Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Пояснительная записка к курсовой работе по дисциплине

«Операционные системы и среды»

Тема: Архиватор, использующий алгоритм арифметического сжатия

Выполнил: Проверил:

Студент гр. 853504 Протько М.И.

Куплевич М. А.

**Минск 2021**

Содержание

Введение........................................................................................................................3

1.Краткие теоретические сведения.............................................................................4

1.1 Алгоритмы сжатия без потерь....................................................................4

1.2 Алгоритм Хаффмана....................................................................................5

1.3 Арифметическое сжатие..............................................................................7

2.Разработка программного средства.......................................................................10

2.1 Выбор инструментов программирования................................................10

2.2 Определение вспомогательных структур данных..................................10

2.3 Реализация алгоритма арифметического сжатия....................................11

3. Демонстрация работы программы........................................................................12

Список использованной литературы........................................................................15

Приложение 1.............................................................................................................16

Приложение 2.............................................................................................................18

**Введение**

Целью курсового проекта является создание простейшего архиватора, использующего эффективный алгоритм сжатия данных. Архиватор – это программа, предназначенная для сжатия или упаковки одного или нескольких файлов в один. В связи с этим выделяют 2 подхода к созданию архиватора.

Первый подход – архиватор, который сжимает данные. Такие архиваторы предназначены для уменьшения размера исходного файла. Выделяют 2 вида алгоритмов сжатия данных: сжатие без потерь и сжатие с потерями.

Суть алгоритма сжатия без потерь в том, что данные перезаписываются в более компактной форме без потери информации. Преимущества такого подхода – это скорость алгоритма и то, что при восстановлении все данные сохраняются. Недостатком является степень сжатия.

При сжатии с потерями часть исходных данных отбрасывается. За счёт этого получается очень высокая степень сжатия, но при восстановлении информации сохраняется не в полном объёме. Поэтому такие алгоритмы применяются там, где потеря информации будет либо незаметна, либо несущественна. Например, они используются для сжатия изображений и видео.

Второй подход – архиватор, который упаковывает несколько файлов в один каталог для удобства перемещения. Как правило, при разработке архиваторов используются оба этих подхода и на выходе получается программа, способная упаковывать множество файлов в один архив и сжать его, для уменьшения занимаемого места. Примерами могут послужить такие программы, как *7-Zip, Info-ZIP, KGB, Archiver, PeaZip, The Unarchiver, Xarchiver.*

В ходе данной работы был реализован архиватор, сжимающий данные без потерь.

**1.Краткие теоретические сведения**

**1.1Алгоритмы сжатия без потерь.**

**Сжатие данных без потерь** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *lossless data compression*) — класс алгоритмов [сжатия данных](https://ru.wikipedia.org/wiki/%D0%A1%D0%B6%D0%B0%D1%82%D0%B8%D0%B5_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85) (видео, аудио, графики, документов, представленных в цифровом виде), при использовании которых закодированные данные однозначно могут быть восстановлены с точностью до [бита](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D1%82), [пикселя](https://ru.wikipedia.org/wiki/%D0%9F%D0%B8%D0%BA%D1%81%D0%B5%D0%BB%D1%8C), [вокселя](https://ru.wikipedia.org/wiki/%D0%92%D0%BE%D0%BA%D1%81%D0%B5%D0%BB) и т.д. При этом оригинальные данные полностью восстанавливаются из сжатого состояния. Этот тип сжатия принципиально отличается от [сжатия данных с потерями](https://ru.wikipedia.org/wiki/%D0%A1%D0%B6%D0%B0%D1%82%D0%B8%D0%B5_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85_%D1%81_%D0%BF%D0%BE%D1%82%D0%B5%D1%80%D1%8F%D0%BC%D0%B8). Для каждого из типов цифровой информации, как правило, существуют свои оптимальные алгоритмы сжатия без потерь.

Сжатие данных без потерь используется во многих приложениях. Например, оно используется во всех [файловых архиваторах](https://ru.wikipedia.org/wiki/%D0%90%D1%80%D1%85%D0%B8%D0%B2%D0%B0%D1%82%D0%BE%D1%80). Оно также используется как компонент в сжатии с потерями.

Сжатие без потерь используется, когда важна идентичность сжатых данных оригиналу. Обычный пример — исполняемые файлы и исходный код. Некоторые графические файловые форматы (например [PNG](https://ru.wikipedia.org/wiki/PNG)) используют только сжатие без потерь, тогда как другие ([TIFF](https://ru.wikipedia.org/wiki/TIFF), [FLIF](https://ru.wikipedia.org/wiki/FLIF) или [GIF](https://ru.wikipedia.org/wiki/GIF)) могут использовать сжатие как с потерями, так и без потерь.

В общих чертах смысл сжатия без потерь таков: в исходных данных находят какую-либо закономерность и с учётом этой закономерности генерируют вторую последовательность, которая полностью описывает исходную.

Большинство алгоритмов сжатия без потерь работают в две стадии: на первой генерируется *статистическая модель* для входящих данных, вторая отображает входящие данные в битовом представлении, используя модель для получения «вероятностных» (то есть часто встречаемых) данных, которые используются чаще, чем «невероятностные».

Среди них выделим 2 наиболее популярных алгоритма: *алгоритм Хаффмана* и *арифметическое кодирование*.

**1.2 Алгоритм Хаффмана**

**Алгоритм Хаффмана** — алгоритм оптимального [префиксного](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B5%D1%84%D0%B8%D0%BA%D1%81%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4) [кодирования](https://ru.wikipedia.org/wiki/%D0%AD%D0%BD%D1%82%D1%80%D0%BE%D0%BF%D0%B8%D0%B9%D0%BD%D0%BE%D0%B5_%D0%BA%D0%BE%D0%B4%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) алфавита с в [1952 году](https://ru.wikipedia.org/wiki/1952_%D0%B3%D0%BE%D0%B4) аспирантом [Массачусетского технологического института](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%81%D1%81%D0%B0%D1%87%D1%83%D1%81%D0%B5%D1%82%D1%81%D0%BA%D0%B8%D0%B9_%D1%82%D0%B5%D1%85%D0%BD%D0%BE%D0%BB%D0%BE%D0%B3%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B8%D0%BD%D1%81%D1%82%D0%B8%D1%82%D1%83%D1%82) [Дэвидом Хаффманом](https://ru.wikipedia.org/wiki/%D0%A5%D0%B0%D1%84%D1%84%D0%BC%D0%B0%D0%BD,_%D0%94%D1%8D%D0%B2%D0%B8%D0%B4) 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.

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

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.

Допустим, у нас есть следующая таблица частот:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Символ** | **А** | **Б** | **В** | **Г** | **Д** |
| **Частота** | 15 | 7 | 6 | 6 | 5 |

Этот процесс можно представить как построение [дерева](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_(%D1%82%D0%B5%D0%BE%D1%80%D0%B8%D1%8F_%D0%B3%D1%80%D0%B0%D1%84%D0%BE%D0%B2)), корень которого — символ с суммой вероятностей объединенных символов, получившийся при объединении символов из последнего шага, его n0 потомков — символы из предыдущего шага и т. д.

Чтобы определить код для каждого из символов, входящих в сообщение, мы должны пройти путь от листа дерева, соответствующего текущему символу, до его корня, накапливая биты при перемещении по ветвям дерева (первая ветвь в пути соответствует младшему биту). Полученная таким образом последовательность битов является кодом данного символа, записанным в обратном порядке.

Для данной таблицы символов коды Хаффмана будут выглядеть следующим образом.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Символ** | **А** | **Б** | **В** | **Г** | **Д** |
| [**Код**](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B5%D1%84%D0%B8%D0%BA%D1%81%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4#%D0%B3%D0%B5%D0%BD%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D1%8F_%D0%B4%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BF%D1%80%D0%B5%D1%84%D0%B8%D0%BA%D1%81%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BA%D0%BE%D0%B4%D0%B0) | 0 | 100 | 101 | 110 | 111 |

Поскольку ни один из полученных кодов не является префиксом другого, они могут быть однозначно декодированы при чтении их из потока. Кроме того, наиболее частый символ сообщения А закодирован наименьшим количеством бит, а наиболее редкий символ Д — наибольшим.

При этом общая длина сообщения, состоящего из приведённых в таблице символов, составит 87 бит (в среднем 2,2308 бита на символ). При использовании равномерного кодирования общая длина сообщения составила бы 117 бит (ровно 3 бита на символ). Заметим, что [энтропия](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%8D%D0%BD%D1%82%D1%80%D0%BE%D0%BF%D0%B8%D1%8F) источника, независимым образом порождающего символы с указанными частотами, составляет ~2,1858 бита на символ, то есть [избыточность](https://ru.wikipedia.org/wiki/%D0%98%D0%B7%D0%B1%D1%8B%D1%82%D0%BE%D1%87%D0%BD%D0%BE%D1%81%D1%82%D1%8C_%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%86%D0%B8%D0%B8) построенного для такого источника кода Хаффмана, понимаемая как отличие среднего числа бит на символ от энтропии, составляет менее 0,05 бит на символ.

Классический алгоритм Хаффмана имеет ряд существенных недостатков. Во-первых, для восстановления содержимого сжатого сообщения декодер должен знать таблицу частот, которой пользовался кодер. Следовательно, длина сжатого сообщения увеличивается на длину таблицы частот, которая должна посылаться впереди данных, что может свести на нет все усилия по сжатию сообщения. Кроме того, необходимость наличия полной частотной статистики перед началом собственно кодирования требует двух проходов по сообщению: одного для построения модели сообщения (таблицы частот и Н-дерева), другого для, собственно, кодирования. Во-вторых, избыточность кодирования обращается в ноль лишь в тех случаях, когда вероятности кодируемых символов являются обратными степенями числа 2. В-третьих, для источника с энтропией, не превышающей 1 (например, для двоичного источника), непосредственное применение кода Хаффмана бессмысленно.

**1.3 Арифметическое сжатие**

**Арифметическое кодирование** (англ. *Arithmetic coding*) — алгоритм сжатия информации без потерь, который при кодировании ставит в соответствие тексту вещественное число из отрезка [0;1). Данный метод, как и [алгоритм Хаффмана](https://neerc.ifmo.ru/wiki/index.php?title=%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%A5%D0%B0%D1%84%D1%84%D0%BC%D0%B0%D0%BD%D0%B0), является [энтропийным](https://neerc.ifmo.ru/wiki/index.php?title=%D0%AD%D0%BD%D1%82%D1%80%D0%BE%D0%BF%D0%B8%D1%8F_%D1%81%D0%BB%D1%83%D1%87%D0%B0%D0%B9%D0%BD%D0%BE%D0%B3%D0%BE_%D0%B8%D1%81%D1%82%D0%BE%D1%87%D0%BD%D0%B8%D0%BA%D0%B0), то есть длина кода конкретного символа зависит от частоты встречаемости этого символа в тексте. Арифметическое кодирование показывает более высокие результаты сжатия, чем алгоритм Хаффмана, для данных с неравномерными распределениями вероятностей кодируемых символов.

При арифметическом кодировании каждый символ кодируется нецелым числом бит, что эффективнее кода Хаффмана (теоретически, символу aa с вероятностью появления p(a) допустимо ставить в соответствие код длины −log2p(a) следовательно, при кодировании алгоритмом Хаффмана это достигается только с вероятностями, равными обратным степеням двойки).

**Кодирование.**

На вход алгоритму передаются текст для кодирования и список частот встречаемости символов.

1. Рассмотрим отрезок [0;1) на координатной прямой.
2. Поставим каждому символу текста в соответствие отрезок, длина которого равна частоте его появления. Формула *High=Lowold+(Highold-Lowold)\*RangeHigh(x)*, где *Lowold*– нижняя граница интервала, *Highold* – верхняя граница интервала *RangeHigh* и *RangeLow* – верхняя и нижняя границы кодируемого символа.
3. Считаем символ из входного потока и рассмотрим отрезок, соответствующий этому символу. Разделим этот отрезок на части, пропорциональные частотам встречаемости символов.
4. Повторим пункт (3)до конца входного потока.
5. Выберем любое число из получившегося отрезка, которое и будет результатом арифметического кодирования.

**Декодирование.**

Алгоритм по вещественному числу восстанавливает исходный текст.

1. Выберем на отрезке [0;1) разделенном на части, длины которых равны вероятностям появления символов в тексте, подотрезок, содержащий входное вещественное число. Символ, соответствующий этому подотрезку, дописываем в ответ.
2. Нормируем подотрезок и вещественное число.
3. Повторим пункты 11—22 до тех пор, пока не получим ответ.

Несмотря на преимущества арифметического кодирования, существует проблема при его практическом применении из-за несовершенства представления чисел с плавающей точкой в памяти компьютера — поскольку некоторые дробные числа не могут быть точно представлены в двоичном коде, используемом современными процессорами (например, 13), границы символов будут округлены, что может повлечь за собой неверную работу алгоритма при больших объёмах данных. В общем случае, алгоритм можно модифицировать так, чтобы результатом было дробное число. В такой реализации вероятность встречи символа представляется в виде рационального числа. Поскольку в каждой итерации будет переход из текущего отрезка в один из его m подотрезков, кратных по длине n, а всего итераций n, в конечном результате знаменатель дроби не превысит nn, а поскольку сумма всех вероятностей встречи символов равна 1, полученная дробь будет находиться в промежутке [0;1).

Пример.

Рассмотрим в качестве примера строку abacaba:

**Кодирование**

|  |  |
| --- | --- |
| **Символ** | **Частота появления** |
| a | 0.571429 |
| b | 0.285714 |
| c | 0.142857 |

|  |  |  |
| --- | --- | --- |
| **Считанный символ** | **Левая граница отрезка** | **Правая граница отрезка** |
|  | 0 | 1 |
| a | 0 | 0.571429 |
| b | 0.326531 | 0.489796 |
| a | 0.326531 | 0.419825 |
| c | 0.406497 | 0.419825 |
| a | 0.406497 | 0.414113 |
| b | 0.410849 | 0.413025 |
| a | 0.410849 | 0.412093 |

Код: 0.411471

### Декодирование

Код: 0.411471

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Декодируемый символ** | **Границы отрезка** | | | |
| a | 0 | 0.571429 | 0.857143 | 1 |
| b | 0 | 0.326531 | 0.489796 | 0.571429 |
| a | 0.326531 | 0.419825 | 0.466472 | 0.489796 |
| c | 0.326531 | 0.379842 | 0.406497 | 0.419825 |
| a | 0.406497 | 0.414113 | 0.417921 | 0.419825 |
| b | 0.406497 | 0.410849 | 0.413025 | 0.414113 |
| a | 0.410849 | 0.412093 | 0.412714 | 0.413025 |

Полученное слово: abacaba.

**2.Разработка программного средства**

**2.1 Выбор инструментов программирования.**

Исходя из анализа предметной области, для разработки архиватора были выбраны следующие инструменты.

1. Язык программирования C++.
2. Операционная система Windows 10 Pro.
3. Интегрированная среда разработки Embarcadero C++ Builder 10.3.

**2.2 Определение вспомогательных структур данных.**

Для хранения таблиц частот символов удобно применять такую динамическую структуру данных, как словарь или ассоциативный массив. В стандартной библиотеке stl есть структура данных map, аналогичная той, что нам нужна. Но использование map довольно неудобно для решаемой задачи, поэтому было принято решение реализовать свой «словарь» с минимальным набором методов, помогающий облегчить выполнение поставленной задачи.

Данная структура размещена в файле Dict.h. Исходный код представн=лен в приложении 1.

Среди основных методов структуры:

1) void Add(wchar\_t key);

2) bool HasKey(wchar\_t key);

3) void sort();

4) int FindPos(wchar\_t key);

Первый метод предназначен для добавления количества символов по ключу – значению самого символа.

Второй – вспомогательный метод. Позволяет определить содержится ли такой ключ в словаре. Если ключ есть, возвращает true, иначе – false.

Третий – сортировка словаря по количеству символов в обратном порядке. Необходима для корректного составления отрезка.

Четвёртый – возвращает номер символа в словаре. Нужен для связи символов в отрезе и таблице.

Также для удобства декодирования был переопределён опереатор []. По номеру элемента в словаре возвращается его ключ.

**2.3 Реализация Алгоритма арифметического сжатия.**

Сам алгоритм представлен в п. 1.3.

Реализация алгоритма в процедурном стиле. Исходный код представлен в приложении 2.

Среди основных функций:

1) Dict symbols(wchar\_t\* text);

2) float encoding(wchar\_t\* msg);

3) wchar\_t\* decode(float code, Dict alph);

Первая возвращает таблицу частот символов в тексте сообщения.

Вторая выполняет кодирование сообщения по алгоритму арифметического сжатия. Результат – число типа float – код сообщения.

Третья выполняет декодирование. Результат строка сообщения.

Также были определены и реализованы некоторые вспомогательные функции среди которых расчёт частоты встречаемости символов в строке, инициализация отрезка, нормализация отрезка.

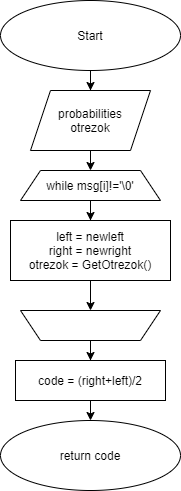
Схема алгоритма кодирования представлена на рис.1.

рис. 1 Схема алгоритма кодирования

**3**.**Демонстрация работы программы.**

На рис. 2 представлен внешний вид главного окна приложения.

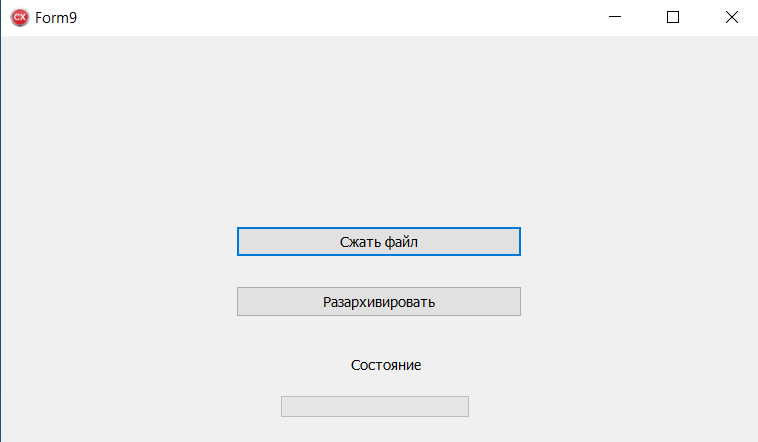


рис.2 Главный экран

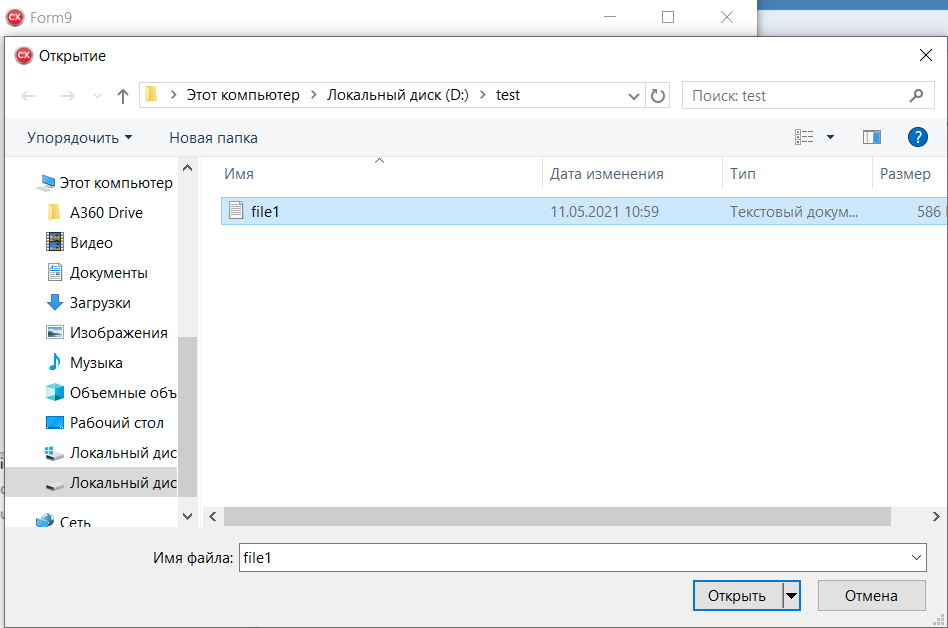
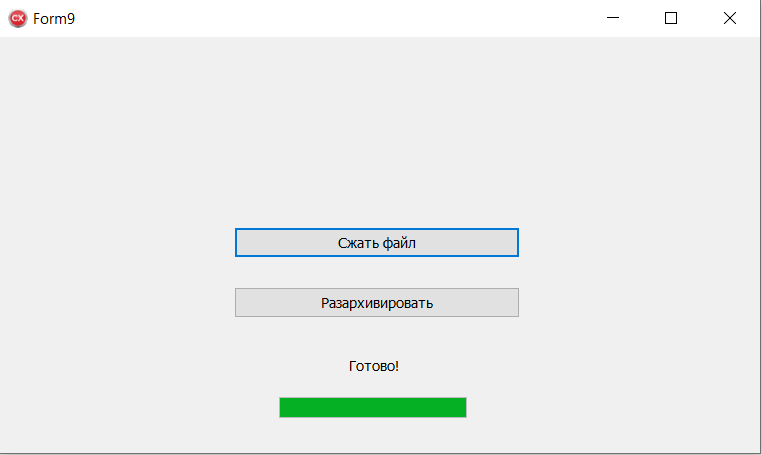


рис. 3 Открытие файла для сжатия

 *рис.4 Программа после выполнения сжатия*

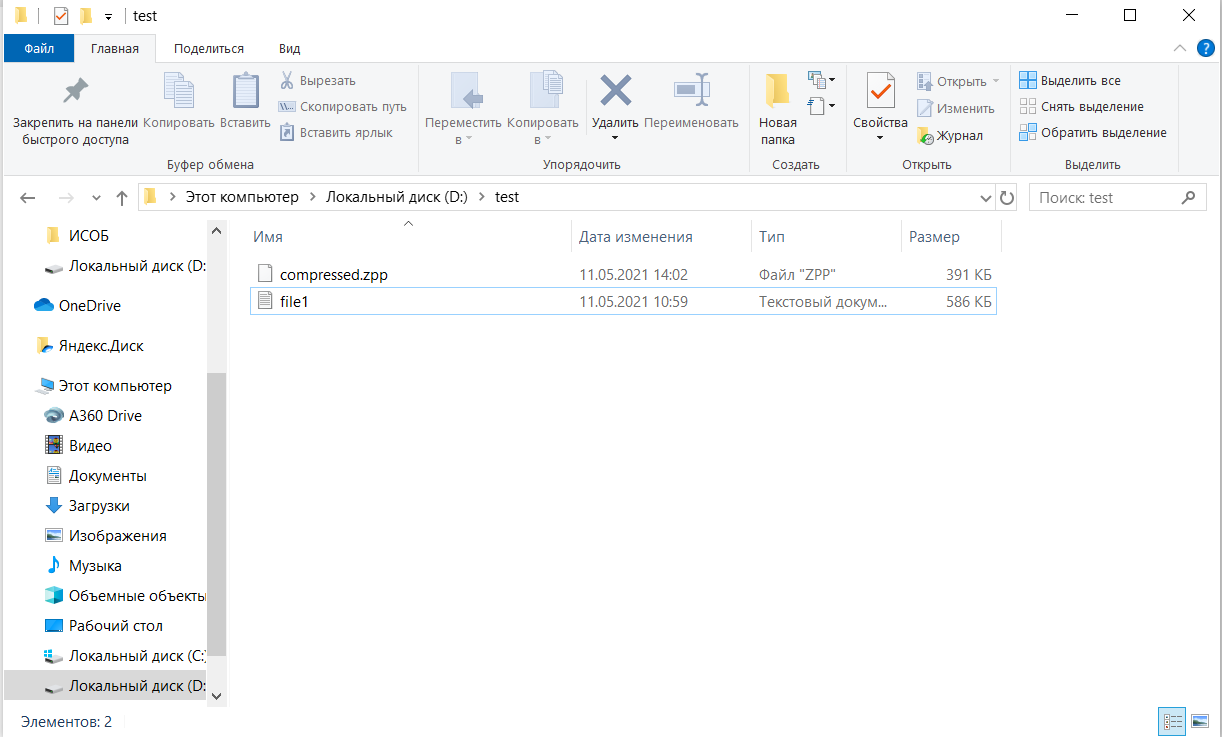


рис.5 Папка test после сжатия

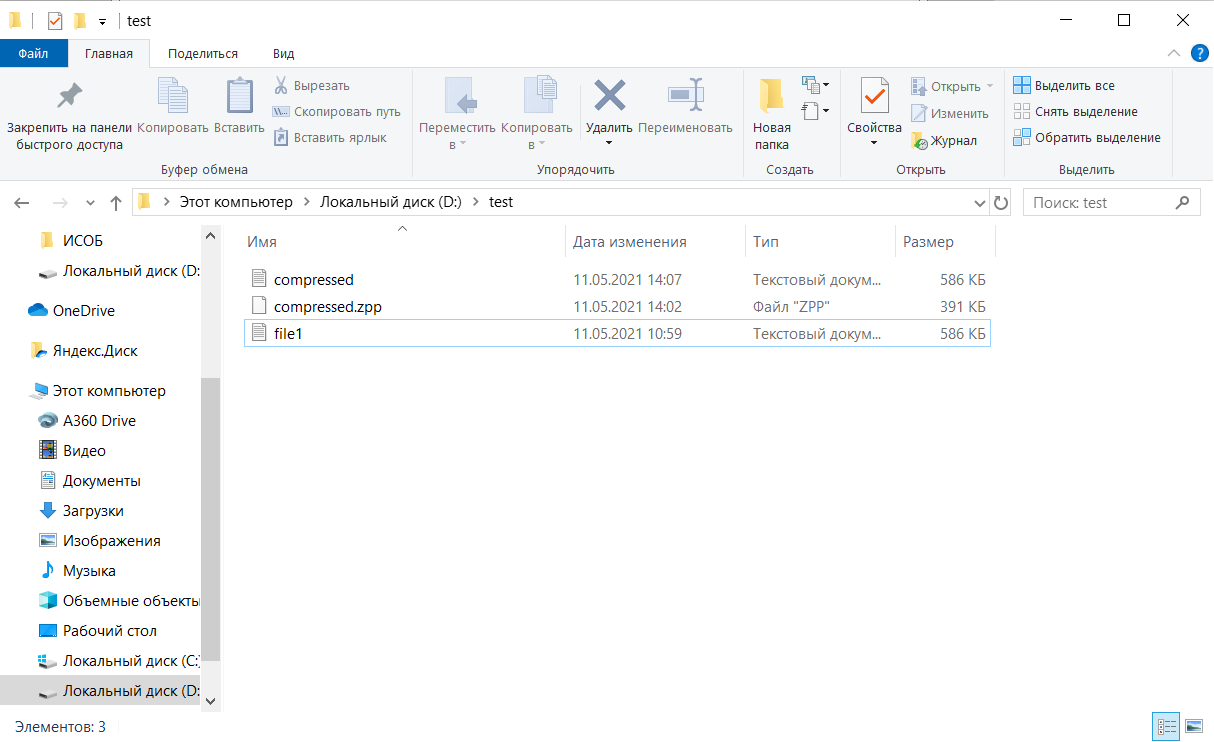


рис.6 Папка test после разархивации compressed.zpp

**Список использованной литературы**

1. <https://ru.wikipedia.org/wiki/Арифметическое_кодирование#Декодирование_сообщения>.
2. <https://habr.com/ru/post/130531/>
3. <http://www.msiit.ru/x/ti/_7______.html> Лекция 7 Адаптивный алгоритм Хаффмена. Арифметическое сжатие.

**Приложение 1.** Исходный код структуры данных Dict.

Dict::Dict()

{

}

bool Dict::HasKey(wchar\_t key)

{

for(int i = 0; i < keys.size(); i++)

{

if(key == keys[i])

return true;

}

return false;

}

void Dict::Add(wchar\_t key, int val)

{

if(!HasKey(key))

{

Node temp;

temp.key = key;

temp.value = val;

data.push\_back(temp);

keys.push\_back(key);

values.push\_back(val);

size++;

}

}

void Dict::Add(wchar\_t key)

{

if(!HasKey(key))

{

Node temp = {key, 1};

data.push\_back(temp);

keys.push\_back(key);

values.push\_back(1);

size++;

}

else

{

for(int i = 0; i < data.size(); i++)

{

if(data[i].key == key)

{

data[i].value++;

values[i]++;

break;

}

}

}

}

int Dict::FindPos(wchar\_t key)

{

for(int i = 0; i < this->size; i++)

{

if(data[i].key == key)

return i;

}

return -1;

}

void Dict::sort()

{

for (int i = 0; i < size - 1; i++) {

for (int j = 0; j < size - i - 1; j++) {

if (data[j].value < data[j + 1].value) {

Node temp = data[j];

data[j] = data[j + 1];

data[j + 1] = temp;

}

}

}

}

**Приложение 2.** Исходный код функций кодирования и декодирования.

Dict symbols(wchar\_t\* text)

{

Dict alph;

for(int i = 0; i < wcslen(text); i++)

{

alph.Add(text[i]);

}

alph.sort();

return alph;

}

float\* GetProbabilities(Dict alph)

{

int count = 0;

for(int i = 0; i < alph.size; i++)

{

count += alph.data[i].value;

}

float\* probabilities = new float[alph.size];

for(int i = 0; i < alph.size; i++)

{

probabilities[i] = (float)alph.data[i].value/count;

}

return probabilities;

}

float\* init\_otrezok(float\* prob, int n)

{

float\* otr = new float[n];

for(int i = 0; i < n; i++)

{

if(i == 0)

{

otr[i] = prob[i];

}

else

if(i != n-1)

{

otr[i] = otr[i-1] + prob[i];

}

else

{

otr[i] = 1;

}

}

return otr;

}

float\* Get\_otrezok(float left, float right, float\* prob, int n)

{

float \*otr = new float[n];

for(int i = 0; i < n; i++)

{

otr[i] = left + (right-left)\*prob[i];

}

return otr;

}

float encoding(wchar\_t\* msg)

{

String s = String(msg) + "`";

msg = s.w\_str();

Dict alph = symbols(msg);

float right = 1, left = 0;

float\* probabilities = GetProbabilities(alph);

float\* otrezok = init\_otrezok(probabilities, alph.size);

int i = 0;

while(msg[i]!='\0')

{

int idx = alph.FindPos(msg[i]);

if(idx != 0)

left = otrezok[idx-1];

right = otrezok[idx];

otrezok = Get\_otrezok(left, right, otrezok, alph.size);

i++;

}

left = otrezok[0];

right = otrezok[alph.size-1];

return right;

}

float normalize\_code(float &code, float left, float right)

{

code = (float)(code - left)/(right-left);

return code;

}

wchar\_t\* decode(float code, Dict alph)

{

float\* probabilities = GetProbabilities(alph);

float\* otrezok = init\_otrezok(probabilities, alph.size);

String s = "";

wchar\_t c = L'k';

float left = 0, right = 1;

while(c != L'`')

{

int j = 0;

for(; code > otrezok[j]; j++)

{

}

if(j != 0)

{

left = otrezok[j-1];

}

right = otrezok[j];

c = alph[j];

if(c!=L'`')

s += c;

else

s +=L'\0';

otrezok = Get\_otrezok(left, right, otrezok, alph.size);

}

return s.w\_str();

}