МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ   
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ   
«ВЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Институт математики и информационных систем

ФАКУЛЬТЕТ КОМПЬЮТЕРНЫХ И ФИЗИКО-МАТЕМАТИЧЕСКИХ НАУК

КАФЕДРА ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ

Допущен к защите

И.о. заведующего кафедрой ПМИ

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Е.В. Разова

**алгоритмы сжатия данных без потерь**

Курсовой проект по дисциплине  
«Проектная и научно-исследовательская деятельность»

Выполнил студент группы ФИб-2302-51-00   / /

Руководитель старший преподаватель кафедры ПМИ   / А.В. Торбеева /

Работа защищена с оценкой     \_\_\_\_.\_\_\_\_.2020 г.

Члены комиссии:     /     /

    /     /

Киров 2020

Содержание

[Введение 3](#_Toc41838179)

[1 Анализ предметной области 4](#_Toc41838180)

[1.1 Основные понятия и определения 4](#_Toc41838181)

[1.2 Постановка задачи 5](#_Toc41838182)

[1.3 Выводы по разделу 5](#_Toc41838183)

[2 Обзор современных алгоритмов сжатия данных без потерь 5](#_Toc41838184)

[2.1 Алгоритмы сжатия данных без потерь 5](#_Toc41838185)

[2.2 Подробное рассмотрение нескольких алгоритмов. 8](#_Toc41838186)

[2.3 Выводы по разделу 14](#_Toc41838187)

[3 Программная реализация алгоритмов сжатия данных без потерь 15](#_Toc41838188)

[3.1 Программная реализация алгоритма LZ77. 15](#_Toc41838189)

[3.2 Программная реализация алгоритма LZW. 19](#_Toc41838190)

[4 Экспериментальное исследование реализованных алгоритмов на различных наборах данных 24](#_Toc41838191)

[4.1 Тестирование программы LZW 24](#_Toc41838192)

[4.2 Тестирование программы LZ77 25](#_Toc41838193)

[4.3 Выводы по разделу 25](#_Toc41838194)

[Заключение 26](#_Toc41838195)

[Библиографический список 27](#_Toc41838196)

[Приложения 28](#_Toc41838197)

[Приложение А. Листинг программы LZ77 28](#_Toc41838198)

[Приложение Б. Листинг программы LZW 32](#_Toc41838199)

# Введение

В данный момент в мире находится множество информации: текст, изображения, видеофайлы и т. д. и чтобы поделиться ей, зачастую приходиться отправлять ее либо в интернет, либо копировать на съемный носитель. А если файл большой, то время отправки или копирования, а затем и скачивания займет довольно большое время. Чтобы исправить эту проблему нужно сжимать файл.

Целью курсового проекта является нахождение оптимального алгоритма сжатия данных без потерь.

Для достижения цели необходимо решить следующие задачи.

1. Ознакомиться с существующими алгоритмами.
2. Исследование алгоритмов.
3. Программная реализация алгоритмов.
4. Вывод.

Курсовой проект включает четыре раздела. В первом разделе знакомство с предметной областью. Во втором разделе обзор современных алгоритмов. В третьем разделе программная реализация алгоритмов сжатия. В четвертом разделе сравнивание алгоритмов.

# Анализ предметной области

## Основные понятия и определения

*Сжатие данных без потерь* (англ. lossless data compression) — метод сжатия, при котором выход декодера всегда тождественно совпадает с исходными данными, поступившими на вход кодера [[1, c. 354]](#_Библиографический_список).

Сжатие без потерь используется, когда важна идентичность сжатых данных оригиналу. Обычный пример – исполняемые файлы и исходный код. Некоторые графические файловые форматы (например PNG) используют только сжатие без потерь, тогда как другие (TIFF, FLIF или GIF) могут использовать сжатие как с потерями, так и без потерь.

Под энтропией символа , имеющего вероятность , подразумевается количество информации, содержащейся в а, которая равна — . Например, если вероятность символа равна 0.5, то его энтропия — = 0.5. Если символы некоторого алфавита с символами от до имеют вероятности от до , то энтропия всего алфавита равна сумме . Если задана строка символов этого алфавита, то для нее энтропия определяется аналогично. С помощью понятия энтропии теория информации показывает, как вычислять вероятности строк символов алфавита, и предсказывает ее наилучшее сжатие, то есть, наименьшее, в среднем, число бит, необходимое для представления этой строки символов [[1, c. 25].](#_Библиографический_список)

Коэффициент сжатия определяется по формуле: [[1, c. 21]](#_Библиографический_список).

Коэффициент 0.6 означает, что сжатые данные занимают 60% от исходного размера. Значения большие 1 говорят о том, что выходной файл больше входного (отрицательное сжатие). Коэффициент сжатия принято измерять в bpb (bit per bit, бит на бит), так как он показывает, сколько в среднем понадобится бит сжатого файла для представления одного бита файла на входе. При сжатии графических изображений аналогично определяется величина bpp (bit per pixel, бит на пиксел). В современных эффективных алгоритмах сжатия текстовой информации имеет смысл говорить о похожей величине bpc (bit per character, бит на символ), то есть, сколько в среднем потребуется бит для хранения одной буквы текста.

Величина, обратная коэффициенту сжатия, называется фактором сжатия:

[[1, c. 22].](#_Библиографический_список)

В этом случае значения большие 1 означают сжатие, а меньшие 1 - расширение. Этот множитель представляется большинству людей более естественным показателем: чем больше фактор, тем лучше компрессия.

## Постановка задачи

Назначение приложения заключается в следующем: тестирование выбранных алгоритмов сжатия данных без потерь на различных наборах данных.

## Выводы по разделу

Таким образом, ознакомились с предметной областью, а также разобрались в определениях и основных понятиях.

# Обзор современных алгоритмов сжатия данных без потерь

## Алгоритмы сжатия данных без потерь

1. Алгоритмы, применяющие метод «скользящего окна».

Началу создания алгоритмов сжатия данных послужил алгоритм LZ77 (1977 год), который представил новую концепцию «скользящего окна», позволившую значительно улучшить сжатие данных. LZ77 использует словарь, содержащий тройки данных – смещение, длина серии и символ расхождения. Смещение – как далеко от начала файла находится фраза. Длина серии – сколько символов, считая от смещения, принадлежат фразе. Символ расхождения показывает, что найдена новая фраза, похожая на ту, что обозначена смещением и длиной, за исключением этого символа. Словарь меняется по мере парсинга файла при помощи скользящего окна.

Примеры:

LZR — Модификация алгоритма LZ77, предложенная Майклом Роуде в 1981 году. В отличие от LZ77 работает за линейное время, однако требует большего объёма памяти. Обычно проигрывает LZ78 в сжатии.

LZSS — Алгоритм Лемпеля-Зива-Сторера-Цимански был представлен в 1982 году. Улучшенная версия LZ77, которая просчитывает, не увеличит ли размер результата замена исходных данных кодированными. До сих пор используется в популярных архиваторах, например RAR. Иногда – для сжатия данных при передаче по сети.

1. Алгоритмы с использованием словаря.

Разбиение данных на слова и замена их на индексы в словаре. В настоящее время это наиболее распространенный подход для сжатия данных, он является естественным обобщением [RLE](https://ru.wikipedia.org/wiki/RLE).

В наиболее распространенном варианте реализации словарь постепенно пополняется словами из исходного блока данных в процессе сжатия.

Основной параметр любого словарного метода — это размер словаря. Чем больше словарь, тем выше эффективность. Однако для неоднородных данных чрезмерно большой размер может быть вреден, так как при резком изменении типа данных словарь будет заполнен неактуальными словами. Для эффективной работы этих методов при сжатии требуется дополнительная память — приблизительно на порядок больше, чем нужно для исходных данных словаря. Существенное преимущество словарных методов — простая и быстрая процедура распаковки. Дополнительная память при этом не требуется. Такая особенность крайне важна, если необходим оперативный доступ к данным.

Примеры:

LZ78 — Алгоритм 1978 года, авторы – Лемпель и Зив. Вместо использования скользящего окна для создания словаря, словарь составляется при парсинге данных из файла. Объём словаря обычно измеряется в нескольких мегабайтах. Отличия в вариантах этого алгоритма строятся на том, что делать, когда словарь заполнен.

LZW — Лемпель-Зив-Велч, 1984 год. Самый популярный вариант LZ78, несмотря на запатентованность. Алгоритм избавляется от лишних символов на выходе и данные состоят только из указателей. Также он сохраняет все символы словаря перед сжатием и использует другие трюки, позволяющие улучшать сжатие – к примеру, кодирование последнего символа предыдущей фразы в качестве первого символа следующей. Используется в GIF, ранних версиях ZIP и других специальных приложениях. Очень быстр, но проигрывает в сжатии более новым алгоритмам.

1. Алгоритмы, не использующие словарь.

PPM — Предсказание по частичному совпадению – использует уже обработанные данные, чтобы предсказать, какой символ будет в последовательности следующим, таким образом уменьшая энтропию выходных данных. Обычно комбинируется с арифметическим кодировщиком или адаптивным кодированием Хаффмана. Вариация PPMd используется в RAR и 7-zip.

bzip2 — Реализация BWT с открытым исходным кодом. При простоте реализации достигает хорошего компромисса между скоростью и степенью сжатия, в связи с чем популярен в UNIX. Сначала данные обрабатываются при помощи RLE, затем BWT, потом данные особым образом сортируются, чтобы получить длинные последовательности одинаковых символов, после чего к ним снова применяется RLE. И, наконец, кодировщик Хаффмана завершает процесс.

## Подробное рассмотрение нескольких алгоритмов.

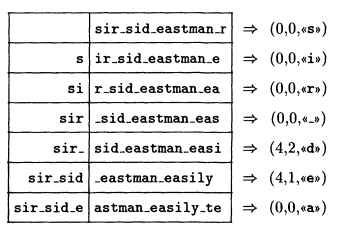
Алгоритм LZ77.

Основная идея этого метода (его еще часто называют методом LZ1, см. [Ziv 77]) состоит в использовании ранее прочитанной части входного файла в качестве словаря. Кодер создает окно для входного файла и двигает его справа налево в виде строки символов, требующих сжатие. Таким образом, метод основан на скользящем окне. Окно разбивается на две части. Часть слева называется буфером поиска. Она будет служить текущим словарем, и в ней всегда содержатся символы, которые недавно поступили и были закодированы. Правая часть окна называется упреждающим буфером, содержащим текст, который будет сейчас закодирован. На практике буфер поиска состоит из нескольких тысяч байт, а длина упреждающего буфера равна нескольким десяткам символов. Вертикальная черта | между символами t и е означает текущее разделение между двумя буферами. Итак, предположим, что текст «sir\_sid\_eastman\_easily\_t» уже сжат, а текст «eases\_sea\_sick\_seals» нуждается в сжатии.

←выход...…←вход Кодер просматривает буфер поиска в обратном порядке (справа налево) и ищет в нем первое появление символа е из упреждающего буфера. Он обнаруживает такой символ в начале слова easily. Значит, е находится (по смещению) на расстоянии 8 от конца буфера поиска. Далее кодер определяет, сколько совпадающих символов следует за этими двумя символами е. В данном случае совпадение по символам eas имеет длину 3. Кодер продолжает поиск, пытаясь найти более длинные совпадающие последовательности. В нашем случае имеется еще одна совпадающая последовательность той же длины 3 в слове eastman со смещением 16. Декодер выбирает самую длинную из них, а при совпадении длин - самую удаленную, и готовит метку (16, 3, «е»). Выбор более удаленной ссылки упрощает кодер. Интересно отметить, что выбор ближайшего совпадения, несмотря на некоторое усложнение программы, также имеет определенные преимущества. При этом выбирается наименьшее смещение. Может показаться, что в этом нет преимущества, поскольку в метке будет предусмотрено достаточно разрядов для хранения самого большого смещения. Однако, можно следовать LZ77 с кодированием Хаффмана или другого статистического кодирования меток, при котором более коротким смещениям присваиваются более короткие коды. Этот метод, предложенный Бернардом Хер дом (Bernard Herd), называется LZH. Если получается много коротких смещений, то при LZH достигается большее сжатие. Следующую идею очень легко усвоить. Декодер не знает, какое именно совпадение выбрал кодер, ближайшее или самое дальнее, но ему это и не надо знать! Декодер просто читает метки и использует смещение для нахождения текстовой строки в буфере поиска. Для него не важно, первое это совпадение или последнее. В общем случае, метка из LZ77 имеет три поля: смещение, длина и следующий символ в упреждающем буфере (в нашем случае, это будет второе е в слове teases. Эта метка записывается в выходной файл, а окно сдвигается вправо (или входной файл сдвигается влево) на четыре позиции: три позиции для совпадающей последовательности и одна позиция для следующего символа.

…

Если обратный поиск не обнаружил совпадение символов, то записывается метка со смещением О и длиной 0. По этой же причине метка будет иметь третью компоненту. Метки с нулевыми смещением и длиной всегда стоят в начале работы, когда буфер поиска пуст или почти пуст. Первые семь шагов кодирования нашего примера приведены ниже.



Ясно, что метки вида (0,0,...), которые кодируют единичные символы, не обеспечивают хорошее сжатие. Легко оценить их длину. Размер смещения равен [] , где S - длина буфера поиска. На практике этот буфер имеет длину в несколько сотен байт, поэтому смещение имеет длину 10 — 12 бит. Поле «длина» имеет размер равный [] , где L - длина упреждающего буфера (в следующем абзаце будет объяснено почему надо вычитать 1). Обычно упреждающий буфер имеет длину нескольких десятков байтов. Поэтому длина этого поля равна нескольким битам. Размер поля «символ», обычно, равен 8 бит, но в общем случае - [log2 А] , где Л ~ размер алфавита. Полный размер метки (0,0,...) одиночного символа равен 11-1-5-1-8=24 бит, что много больше длины 8 «сырого» символа без нулевых элементов кода. Следующий пример показывает, почему поле «длина» может быть длиннее размера упреждающего буфера:



Первый символ в упреждающем буфере совпадает с 5-ым а буфера поиска. Может показаться, что оба крайних подходят с длиной совпадения 3, поэтому кодер выберет самый левый символ и создаст метку (28,3,«а»). На самом деле кодер образует метку (3,4,«\_»). Строка из четырех символов «alf а» из упреждающего буфера совпадает с тремя символами «alf» буфера поиска и первым символом «а» упреждающего буфера. Причина этого заключается в том, что декодер может обращаться с такой меткой очень естественно безо всяких модификаций. Он начинает с позиции 3 буфера поиска и копирует следующие 4 символа, один за другим, расширяя буфер вправо. Первые 3 символа копируются из старого содержимого буфера, а четвертый является копией первого из этих новых трех. Следующий пример еще более убедителен (хотя он немного надуман):



Кодер создает метку (1,9,«А») для совпадения первых девяти копий символа «А» в упреждающем буфере, включая десятый символ А. Вот почему длина совпадения, в принципе, может быть равна размеру упреждающего буфера минус 1. Декодер метода LZ77 гораздо проще кодера (то есть, метод LZ77 является асимметричным методом сжатия). Он должен соорудить такой же буфер поиска, как и кодер. Декодер вводит метку, находит совпадение в своем буфере, записывает совпадающие символы и символ из третьего поля в буфер. Этот метод или его модификация хорошо работает, если файл сжимается один раз (или несколько раз), но часто разжимается. Часто используемый архив старых сжатых файлов может служить хорошим примером использования этого алгоритма [[1, c.84 – 86]](#_Библиографический_список).

Алгоритм LZW.

Это весьма популярный вариант алгоритма LZ78, который был разработан Терри Уэлчем (Terry Welch) в 1984 ([Welch 84] и [Phillips 92]). Его главной особенностью является удаление второго поля из метки. Метка LZW состоит только из указателя на место в словаре. Для лучшего понимания алгоритма LZW мы временно забудем, что словарь является деревом и будем предполагать, что словарь — это просто массив, состоящий из строк разной длины. Метод LZW начинается инициализацией словаря всеми символами исходного алфавита. В общем случае 8-битного алфавита, первые 256 записей (отдельные символы с номерами от О до 255) заносятся в словарь до поступления сжимаемых данных. Поскольку словарь уже частично заполнен, первые поступившие символы всегда будут обнаружены в словаре, поэтому метка может состоять всего лишь из указателя, и нет надобности дополнительно хранить код символа, как в алгоритмах LZ77 и LZ78. (Алгоритм LZW запатентован и для его использования необходима лицензия. Издание патентов для программного обеспечения обсуждается в [Salomon 00].) Метод LZW накапливает поступающие на вход символы в строке I. После каждого нового символа, добавленного в строку I, кодер ищет I в словаре. Если строка обнаружена, то процесс удлинения I продолжается. В некоторый момент добавление нового символа х приводит к не обнаружению строки 1х (символ х был добавлен к I). Тогда кодер (1) записывает в выходной файл указатель в словаре на строку I, (2) сохраняет строку 1х (которая теперь будет называться фразой) в словаре на следующей допустимой позиции и (3) инициализирует (присваивает) строке I новое значение х.

Для того, чтобы понять как работает декодер метода LZW, прежде всего еще раз напомним основные три шага, которые выполняет кодер каждый раз, делая очередную запись в выходной файл: (1) он заносит туда словарный указатель на строку I, (2) сохраняет строку Iх в следующей свободной позиции словаря и (3) инициализирует строку I символом X. Декодер начинает с заполнения словаря первыми символами алфавита (их, обычно, 256). Затем он читает входной файл, который состоит из указателей в словаре, использует каждый указатель для того, чтобы восстановить несжатые символы из словаря и записать их в выходной файл. Кроме того, он строит словарь тем же методом, что и кодер (этот факт, обычно, отражается фразой: кодер и декодер работают синхронно или шаг в шаг). На первом шаге декодирования декодер вводит первый указатель и использует его для восстановления словарного элемента I. Это строка символов, и она записывается декодером в выходной файл. Далее следует записать в словарь строку 1х, однако символ х еще неизвестен; это будет первый символ следующей строки, извлеченной из словаря. На каждом шаге декодирования после первого декодер вводит следующий указатель, извлекает следующую строку J из словаря, записывает ее в выходной файл, извлекает ее первый символ х и заносит строку Iх в словарь на свободную позицию (предварительно проверив, что строки 1х нет в словаре). Затем декодер перемещает J в I. Теперь он готов к следующему шагу декодирования. В нашем примере «sir.sid . . .» первым символом входного файла декодера является указатель 115. Он соответствует строке s, которая извлекается из словаря, сохраняется в I и становится первой строкой, записанной в выходной (разжатый) файл. Следующий указатель - 105, поэтому в J заносится строка i, а содержимое J записывается в выходной файл. Первый символ строки J добавляется к переменной , образуя строку , которой нет в словаре, поэтому она добавляется в словарь на позиции 256. Содержимое переменной J переносится в переменную I; теперь I равно i. Следующий указатель - 114, поэтому, из словаря извлекается строка г, заносится в J и пишется в выходной файл. Переменная дополняется до значения ; такой строки нет в словаре, поэтому она туда заносится под номером 257. Переменная J переписывается в I, теперь I равно . На следующем шаге декодер читает указатель 32, записывает «\_» в файл и сохраняет в словаре строку [[1, c. 97, 101 – 102]](#_Библиографический_список).

Особенность LZW заключается в том, что для декомпрессии нам не надо сохранять таблицу строк в файл для распаковки. Алгоритм построен таким образом, что мы в состоянии восстановить таблицу строк, пользуясь только потоком кодов [[2, c. 294]](#_Библиографический_список).

Достоинства:

1) Не требует вычисления вероятностей встречаемости символов или кодов.

2) Для декомпрессии не надо сохранять таблицу строк в файл для распаковки. Алгоритм построен таким образом, что мы в состоянии восстановить таблицу строк, пользуясь только потоком кодов.

3) Данный тип компрессии не вносит искажений в исходный графический файл, и подходит для сжатия растровых данных любого типа.

Недостаток: Алгоритм не проводит анализ входных данных, поэтому не оптимален.

## Выводы по разделу

Разобрались в алгоритмах сжатия данных без потерь, в их работе, достоинствах. А также рассмотрели подробно некоторые алгоритмы сжатия данных.

# Программная реализация алгоритмов сжатия данных без потерь

## Программная реализация алгоритма LZ77.

Программную реализацию алгоритма было решено писать на языке C++, в обычном консольном приложении.

Программа состоит из класса archive, функций класса, а также функций: компрессии, декомпрессии, поиска совпадения в словаре, добавления символа в словарь.

Класс archive, и функции, и процедуры класса отвечают за ввод/вывод файла в алгоритм:

class archiving

{

int handle;

unsigned char buffer, mask;

public:

void assign\_read(int h); // связывание для чтения

void assign\_write(int h); // связывание для записи

void write\_zero(); // заполнение нулями

void petit(bool val); // запись бита

bool getbit(); // чтение бита

void putbits(int val, int n); // запись n битов

int getbits(int n); // чтение n битов

};

Функции и процедуры класса:

void archiving::write\_zero()

{

for (int i = 0; i < 7; i++) putbit(0);

}

// связывание для записи

void archiving::assign\_write(int h)

{

handle = h;

buffer = 0;

mask = 128;

}

// связывание для чтения

void archiving::assign\_read(int h)

{

handle = h;

buffer = 0;

mask = 0;

}

//чтение бита

bool archiving::getbit()

{

if ((mask >>= 1) == 0)

{

\_read(handle, &buffer, 1);

mask = 128; //двоичное 10000000

}

return (buffer & mask) != 0;

}

// запись бита

void archiving::putbit(bool val)

{

if (val) buffer |= mask;

if ((mask >>= 1) == 0)

{

\_write(handle, &buffer, 1);

buffer = 0;

mask = 128; //двоичное 10000000

}

}

// запись n битов

void archiving::putbits(int val, int n)

{

int m = 1;

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

{

putbit((val & m) != 0);

m <<= 1;

}

}

// чтение n битов

int archiving::getbits(int n)

{

int result = 0;

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

result |= getbit() << i;

return result;

}

Остальные функции программы:

// Возвращает позицию sub от КОНЦА src. от 1 до len

// Если не найдено, возвращает 0

int strpos(char \*src, int len, char \*sub, int sub\_len)

{

for (int i = 0; i <= len - sub\_len; i++)

if (memcmp(src + i, sub, sub\_len) == 0) return len - i;

return 0;

}

Процедура поиска совпадения в словаре:

void find\_match()

{

match\_len = 0; //длина совпадения

match\_pos = 1; //позиция совпадения

while(match\_len < BUF\_LEN) //пока размер совпадения меньше размера буфера

{

\_read(in\_file, &buffer[match\_len], 1); //чтение файла

if (\_eof(in\_file)) break; //если файл пустой – выход

int pos1 = strpos(dict, dict\_pos, buffer, match\_len + 1);

if (pos1 == 0) break; //если позиция равна 0 - выход

match\_pos = pos1; //указывание позиции совпадения

match\_len++; //увеличение длины совпадения

}

unmatched = buffer[match\_len];

}

Процедура добавления символа в словарь:

void add\_dict(char c)

{

if (dict\_pos == (DICT\_LEN - 1)) //если позиция словаря равна размеру словаря-1

{

memcpy(dict, dict + 1, DICT\_LEN - 1); //копирование DICT\_LEN-1 байтов из блока dict+1 в блок dict

dict[dict\_pos - 1] = c; //добавление символа в словарь на позицию dict\_pos-1

}

else //иначе

{

dict[dict\_pos] = c; //добавление символа в словарь на позицию dict\_pos

dict\_pos++; //увеличение позиции

}

}

Процедура кодирования:

void encode()

{

while(!\_eof(in\_file)) //пока файл не пуст

{

find\_match(); //ищем совпадение

archive.putbits(match\_pos, OFFS\_LN); //запись в сжатый файл

archive.putbits(match\_len, LEN\_LN); //запись в сжатый файл

if (match\_len < BUF\_LEN) //если размер совпадение меньше размера буфера

archive.putbits(unmatched, BYTE\_LN); //запись в сжатый файл

for (int i = 0; i < match\_len; i++)

add\_dict(buffer[i]); //добавление в словарь

if (match\_len < BUF\_LEN) //если размер совпадение меньше размера буфера

add\_dict(unmatched); //добавление в словарь

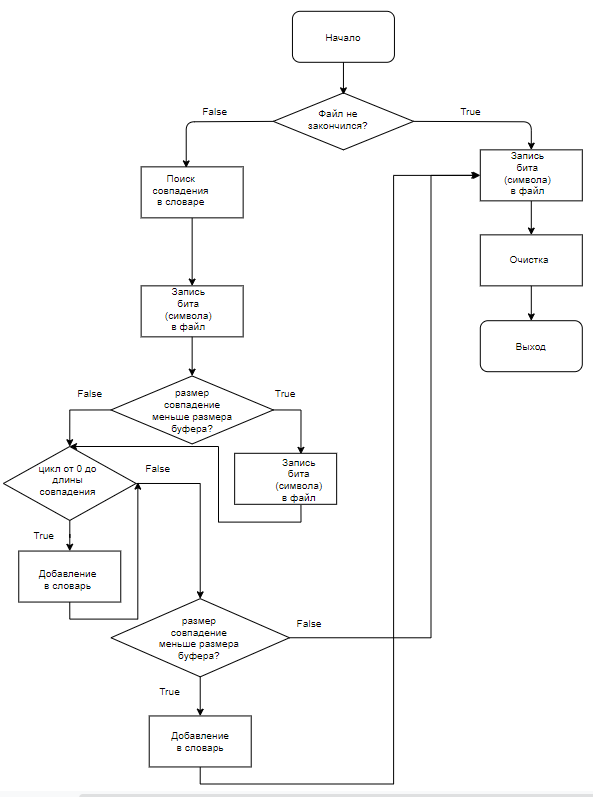
}

archive.putbits(0, BYTE\_LN); //запись нулей

archive.write\_zero(); //заполнение нулями

}

Блок схема процедуры кодирования:



Процедура декодирования:

void decode()

{

char c;

int i;

for(;;) //пустой цикл для того, чтобы потом можно было выйти

{

match\_pos = archive.getbits(OFFS\_LN); //позиция совпадения

if (match\_pos == 0) break; //если позиция совпадение равна 0 - выход

match\_len = archive.getbits(LEN\_LN); //размер совпадение

memcpy(buffer, dict + dict\_pos - match\_pos, match\_len); //копирование match\_len байтов из блока dict + dict\_pos - match\_pos в блок buffer

\_write(out\_file, buffer, match\_len); //запись в раскодированный файл

for (i = 0; i < match\_len; i++)

add\_dict(buffer[i]); //добавление в словарь

if (match\_len < BUF\_LEN) //если размер совпадение меньше размера буфера

{

c = archive.getbits(BYTE\_LN); //получение бита

add\_dict(c); //запись бита в словарь

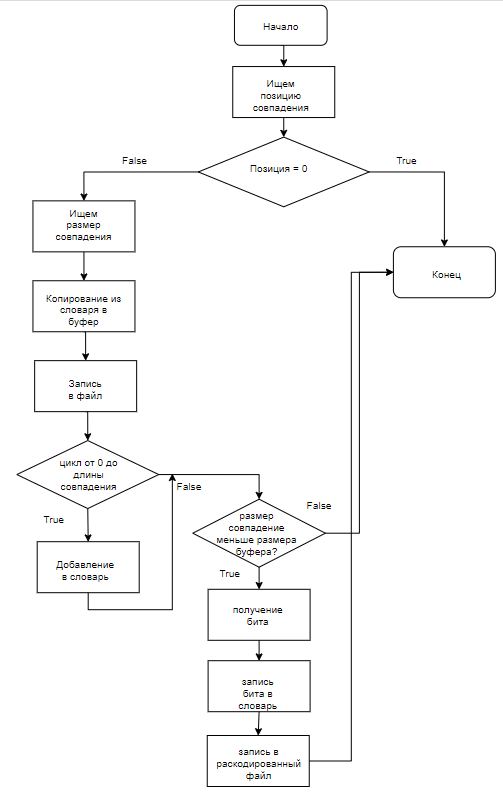
\_write(out\_file, &c, 1); //запись в раскодированный файл

}

}

}

Блок схема процедуры декодирования:



## Программная реализация алгоритма LZW.

Программную реализацию алгоритма было решено писать на языке C++, в обычном консольном приложении.

Программа состоит из процедур кодирования, декодирования, добавления в словарь, а также вспомогательных процедур для ввода или вывода в файл.

Процедура кодирования:

void compress(FILE\* input, FILE\* output) // Процедура сжатия.

{

unsigned int next\_code;

unsigned int character;

unsigned int string\_code;

unsigned int index;

int i;

next\_code = 256; // Next\_code - следующий доступный код строки

for (i = 0; i < TABLE\_SIZE; i++) // Очистка таблицы строк перед стартом

code\_value[i] = -1;

i = 0;

string\_code = getc(input);

// Основной цикл. Он выполняется до тех пор, пока возможно чтение

// входного потока. Отметим, что он прекращает заполнение таблицы

// строк после того, как все возможные коды были использованы.

while ((character = getc(input)) != (unsigned)EOF)

{

index = find\_match(string\_code, character); // Смотрит, есть ли строка

if (code\_value[index] != -1) // в таблице. Если есть,

string\_code = code\_value[index]; // получает значение кода.

else // Если нет, добавляет ее

{ // в таблицу.

if (next\_code <= MAX\_CODE)

{

code\_value[index] = next\_code++;

prefix\_code[index] = string\_code;

append\_character[index] = character;

}

output\_code(output, string\_code); // Когда обнаруживается, что

string\_code = character; // строки нет в таблице,

} // выводится последняя строка

} // перед добавлением новой

output\_code(output, string\_code); // Вывод последнего кода

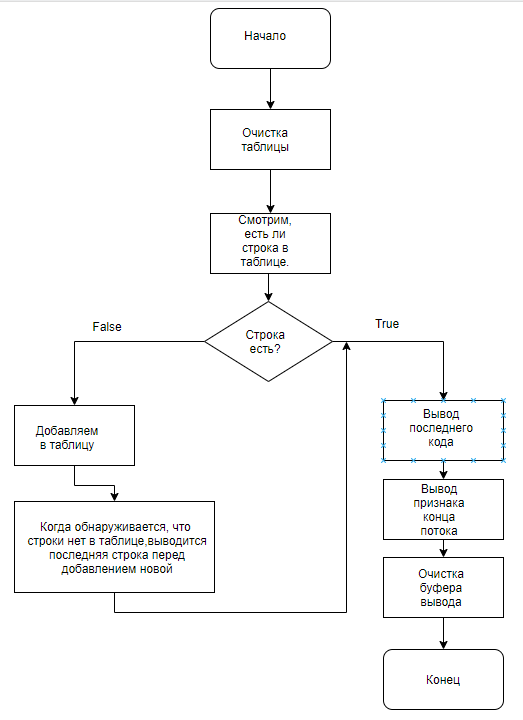
output\_code(output, MAX\_VALUE); // Вывод признака конца потока

output\_code(output, 0); // Очистка буфера вывода

cout << endl;

}

Блок-схема процедуры кодирования:



Процедура декодирования:

// Процедура распаковки.

void decode(FILE\* input, FILE\* output)

{

unsigned int next\_code;

unsigned int new\_code;

unsigned int old\_code;

int character;

int counter;

unsigned char\* stringg;

next\_code = 256; // Следующий доступный код.

counter = 0; // Используется при выводе на экран.

old\_code = input\_code(input); // Читается первый код, инициализируется

character = old\_code; // переменная character и посылается первый

putc(old\_code, output); // код в выходной файл.

// Основной цикл распаковки. Читаются коды из файла до тех пор,

// пока не встретится специальный код, указывающий на конец данных.

while ((new\_code = input\_code(input)) != (MAX\_VALUE))

{

// Проверка кода для специального случая STRING+CHARACTER+STRING+CHARACTER+

// STRING, когда генерируется неопределенный код. Это заставляет его

// декодировать последний код, добавив CHARACTER в конец декод. строки.

if (new\_code >= next\_code)

{

\*decode\_stack = character;

stringg = decode\_string(decode\_stack + 1, old\_code);

}

// Иначе декодируется новый код.

else

stringg = decode\_string(decode\_stack, new\_code);

// Выводится декодируемая строка в обратном порядке.

character = \*stringg;

while ((unsigned char\*)stringg >= decode\_stack)

putc(\*stringg--, output);

// Наконец, если возможно, добавляется новый код в таблицу строк.

if (next\_code <= MAX\_CODE)

{

prefix\_code[next\_code] = old\_code;

append\_character[next\_code] = character;

next\_code++;

}

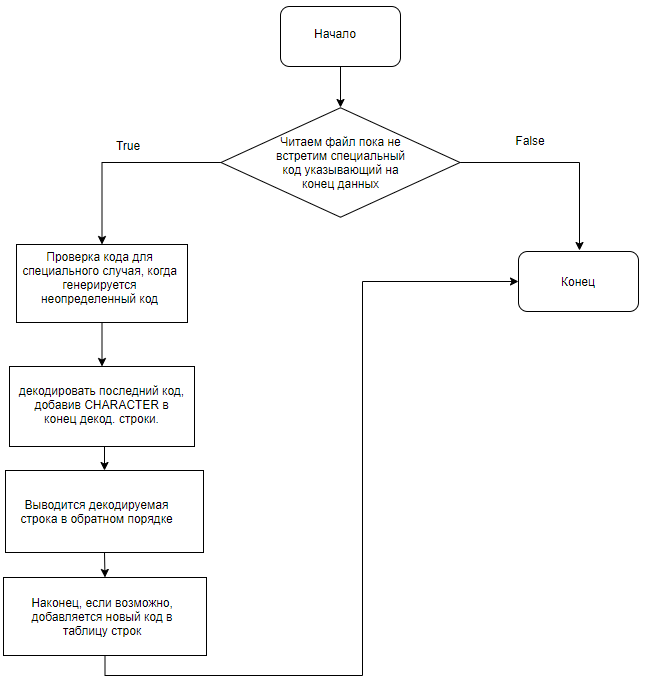
old\_code = new\_code;

}

cout << endl;

}

Блок-схема процедуры декодирования:



Процедура поиска совпадения:

// Процедура хэширования. Она пытается найти сопоставление для строки

// префикс+символ в таблице строк. Если найдено, возвращается индекс.

// Если нет, то возвращается первый доступный индекс.

int find\_match(int hash\_prefix, unsigned int hash\_character)

{

int index;

int offset;

index = (hash\_character << HASHING\_SHIFT) ^ hash\_prefix;

if (index == 0)

offset = 1;

else

offset = TABLE\_SIZE - index;

while (1)

{

if (code\_value[index] == -1)

return(index);

if (prefix\_code[index] == hash\_prefix && append\_character[index] == hash\_character)

return(index);

index -= offset;

if (index < 0)

index += TABLE\_SIZE;

}

}

Процедура декодирования строки из таблицы строк:

// Процедура простого декодирования строки из таблицы строк, сохраняющая

// результат в буфер. Этот буфер потом может быть выведен в обратном

// порядке программой распаковки.

unsigned char\* decode\_string(unsigned char\* buffer, unsigned int code)

{

int i;

i = 0;

while (code > 255)

{

\*buffer++ = append\_character[code];

code = prefix\_code[code];

if (i++ >= 4094)

{

cout << "Fatal error during code expansion." << endl;

exit(0);

}

}

\*buffer = code;

return buffer;

}

Следующие две процедуры управляют вводом/выводом кодов переменной длины:

int input\_code(FILE\* input)

{

unsigned int return\_value;

static int input\_bit\_count = 0;

static unsigned long input\_bit\_buffer = 0L;

while (input\_bit\_count <= 24)

{

input\_bit\_buffer |= (unsigned long)getc(input) << (24 - input\_bit\_count);

input\_bit\_count += 8;

}

return\_value = input\_bit\_buffer >> (32 - BITS);

input\_bit\_buffer <<= BITS;

input\_bit\_count -= BITS;

return(return\_value);

}

void output\_code(FILE\* output, unsigned int code)

{

static int output\_bit\_count = 0;

static unsigned long output\_bit\_buffer = 0L;

output\_bit\_buffer |= (unsigned long)code << (32 - BITS - output\_bit\_count);

output\_bit\_count += BITS;

while (output\_bit\_count >= 8)

{

putc(output\_bit\_buffer >> 24, output);

output\_bit\_buffer <<= 8;

output\_bit\_count -= 8;

}

}

# Экспериментальное исследование реализованных алгоритмов на различных наборах данных

Экспериментальное исследование алгоритмов LZ77 и LZW проводилось на ПК с такими характеристиками:

Процессор: Intel Pentium CPU G4600 3.60GHz 2 ядра 4 потока, Кеш-память: 3MB

ОЗУ: 8,00 ГБ

Система: Windows 10 Pro x64

Данные тестирования занесены в таблицы 4.1, 4.2, которые состоят из:

1) Тип файла

2) Размер файла

3) Время на сжатие

4) Время на распаковку

5) Фактора сжатия

## Тестирование программы LZW

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Тип файла | Размер файла(MB) | Время на сжатие (с) | Время на распаковку (с) | Фактор сжатия |
| .txt | 5.11 | 0.88 | 0.71 | 2.27 |
| .bmp | 0.71 | 0.22 | 0.17 | 0.75 |
| .png | 11.9 | 2.7 | 2.2 | 1.10 |
| .gif | 2.03 | 0.6 | 0.51 | 0.71 |
| .jpg | 2.97 | 0.85 | 0.74 | 0.71 |
| .pdf | 7.26 | 2.03 | 1.78 | 0.73 |

Таблица 4.1

## Тестирование программы LZ77

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Тип файла | Размер файла(MB) | Время на сжатие (с) | Время на распаковку (с) | Фактор сжатия |
| .txt | 5.11 | 68.7 | 19.6 | 1.68 |
| .bmp | 0.71 | 13.3 | 5.8 | 0.71 |
| .png | 11.9 | 222.1 | 70.9 | 1.027 |
| .gif | 2.03 | 45.4 | 17.8 | 0.66 |
| .jpg | 2.97 | 69.5 | 26.7 | 0.66 |
| .pdf | 7.26 | 154.8 | 57.3 | 0.75 |

Таблица 4.2

## Выводы по разделу

Как видно из таблиц, удалось сжать только файлы типов .txt и .png. Лучше всего справилось приложение с LZW алгоритмом:

.txt – 0.8 секунды против 68.8 секунды и 2.27 против 1.68 факторы сжатия;

.png – 2.7 секунды против 222.1 секунды и 1.1 против 1.02 факторы сжатия.

Хоть и в последнем разница факторов сжатия маленькая, но разница во времени сжатия равна 219.

Из этих двух алгоритмов логичнее использовать LZW.

# Заключение

При разработке приложения стало ясно, что люди ежедневно отправляют друг другу сотни файлов, на что тратится время с соответствием размера файла, и чтобы решить эту проблему были разобраны алгоритмы сжатия данных без потерь.

Разработанные приложения имеют вид классического консольного приложения, поэтому файлы для сжатия должны находиться в одном месте с приложением. В дальнейшем планируется добавить возможность выбора места сохранения, а также выбор файла в независимости от его расположения.

При помощи данного курсового проекта были получены навыки работы с информацией, а также навыки работы с C++.

# Библиографический список

* 1. Д.Сэломон. Сжатие данных, изображений и звука. Москва: Техносфера, 2004. - 368с. ISBN 5-94836-027-Х.
  2. Ватолин Д., Ратушняк А., Смирнов М., Юкин В. Методы сжатия данных. Устройство архиваторов, сжатие изображений и видео. - М.: ДИАЛОГ-МИФИ, 2003. - 384 с. ISBN 5-86404-170-х
  3. Weinberger М. J., Seroussi С , Sapiro G. (1996) "LOCO-I: A Low Complexity, Context-Based, Lossless Image Compression Algorithm," in Proceedings of Data Compression Conference, Storer J., editor, Los Alamitos, GA, IEEE Computer Society Press, pp. 140-149.
  4. Salomon D. (2000) Data Compression: The Complete Reference, New York, NY, Springer-Verlag.

# Приложения

## Приложение А. Листинг программы LZ77

#define \_CRT\_SECURE\_NO\_WARNINGS

#include "iostream"

#include <io.h>

#include <fcntl.h>

#include <sys/stat.h>

#include "string"

#include <ctime>

#include <fstream>

#include <tchar.h>

using namespace std;

// Класс для ввода/вывода

class archiving

{

int handle;

unsigned char buffer, mask;

public:

void assign\_read(int h); // связывание для чтения

void assign\_write(int h); // связывание для записи

void write\_zero(); // заполнение нулями

void putbit(bool val); // запись бита

bool getbit(); //чтение бита

void putbits(int val, int n); // запись n битов

int getbits(int n); // чтение n битов

};

// заполнение нулями

void archiving::write\_zero()

{

for (int i = 0; i < 7; i++) putbit(0);

}

// связываение для записи

void archiving::assign\_write(int h)

{

handle = h;

buffer = 0;

mask = 128;

}

// связывание для чтения

void archiving::assign\_read(int h)

{

handle = h;

buffer = 0;

mask = 0;

}

//чтение бита

bool archiving::getbit()

{

if ((mask >>= 1) == 0)

{

\_read(handle, &buffer, 1);

mask = 128;//двоичное 10000000

}

return (buffer & mask) != 0;

}

// запись бита

void archiving::putbit(bool val)

{

if (val) buffer |= mask;

if ((mask >>= 1) == 0)

{

\_write(handle, &buffer, 1);

buffer = 0;

mask = 128;//двоичное 10000000

}

}

// запись n битов

void archiving::putbits(int val, int n)

{

int m = 1;

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

{

putbit((val & m) != 0);

m <<= 1;

}

}

// чтение n битов

int archiving::getbits(int n)

{

int result = 0;

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

result |= getbit() << i;

return result;

}

#define OFFS\_LN 8

#define LEN\_LN 4

#define BYTE\_LN 8

//#define BUF\_LEN 15

#define BUF\_LEN ((1 << LEN\_LN) - 1)

//#define DICT\_LEN 256

#define DICT\_LEN ((1 << OFFS\_LN) - 1)

int in\_file; // входной файл

int out\_file; // выходной файл

static archiving archive;

/\*

Сделаем буфер на 1 больше, т. к. в find\_match

`unmatched = buffer[match\_len];` а match\_len

может достигать BUF\_LEN

\*/

char buffer[BUF\_LEN + 1], dict[DICT\_LEN];

int match\_pos, match\_len, unmatched;

int dict\_pos = 0;

// Возвращает позицию sub от КОНЦА src. от 1 до len

// Если не найдено, возвращает 0

int strpos(char \*src, int len, char \*sub, int sub\_len)

{

for (int i = 0; i <= len - sub\_len; i++)

if (memcmp(src + i, sub, sub\_len) == 0) return len - i;

return 0;

}

// добавление в словарь символа

void add\_dict(char c)

{

if (dict\_pos == (DICT\_LEN - 1)) //если позиция словаря равна размеру словаря-1

{

memcpy(dict, dict + 1, DICT\_LEN - 1); //копирование DICT\_LEN-1 байтов из блока dict+1 в блок dict

dict[dict\_pos - 1] = c; //добавление символа в словарь на позицию dict\_pos-1

}

else //иначе

{

dict[dict\_pos] = c; //добавление символа с словарь на позицию dict\_pos

dict\_pos++; //увеличение позиции

}

}

// поиск совпадения в словаре

void find\_match()

{

match\_len = 0;//длина совпадения

match\_pos = 1;//позиция совпадения

while (match\_len < BUF\_LEN) //пока размер совпадения меньше размера буфера

{

\_read(in\_file, &buffer[match\_len], 1); //чтение файла

if (\_eof(in\_file)) break; //если файл пустой – выход

int pos1 = strpos(dict, dict\_pos, buffer, match\_len + 1);

if (pos1 == 0) break; //если позиция равна 0 - выход

match\_pos = pos1; //указывание позиции совпадения

match\_len++; //увеличение длины совпадения

}

unmatched = buffer[match\_len];

}

// компрессия

void encode()

{

while(!\_eof(in\_file)) //пока файл не пуст

{

find\_match(); //ищем совпадение

archive.putbits(match\_pos, OFFS\_LN); //запись в сжатый файл

archive.putbits(match\_len, LEN\_LN); //запись в сжатый файл

if (match\_len < BUF\_LEN) //если размер совпадение меньше размера буфера

archive.putbits(unmatched, BYTE\_LN); //запись в сжатый файл

for (int i = 0; i < match\_len; i++)

add\_dict(buffer[i]); //добавление в словарь

if (match\_len < BUF\_LEN) //если размер совпадение меньше размера буфера

add\_dict(unmatched); //добавление в словарь

}

archive.putbits(0, BYTE\_LN); //запись нулей

archive.write\_zero(); //заполнение нулями

}

// декомпрессия

void decode()

{

char c;

int i;

for(;;) //пустой цикл для того чтобы потом можно было выйти

{

match\_pos = archive.getbits(OFFS\_LN); //позиция совпадения

if (match\_pos == 0) break; //если позиция совпадение равна 0 - выход

match\_len = archive.getbits(LEN\_LN); //размер совпадение

memcpy(buffer, dict + dict\_pos - match\_pos, match\_len); //копирование match\_len байтов из блока dict + dict\_pos - match\_pos в блок buffer

\_write(out\_file, buffer, match\_len); //запись в раскодированный файл

for (i = 0; i < match\_len; i++)

add\_dict(buffer[i]); //добавление в словарь

if (match\_len < BUF\_LEN) //если размер совпадение меньше размера буфера

{

c = archive.getbits(BYTE\_LN);//получение бита

add\_dict(c);//запись бита в словарь

\_write(out\_file, &c, 1); //запись в раскодированный файл

}

}

}

// главная процедура

int main()

{

setlocale(LC\_ALL, "rus");

string intro=R"(Введите режим работы:

e - сжатие

d - распаковка)";

char input\_filename[100];

char output\_filename[100] = "out.lz77";

cout << "Введите название файла:" << endl << ">";

cin >> input\_filename;

fstream file(input\_filename);

int size = 0;

file.seekg(0, std::ios::end);

size = file.tellg();

file.close();

double size1 = (double)size / 1048576;

cout << intro << endl << "\n>";

string tmp;

cin >> tmp;

system("cls");

if (tmp == "d")

{

cout << "Введите старое название файла:" << endl << "> ";

cin >> output\_filename;

}

// открытие входного и выходного файлов

in\_file = \_open(input\_filename, \_O\_BINARY | \_O\_RDWR, \_S\_IREAD | \_S\_IWRITE);

out\_file = \_open(output\_filename, \_O\_BINARY | \_O\_WRONLY | \_O\_CREAT | \_O\_TRUNC, \_S\_IREAD | \_S\_IWRITE);

unsigned int start = clock();

if (tmp == "e") // компрессия

{

archive.assign\_write(out\_file);

encode();

}

else if (tmp == "d") // декомпрессия

{

archive.assign\_read(in\_file);

decode();

}

else printf("Nothing to do\n");

\_close(in\_file);

\_close(out\_file);

cout << "Время работы в миллисекундах: " << clock() - start << endl << endl << endl;

fstream file1(output\_filename);

file1.seekg(0, std::ios::end);

int size2 = file1.tellg();

file1.close();

double size3 = (double)size2 / 1048576;

cout << "Входной файл весит : " << size << " байт -> ~" << size1 << " мб" << endl;

cout << "Выходной файл весит : " << size2 << " байт -> ~" << size3 <<" мб" << endl;

double k = (double)size / size2;

if (tmp == "e")

cout << "Коэффициент сжатия= " << k << endl;

system("pause");

return 0;

}

## Приложение Б. Листинг программы LZW

#define \_CRT\_SECURE\_NO\_WARNINGS

#include "iostream"

#include <io.h>

#include <fcntl.h>

#include <sys/stat.h>

#include <ctime>

#include <fstream>

#include <cstdlib>

#include <string>

using namespace std;

#define BITS 12 //Установка длины кода равной 12, 13, 14

#define HASHING\_SHIFT BITS-8

#define MAX\_VALUE (1 << BITS) - 1

#define MAX\_CODE MAX\_VALUE - 1

#if BITS == 14

#define TABLE\_SIZE 18041 // Размер таблицы строк должен быть

#endif // простым числом, несколько большим,

#if BITS == 13 // чем 2\*BITS.

#define TABLE\_SIZE 9029

#endif

#if BITS <= 12

#define TABLE\_SIZE 5021

#endif

int\* code\_value; // Это массив для значений кодов

unsigned int\* prefix\_code; // Этот массив содержит префиксы кодов

unsigned char\* append\_character; // Этот массив содержит добавочные символы

unsigned char decode\_stack[4000]; // Этот массив содержит декодируемые строки

void decode(FILE\* input, FILE\* output);

void compress(FILE\* input, FILE\* output);

int find\_match(int hash\_prefix, unsigned int hash\_character);

unsigned char\* decode\_string(unsigned char\* buffer, unsigned int code);

int input\_code(FILE\* input);

void output\_code(FILE\* output, unsigned int code);

void compress(FILE\* input, FILE\* output) // Процедура сжатия.

{

unsigned int next\_code;

unsigned int character;

unsigned int string\_code;

unsigned int index;

int i;

next\_code = 256; // Next\_code - следующий доступный код строки

for (i = 0; i < TABLE\_SIZE; i++) // Очистка таблицы строк перед стартом

code\_value[i] = -1;

i = 0;

string\_code = getc(input);

// Основной цикл. Он выполняется до тех пор, пока возможно чтение

// входного потока. Отметим, что он прекращает заполнение таблицы

// строк после того, как все возможные коды были использованы.

while ((character = getc(input)) != (unsigned)EOF)

{

index = find\_match(string\_code, character); // Смотрит, есть ли строка

if (code\_value[index] != -1) // в таблице. Если есть,

string\_code = code\_value[index]; // получает значение кода.

else // Если нет, добавляет ее

{ // в таблицу.

if (next\_code <= MAX\_CODE)

{

code\_value[index] = next\_code++;

prefix\_code[index] = string\_code;

append\_character[index] = character;

}

output\_code(output, string\_code); // Когда обнаруживается, что

string\_code = character; // строки нет в таблице,

} // выводится последняя строка

} // перед добавлением новой

output\_code(output, string\_code); // Вывод последнего кода

output\_code(output, MAX\_VALUE); // Вывод признака конца потока

output\_code(output, 0); // Очистка буфера вывода

printf("\n");

}

// Процедура хэширования. Она пытается найти сопоставление для строки

// префикс+символ в таблице строк. Если найдено, возвращается индекс.

// Если нет, то возвращается первый доступный индекс.

int find\_match(int hash\_prefix, unsigned int hash\_character)

{

int index;

int offset;

index = (hash\_character << HASHING\_SHIFT) ^ hash\_prefix;

if (index == 0)

offset = 1;

else

offset = TABLE\_SIZE - index;

while (1)

{

if (code\_value[index] == -1)

return(index);

if (prefix\_code[index] == hash\_prefix && append\_character[index] == hash\_character)

return(index);

index -= offset;

if (index < 0)

index += TABLE\_SIZE;

}

}

// Процедура распаковки.

void decode(FILE\* input, FILE\* output)

{

unsigned int next\_code;

unsigned int new\_code;

unsigned int old\_code;

int character;

int counter;

unsigned char\* stringg;

next\_code = 256; // Следующий доступный код.

counter = 0; // Используется при выводе на экран.

old\_code = input\_code(input); // Читается первый код, инициализируется

character = old\_code; // переменная character и посылается первый

putc(old\_code, output); // код в выходной файл.

// Основной цикл распаковки. Читаются коды из файла до тех пор,

// пока не встретится специальный код, указывающий на конец данных.

while ((new\_code = input\_code(input)) != (MAX\_VALUE))

{

// Проверка кода для специального случая STRING+CHARACTER+STRING+CHARACTER+

// STRING, когда генерируется неопределенный код. Это заставляет его

// декодировать последний код, добавив CHARACTER в конец декод. строки.

if (new\_code >= next\_code)

{

\*decode\_stack = character;

stringg = decode\_string(decode\_stack + 1, old\_code);

}

// Иначе декодируется новый код.

else

stringg = decode\_string(decode\_stack, new\_code);

// Выводится декодируемая строка в обратном порядке.

character = \*stringg;

while ((unsigned char\*)stringg >= decode\_stack)

putc(\*stringg--, output);

// Наконец, если возможно, добавляется новый код в таблицу строк.

if (next\_code <= MAX\_CODE)

{

prefix\_code[next\_code] = old\_code;

append\_character[next\_code] = character;

next\_code++;

}

old\_code = new\_code;

}

cout << endl;

}

// Процедура простого декодирования строки из таблицы строк, сохраняющая

// результат в буфер. Этот буфер потом может быть выведен в обратном

// порядке программой распаковки.

unsigned char\* decode\_string(unsigned char\* buffer, unsigned int code)

{

int i;

i = 0;

while (code > 255)

{

\*buffer++ = append\_character[code];

code = prefix\_code[code];

if (i++ >= 4094)

{

cout << "Fatal error during code expansion." << endl;

exit(0);

}

}

\*buffer = code;

return buffer;

}

// Следующие две процедуры управляют вводом/выводом кодов переменной длины.

int input\_code(FILE\* input)

{

unsigned int return\_value;

static int input\_bit\_count = 0;

static unsigned long input\_bit\_buffer = 0L;

while (input\_bit\_count <= 24)

{

input\_bit\_buffer |= (unsigned long)getc(input) << (24 - input\_bit\_count);

input\_bit\_count += 8;

}

return\_value = input\_bit\_buffer >> (32 - BITS);

input\_bit\_buffer <<= BITS;

input\_bit\_count -= BITS;

return(return\_value);

}

void output\_code(FILE\* output, unsigned int code)

{

static int output\_bit\_count = 0;

static unsigned long output\_bit\_buffer = 0L;

output\_bit\_buffer |= (unsigned long)code << (32 - BITS - output\_bit\_count);

output\_bit\_count += BITS;

while (output\_bit\_count >= 8)

{

putc(output\_bit\_buffer >> 24, output);

output\_bit\_buffer <<= 8;

output\_bit\_count -= 8;

}

}

int main()

{

setlocale(LC\_ALL, "rus");

FILE\* input\_file;

FILE\* output\_file;

char input\_filename[100];

char output\_filename[100] = "out.lzw";

cout << "Введите название файла:" << endl << "> ";

cin >> input\_filename;

fstream file(input\_filename);

int size = 0;

file.seekg(0, std::ios::end);

size = file.tellg();

file.close();

double size1 = (double)size / 1048576;

// Эти три буфера необходимы на стадии упаковки.

code\_value = (int\*)malloc(TABLE\_SIZE \* sizeof(unsigned int));

prefix\_code = (unsigned int\*)malloc(TABLE\_SIZE \* sizeof(unsigned int));

append\_character = (unsigned char\*)malloc(TABLE\_SIZE \* sizeof(unsigned char));

if (code\_value == NULL || prefix\_code == NULL || append\_character == NULL)

{

cout << "Fatal error allocating table space!" << endl;

exit(0);

}

string intro = R"(Введите режим работы:

e - сжатие

d - распаковка)";

cout << intro << endl << "\n>";

string tmp;

cin >> tmp;

system("cls");

if (tmp == "d")

{

cout << "Введите старое название файла:" << endl << "> ";

cin >> output\_filename;

}

input\_file = fopen(input\_filename, "rb");

output\_file = fopen(output\_filename, "wb");

if (input\_file == NULL || output\_file == NULL)

{

printf("Fatal error opening files.\n");

exit(0);

}

unsigned int start = clock();

if (tmp == "e") //compress

{

compress(input\_file, output\_file);

fclose(input\_file);

fclose(output\_file);

free(code\_value);

}

else

{

if (tmp == "d")//decompress

{

input\_file = fopen(input\_filename, "rb");

output\_file = fopen(output\_filename, "wb");

if (input\_file == NULL || output\_file == NULL)

{

cout << "Fatal error opening files." << endl;

exit(0);

};

decode(input\_file, output\_file);

fclose(input\_file);

fclose(output\_file);

free(prefix\_code);

free(append\_character);

}

else

{

cout << "Выход." << endl;

exit(0);

}

}

cout << "Время работы в миллисекундах: " << clock() - start << endl << endl << endl;

fstream file1(output\_filename);

file1.seekg(0, std::ios::end);

int size2 = file1.tellg();

file1.close();

double size3 = (double)size2 / 1048576;

cout << "Входной фаил весит: " << size << " байт -> ~" << size1 << " мб" << endl;

cout << "Выходной фаил весит: " << size2 << " байт -> ~" << size3 << " мб" << endl << endl;

double k = (double)size / size2;

if (tmp == "e")

cout << "Коэффициент сжатия= " << k << endl;

system("pause");

return 0;

}