создаём объект для записи сжатых данных
    int count = 0;//для подсчёта битов
    char buffer = 0;//для закодированного символа(байта)
    //пока не конец файла заполняю файл - fileCompressed моими кодами из
таблицы
    while(!fileCodes.eof())

```

```

    {
        //получаю символ, который нужно закодировать
        char ch = fileCodes.get();
        vector<bool> x = table[ch]; //записываю вектор данного символа
        for(int i = 0; i < (int)x.size(); i++) //пока не пройдем по коду
из таблицы (x.size)
        {
            //разбиваем наши байты по 8
            buffer = buffer | x[i] << (7-count); //проверяем биты и
записываем их в байт (7 - count, т.к. мы проверяем слева направо)
            count++;
            //если прошли 8 битов, то записываем байт в файл
fileCompressed
            if(count == 8)
            {
                count = 0; //обнуляем, для считывания следующих 8 битов (1
байт)
                fileCompressed << buffer;
                buffer = 0;
            }
        }
    }
    //закрываем файлы
    fileCodes.close();
    fileCompressed.close();
}

```

```

//функция формирования таблицы
void CompressorHuffman::buildTable(Node *root)
{
    if (root->left) //если слева не NULL (т.е. слева есть сын)
    {
        code.push_back(0); //ставим вектор 0
        buildTable(root->left); //и опять рекурсивно запускаем функцию
root->left
    }

    if (root->right) //если справа не NULL (т.е. справа есть сын)
    {
        code.push_back(1); //ставим вектор 1
        buildTable(root->right); //и рекурсивно запускаем функцию с root-
>right
    }

    if (root->symbol) //если нашли букву
        table[root->symbol] = code; //ассоциируем букву с символом

    if (code.size()) //Если размер кода не NULL
        code.pop_back(); //то уменьшаем на 1, чтобы вернуться назад и
проверить остальное дерево
}

//функция декодирования сжатых данных
void CompressorHuffman::decompressHuffman(QString &hufFileName, QString
&decompressFileName)

```

```

{
    //создаём объекты для чтения и записи
    ifstream file(huffFileName.toStdString(), ios::in | ios::binary);
    ofstream decompressFile(decompressFileName.toStdString());
    //узел, в который присваиваем наш корень от которого мы будем идти по
    дереву
    Node *p = this->root;
    int count = 0;
    char byte;
    //считываем байт
    byte = file.get();
    while(!file.eof())
    {
        bool b = byte &(1 << (7 - count)); //проверяем биты и получаем 0
или 1 (7 - count, т.к. мы проверяем слева направа)
        //проверяем в какую сторону идти: 1-право, 0-лево
        if(b)
            p = p->right; //переходим вправо
        else
            p = p->left; //переходим влево
        //если дошли до символа, т.е. конца какого-либо уза
        if(p->left == nullptr && p->right == nullptr)
        {
            decompressFile<<p->symbol; //записываем символ в файл
            p = root;
        }
        count++;
        //если прошли 8 битов, то обнуляем count и считываем дальше
        if(count == 8)
        {
            count = 0;
            byte = file.get();
        }
    }
    //закрываем файлы
    file.close();
    decompressFile.close();
}

```

В итоге мы получаем алгоритм работы программы:

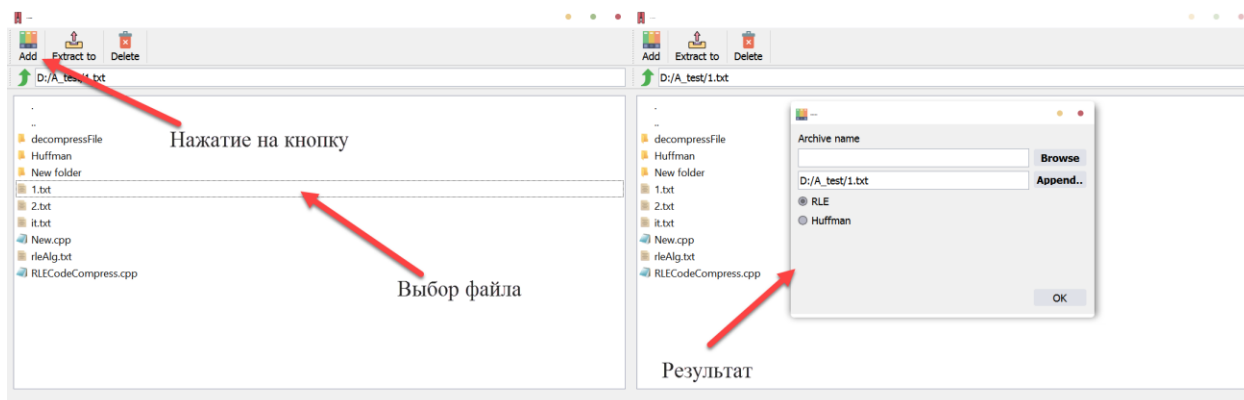
1. Начало программы. Взаимодействие пользователя с графическим интерфейсом
2. Получение файла для архивации
3. Определение алгоритма
4. Реализация того или иного алгоритма
5. Создание заархивированного файла
 - а. При необходимости получение файла для разархивации
 - б. Выбор соответствующего алгоритма декомпрессии, в зависимости от файла
 - с. Создание разархивированного текстового файла
6. Конец программы

5. Тестирование программы и руководство пользователя

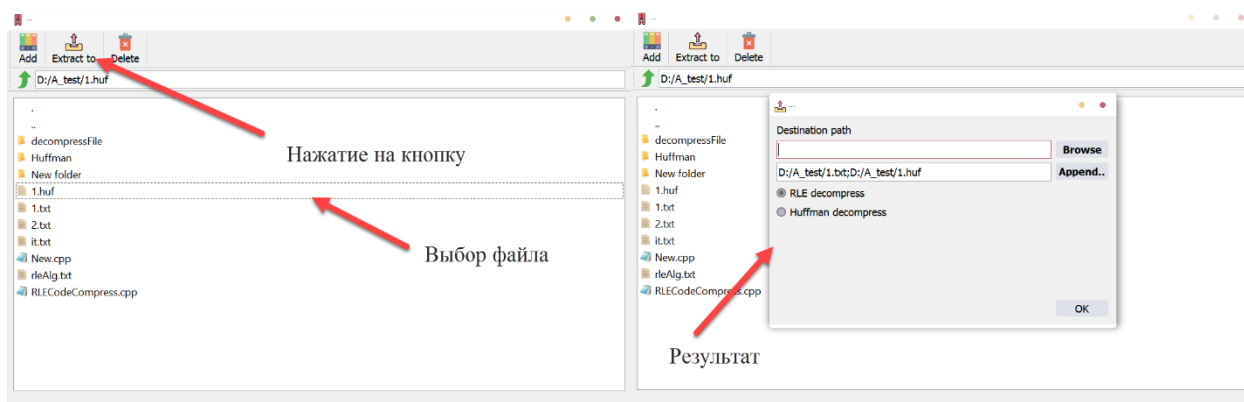
Проведём тестирование некоторых команд Архиватора.

| Команда | Действие | Ожидаемый результат | Успешность тестирования |
|--------------------------|---|---|-------------------------|
| 1 | 2 | 3 | 4 |
| Add | <p>Нажатие на кнопку в левом верхнем углу</p>  | Появляется окно, в котором необходимо выбрать файл для архивации, место, куда сохранить сжатый файл и способ сжатия | Успешно |
| Extract | <p>Нажатие на кнопку в левом верхнем углу</p>  | Появляется окно, в котором необходимо выбрать архив(определённого сжатия), место, куда необходимо сохранить разархивированный файл и способ декодирования | Успешно |
| Delete | <p>Нажатие на кнопку в левом верхнем углу</p>  | Происходит удаление выбранного пользователем элемента, в той или иной директории | Успешно |
| Create compressed file | После выполнения первой команды (Add) появляется новое окно для создания архива | Создание и размещения в нужном месте заархивированного файла (определённым алгоритмом) | Успешно |
| Create decompressed file | После выполнения первой команды (Extract) появляется новое окно для открытия архива | Создание и размещение разархивированного файла | Успешно |

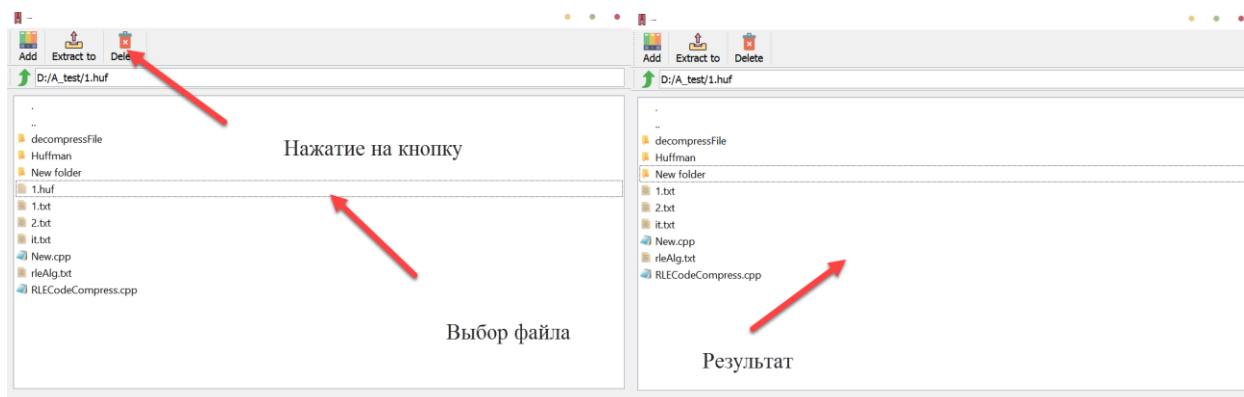
Тестирование команды Add:



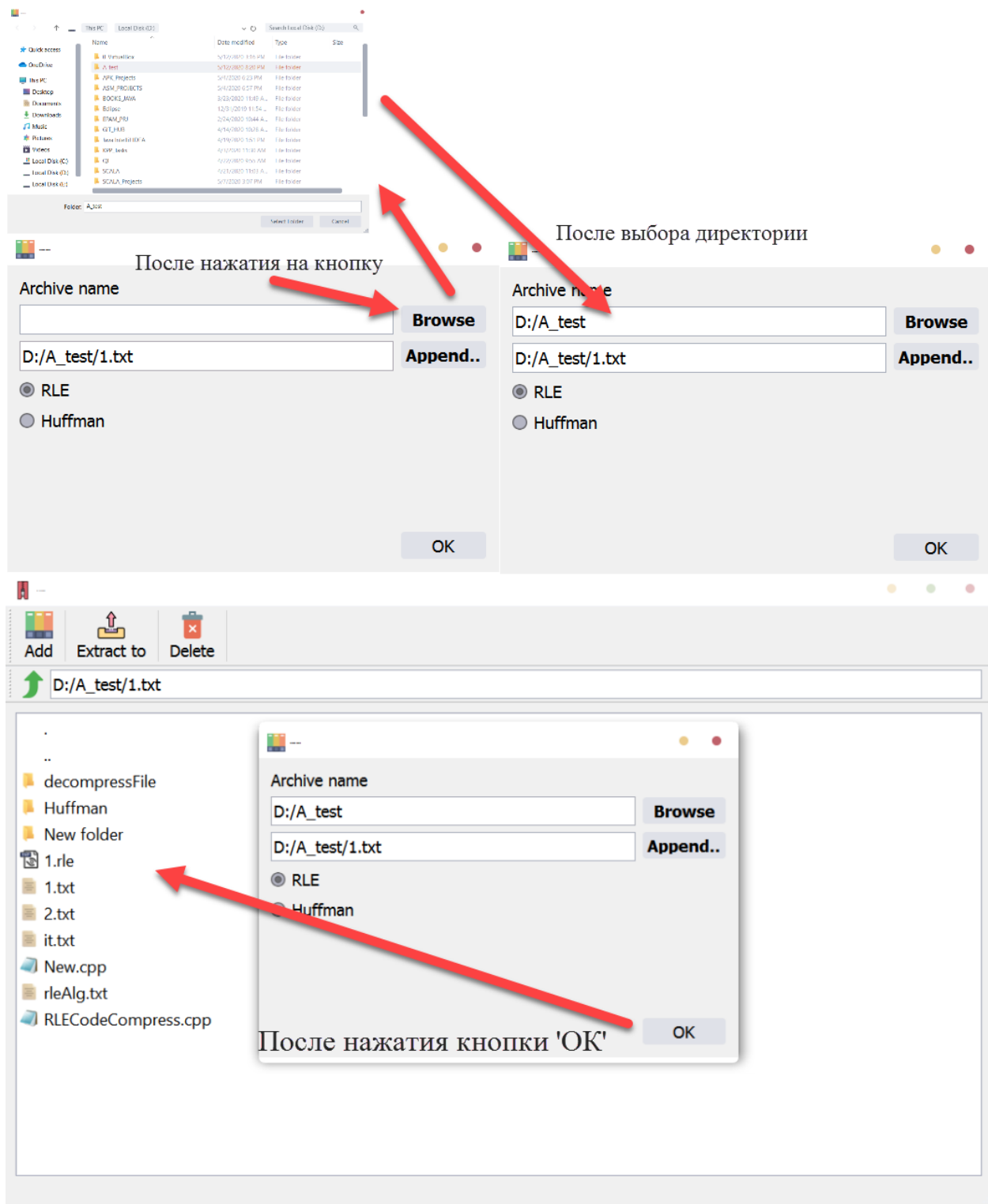
Тестирование команды Extract:



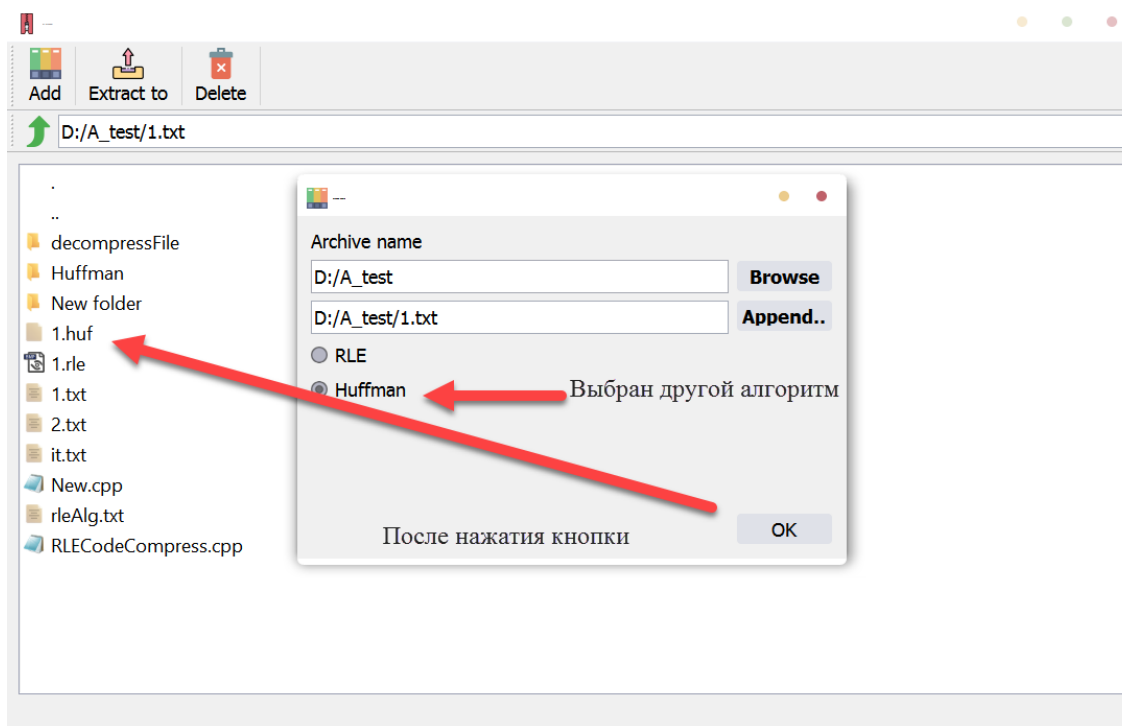
Тестирование команды Delete:



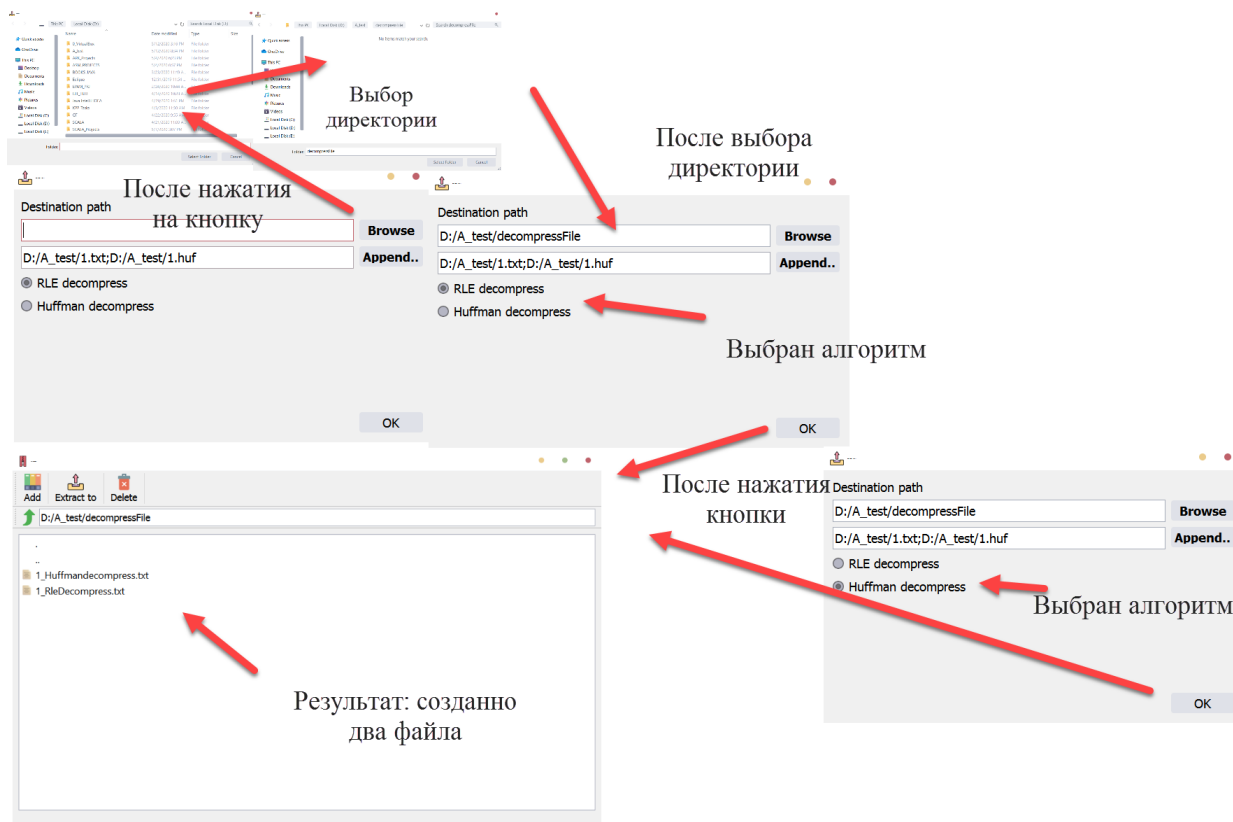
Тестирование команды Create compressed file:



Также можно выбрать другой алгоритм сжатия (Huffman)

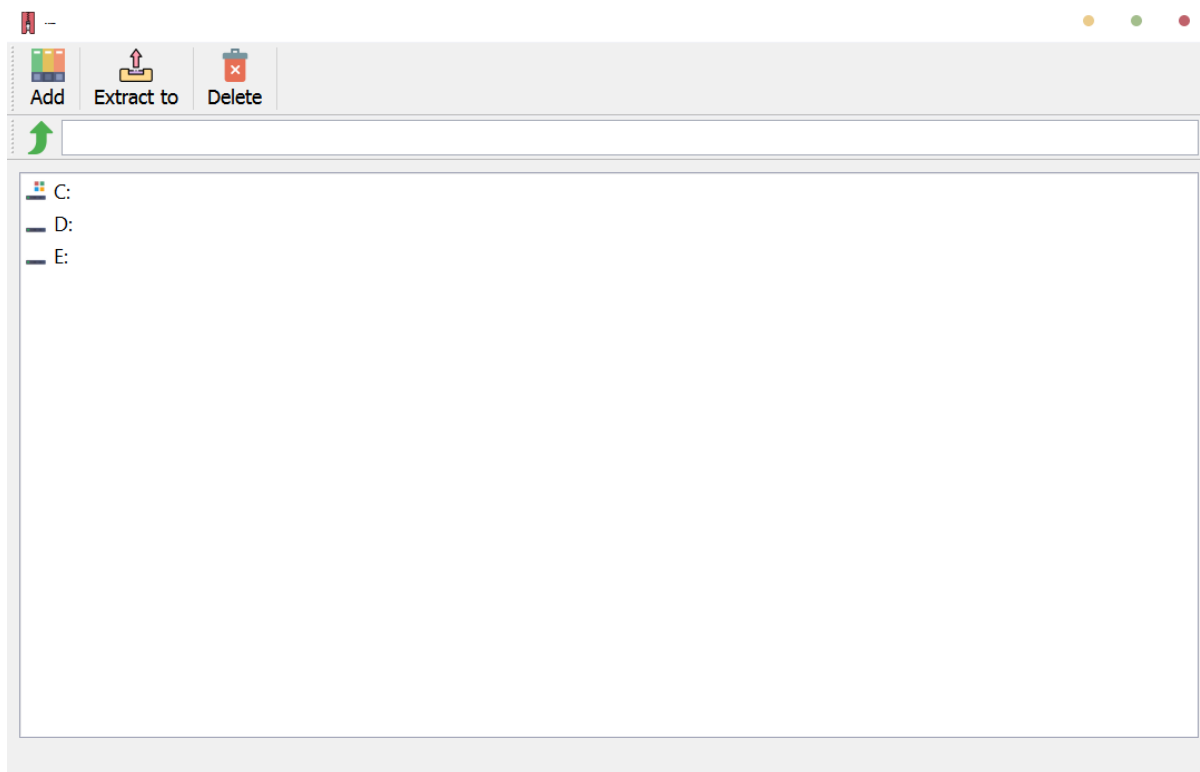


Тестирование команды Create decompressed file:

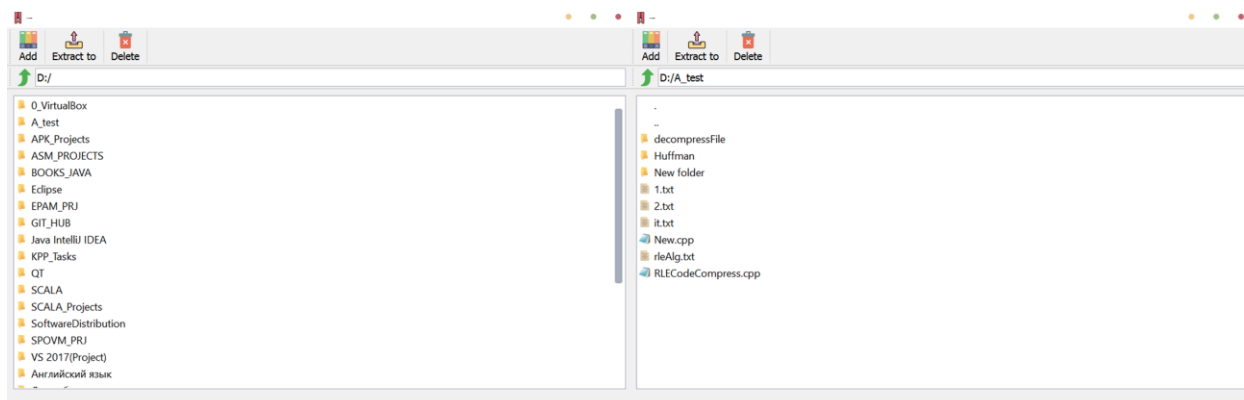


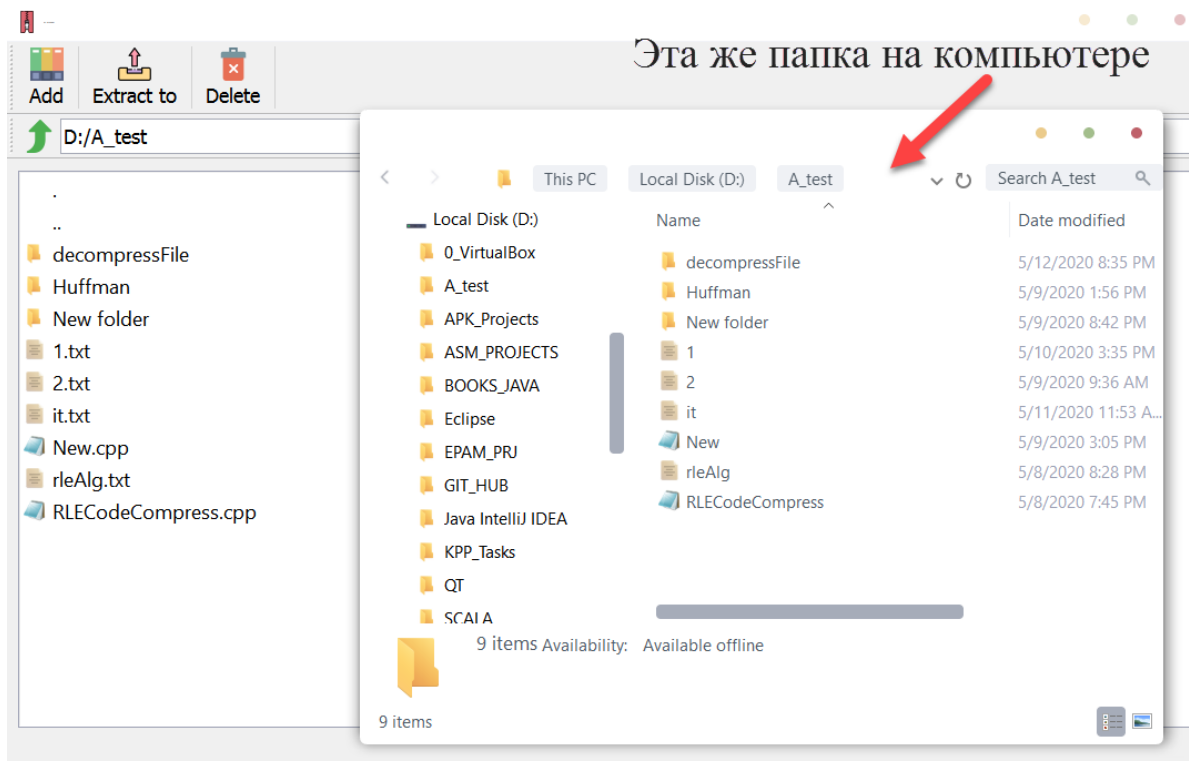
Теперь рассмотрим весь процесс создания сжатого файла и его декодирование:

1. Запуск программы



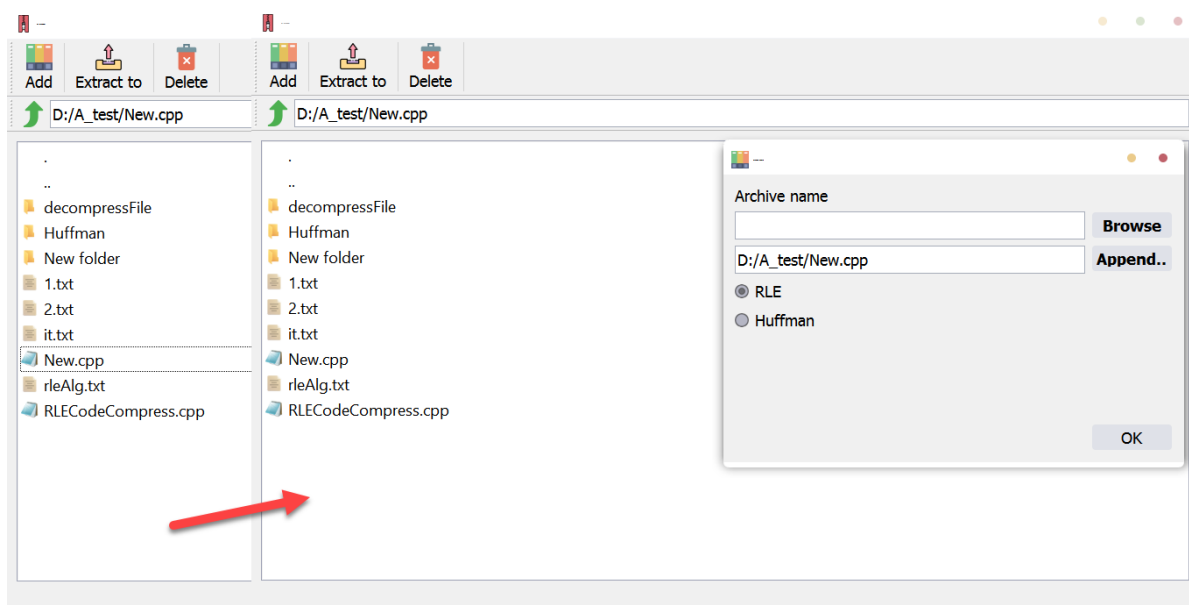
2. Поиск нужной директории с файлом



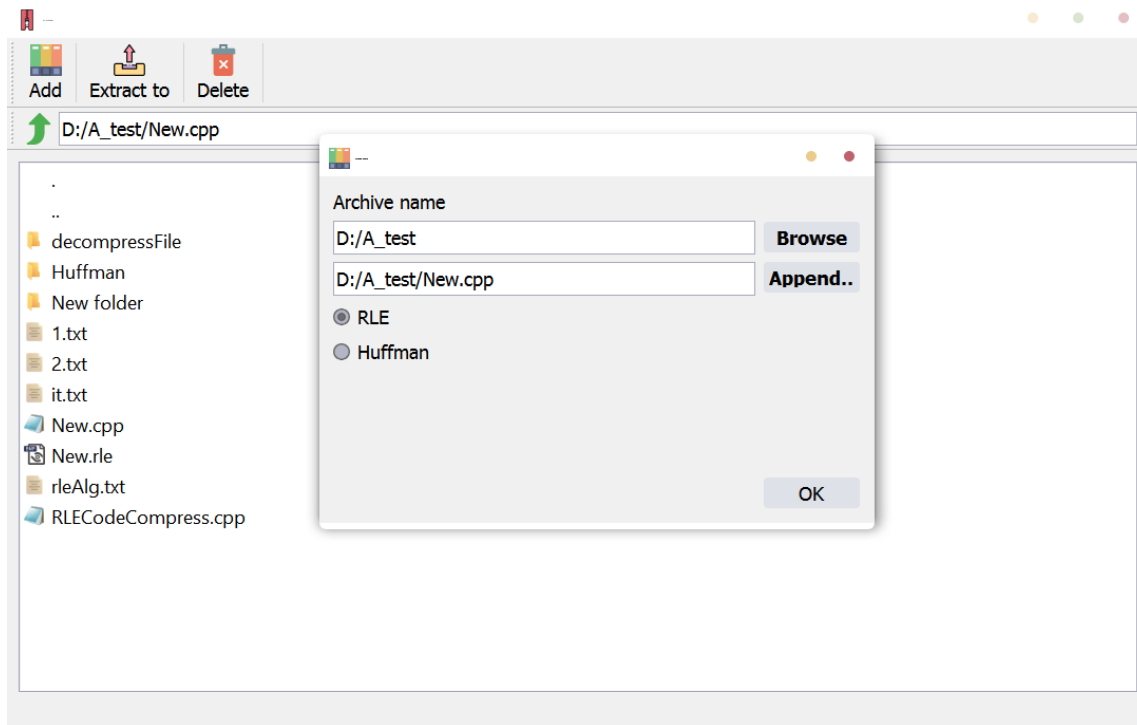


3.Проведём алгоритм сжатия на примере файла .txt, так же можно сжимать любые файлы текстового формата, такие как - .cpp, .java и .txt (при желании можно добавить еще какие-нибудь):

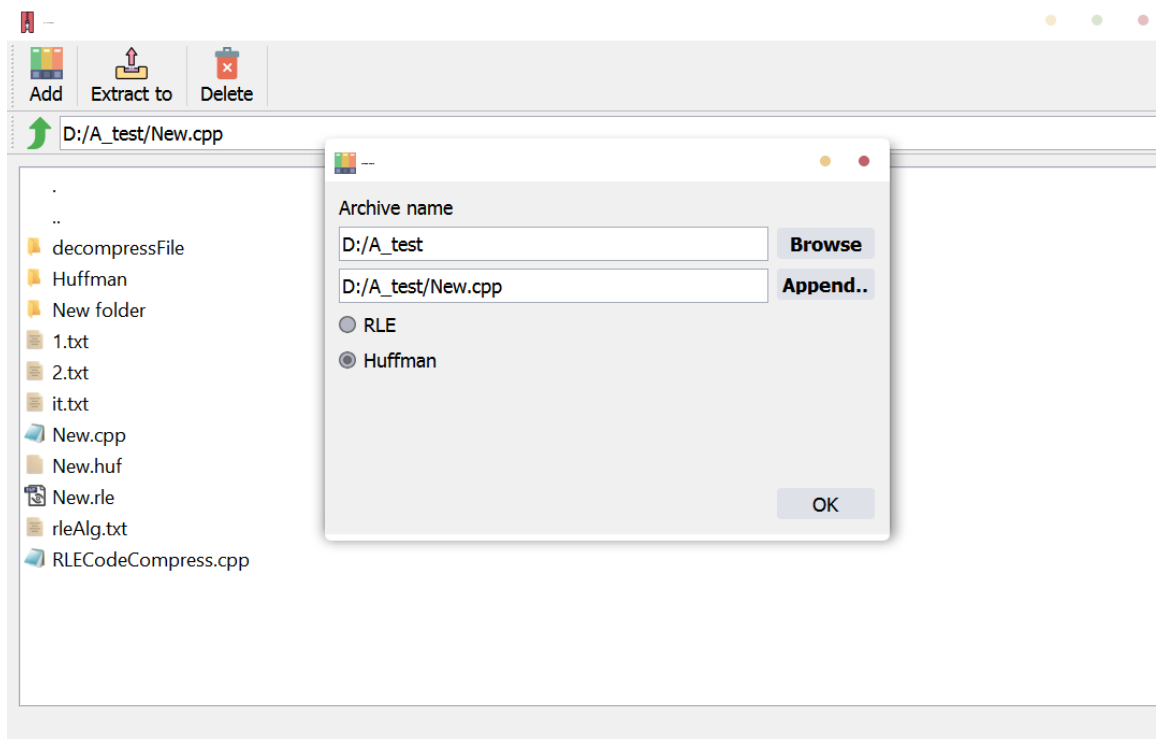
3.1. Выбор файла

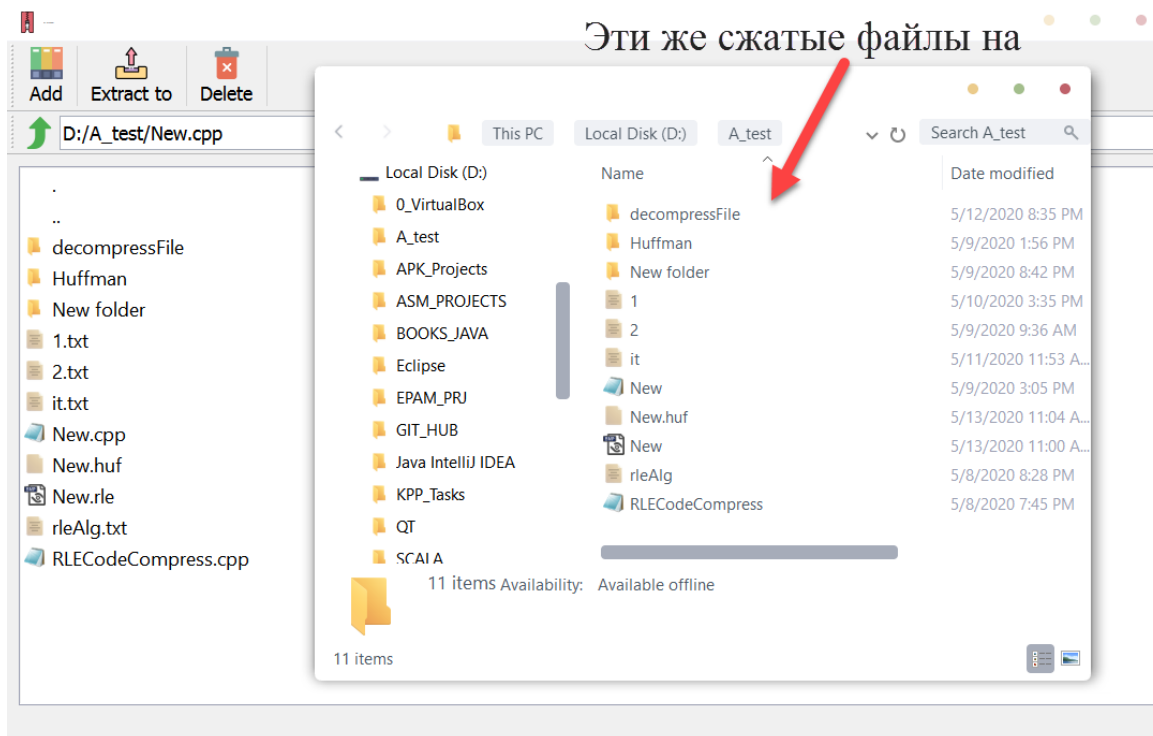


3.2. Заполнение окна и создание сжатого файла по алгоритму RLE

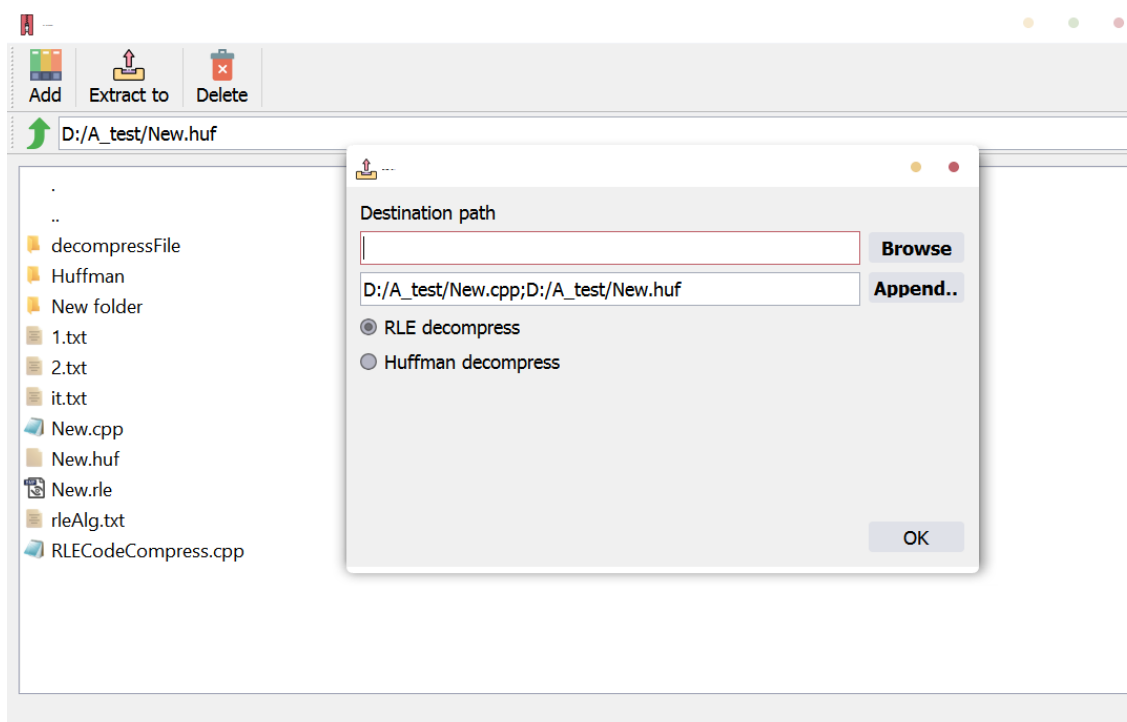


3.3. Выбор алгоритма Хаффмана для того же файла и создание сжатого файла

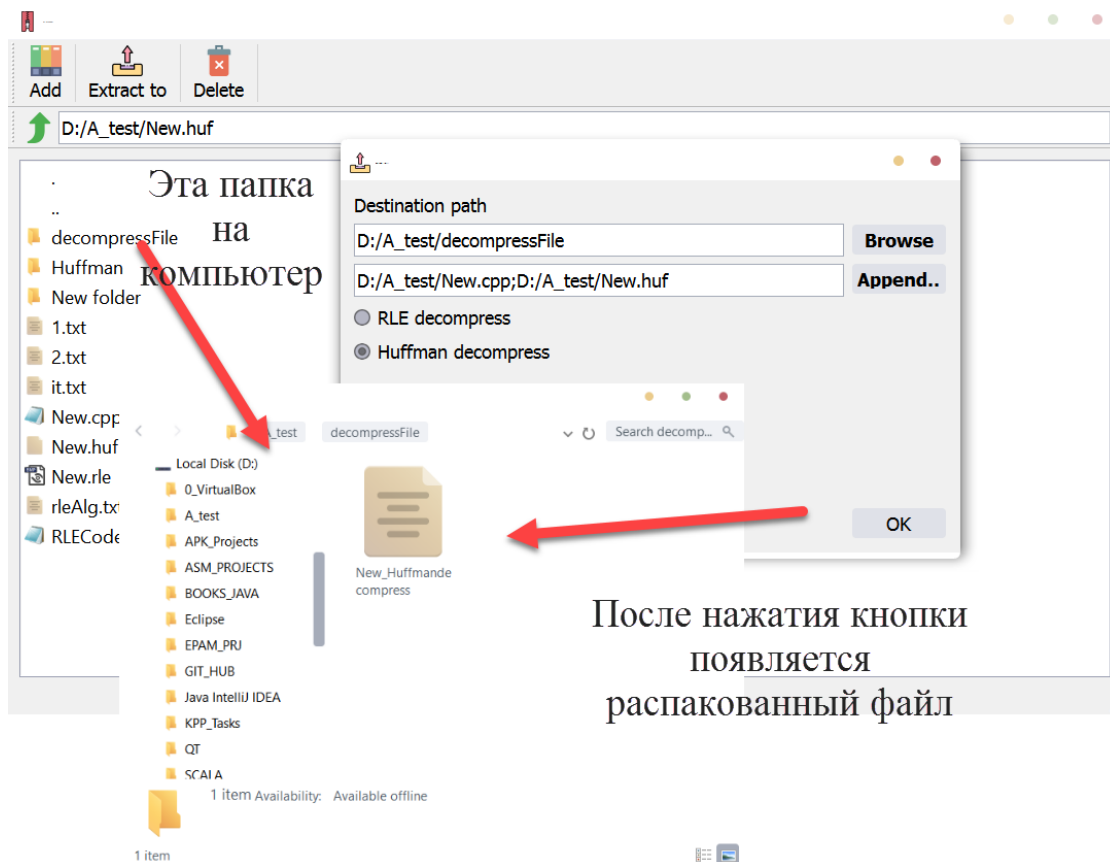
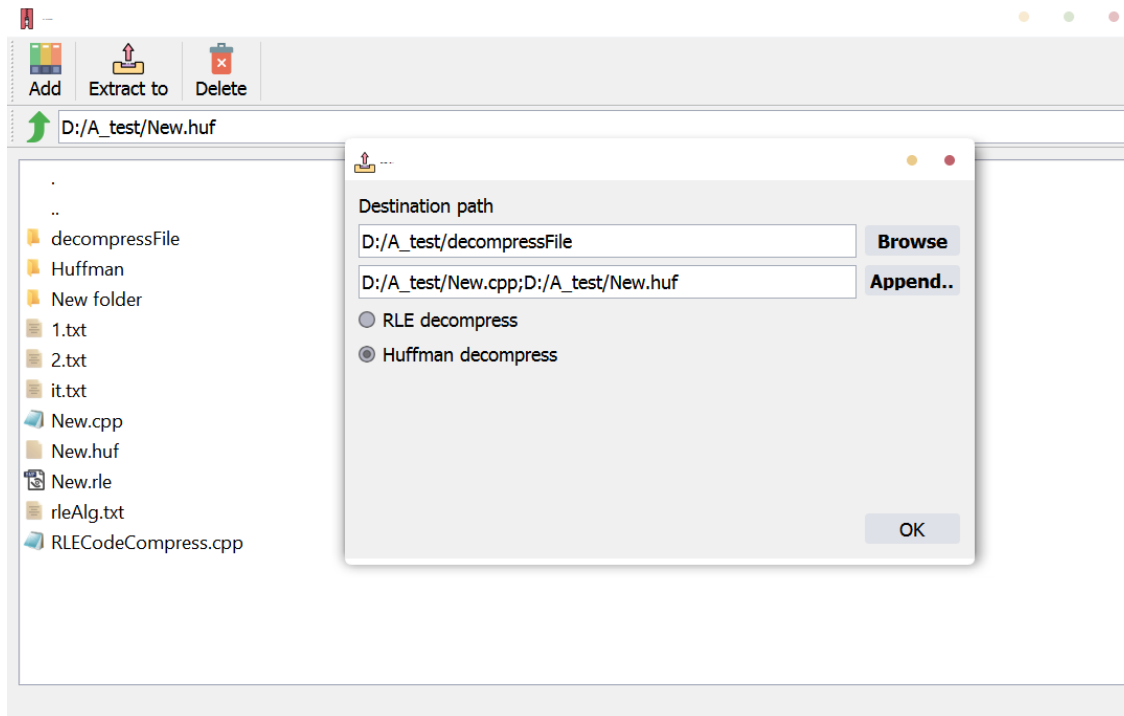




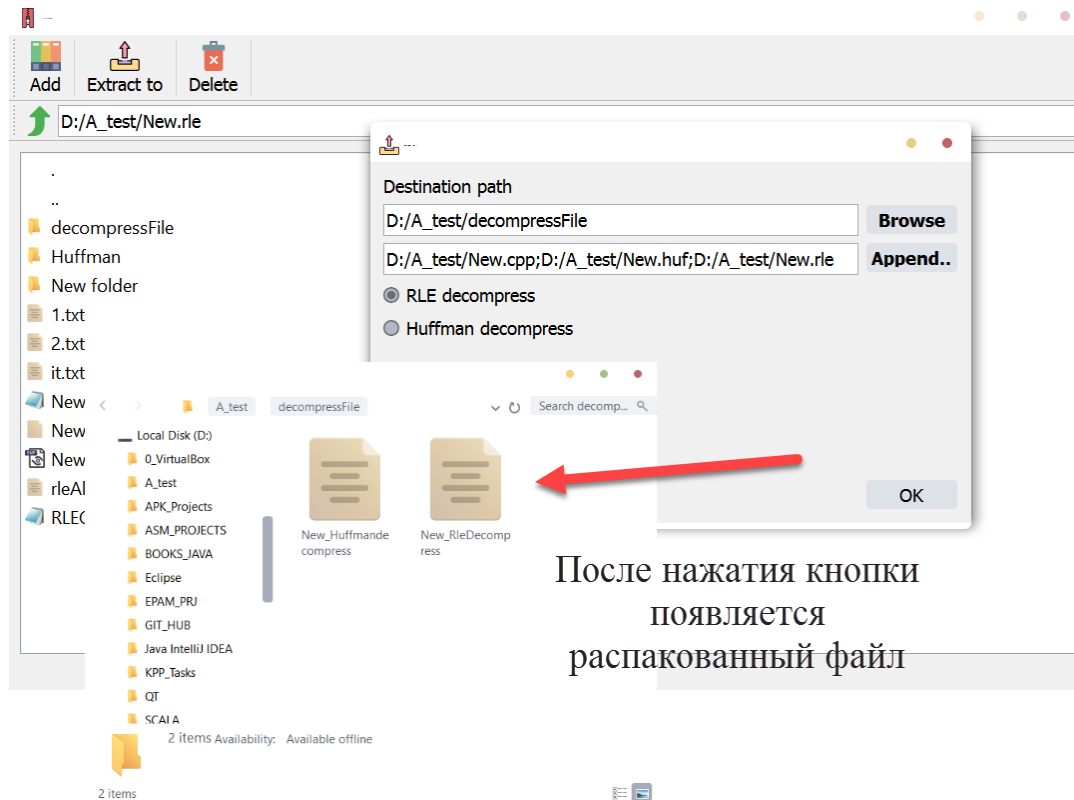
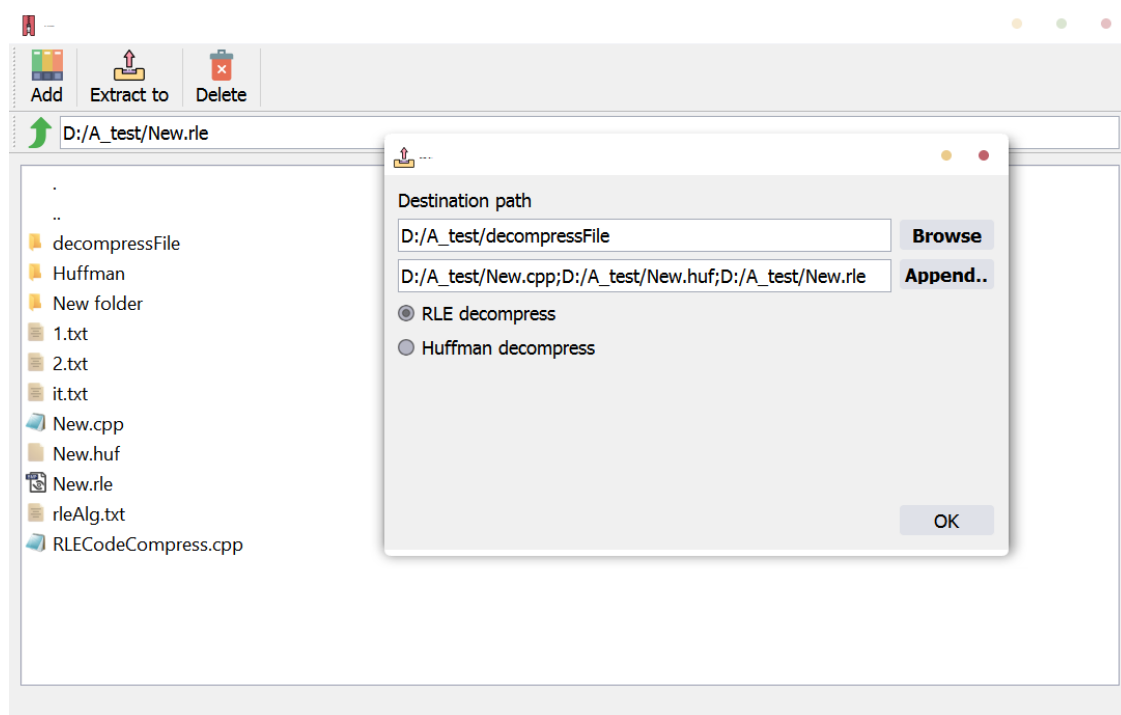
4. Теперь рассмотрим алгоритм декомпрессии, он похож по действиям



4.1. Сначала распакуем файл .haf по алгоритму - Хаффмана (важно не забыть указать тот алгоритм декодирования и выбрать место)



4.2. Теперь распакуем файл .rle по соответствующему алгоритму



Таким образом происходит создание сжатого файла и его декодирование.

Более подробный и наглядный пример можно посмотреть по ссылке:

<https://youtu.be/JKxpSfTRkAM>

ЗАКЛЮЧЕНИЕ

В итоге мы получили удобное приложение для сжатия данных – «Архиватор».

Плюсы данного приложения:

- удобный и понятный пользовательский интерфейс;
- возможность выбора того или иного алгоритма (одного из двух).

Минусы данного приложения:

- обработка файлов формата текста, например, .txt, .cpp, .json и другие (обработка таких форматов, как .doc, .pdf и т.п. не поддерживается).

Список использованных источников

Форумы и сайты:

[1] <https://www.youtube.com/?gl=BY>

[2] <http://www.cplusplus.com/forum/beginner/28542/>

[3] http://mf.grsu.by/UchProc/livak/en/po/comprsite/pract_rle.html

[4] http://mf.grsu.by/UchProc/livak/po/comprsite/theory_huffman.html

[5] <http://algolist.ru/compress/standard/huffman.php>

[6] <https://habr.com/ru/post/132683/>

Литература:

[7] В.А. Скляров «Язык С++ и объектно-ориентированное программирование» из книги были взяты основы программирования на С++ и технология объектно-ориентированного программирования.

[8] Р.Лафоре «Объектно-ориентированное программирование в С++», 4-е издание. В книге содержатся основы языка С++ и базовые алгоритмы взаимодействия объектов

[9] С. Макконнел «Совершенный код». Книга описывает лучшие практики в создании приложений.

[10] И другие ...

ПРИЛОЖЕНИЕ А
(Обязательное)

Структурная схема программы

ПРИЛОЖЕНИЕ Б (Обязательное)

Диаграмма классов