# Сжатие данных методами Хафмана и Шеннона-Фано

Введение

Думая о данных, обычно мы представляем себе ни что иное, как передаваемую этими данными информацию: список клиентов, мелодию на аудио компакт-диске, письмо и тому подобное. Как правило, мы не слишком задумываемся о физическом представлении данных. Заботу об этом - отображении списка клиентов, воспроизведении компакт-диска, печати письма - берет на себя программа, манипулирующая данными.

### 1. Представление данных

Рассмотрим двойственность природы данных: с одной стороны, содержимое информации, а с другой - ее физическое представление. В 1950 году Клод Шеннон (Claude Shannon) заложил основы теории информации, в том числе идею о том, что данные могут быть представлены определенным минимальным количеством битов. Эта величина получила название энтропии данных (термин был заимствован из термодинамики). Шеннон установил также, что обычно количество бит в физическом представлении данных превышает значение, определяемое их энтропией.

В качестве простого примера рассмотрим исследование понятия вероятности с помощью монеты. Можно было бы подбросить монету множество раз, построить большую таблицу результатов, а затем выполнить определенный статистический анализ этого большого набора данных с целью формулирования или доказательства какой-то теоремы. Для построения набора данных, результаты подбрасывания монеты можно было бы записывать несколькими различными способами: можно было бы записывать слова "орел" или "решка"; можно было бы записывать буквы "О" или "Р"; или же можно было бы записывать единственный бит (например "да" или "нет", в зависимости от того, на какую сторону падает монета). Согласно теории информации, результат каждого подбрасывания монеты можно закодировать единственным битом, поэтому последний приведенный вариант был бы наиболее эффективным с точки зрения объема памяти, необходимого для кодирования результатов. С этой точки зрения первый вариант является наиболее расточительным, поскольку для записи результата единственного подбрасывания монеты требовалось бы четыре или пять символов.

Однако посмотрим на это под другим углом: во всех приведенных примерах записи данных мы сохраняем одни и те же результаты - одну и ту же информацию - используя все меньший и меньший объем памяти. Другими словами, мы выполняем сжатие данных.

### 1.1.Сжатие данных

Сжатие данных (data compression) - это алгоритм эффективного кодирования информации, при котором она занимает меньший объем памяти, нежели ранее. Мы избавляемся от избыточности (redundancy), т.е. удаляем из физического представления данных те биты, которые в действительности не требуются, оставляя только то количество битов, которое необходимо для представления информации в соответствии со значением энтропии. Существует показатель эффективности сжатия данных: коэффициент сжатия (compression ratio). Он вычисляется путем вычитания из единицы частного от деления размера сжатых данных на размер исходных данных и обычно выражается в процентах. Например, если размер сжатых данных равен 1000 бит, а несжатых - 4000 бит, коэффициент сжатия составит 75%, т.е. мы избавились от трех четвертей исходного количества битов.

Конечно, сжатые данные могут быть записаны в форме недоступной для непосредственного считывания и понимания человеком. Люди нуждаются в определенной избыточности представления данных, способствующей их эффективному распознаванию и пониманию. Применительно к эксперименту с подбрасыванием монеты последовательности символов "О" и "Р" обладают большей наглядностью, чем 8-битовые значения байтов. (Возможно, что для большей наглядности пришлось бы разбить последовательности символов "О" и "Р" на группы, скажем, по 10 символов в каждой.) Иначе говоря, возможность выполнения сжатия данных бесполезна, если отсутствует возможность их последующего восстановления. Эту обратную операцию называют декодированием (decoding).

### 1.2 Типы сжатия

Существует два основных типа сжатия данных: с потерями (lossy) и без потерь (lossless). Сжатие без потерь проще для понимания. Это метод сжатия данных, когда при восстановлении данных возвращается точная копия исходных данных. Такой тип сжатия используется программой PKZIB®1: распаковка упакованного файла приводит к созданию файла, который имеет в точности то же содержимое, что и оригинал перед его сжатием. И напротив, сжатие с потерями не позволяет при восстановлении получить те же исходные данные. Это кажется недостатком, но для определенных типов данных, таких как данные изображений и звука, различие между восстановленными и исходными данными не имеет особого значения: наши зрение и слух не в состоянии уловить образовавшиеся различия. В общем случае алгоритмы сжатия с потерями обеспечивают более эффективное сжатие, чем алгоритмы сжатия без потерь (в противном случае их не стоило бы использовать вообще). Для примера можно сравнить предназначенный для хранения изображений формат с потерями JPEG с форматом без потерь GIF. Множество форматов потокового аудио и видео, используемых в Internet для загрузки мультимедиа-материалов, являются алгоритмами сжатия с потерями.

В случае экспериментов с подбрасыванием монеты было очень легко определить наилучший способ хранения набора данных. Но для других данных эта задача становится более сложной. При этом можно применить несколько алгоритмических подходов. Два класса сжатия, которые будут рассмотрены в этой главе, представляют собой алгоритмы сжатия без потерь и называются кодированием с минимальной избыточностью (minimum redundancy coding) и сжатием с применением словаря (dictionary compression).

Кодирование с минимальной избыточностью - это метод кодирования байтов (или, более строго, символов), при котором чаще встречающиеся байты кодируются меньшим количеством битов, чем те, которые встречаются реже. Например, в тексте на английском языке буквы Е, Т и А встречаются чаще, нежели буквы Q, X и Z. Поэтому, если бы удалось закодировать буквы Е, Т и А меньшим количеством битов, чем 8 (как должно быть в соответствии со стандартом ASCII), а буквы Q, X и Z - большим, текст на английском языке удалось бы сохранить с использованием меньшего количества битов, чем при соблюдении стандарта ASCII.

При использовании сжатия с применением словаря данные разбиваются на большие фрагменты (называемые лексемами), чем символы. Затем применяется алгоритм кодирования лексем определенным минимальным количеством битов. Например, слова "the", "and" и "to" будут встречаться чаще, чем такие слова, как "electric", "ambiguous" и "irresistible", поэтому их нужно закодировать меньшим количеством битов, чем требовалось бы при кодировании в соответствии со стандартом ASCII.

### 2. Сжатие с минимальной избыточностью

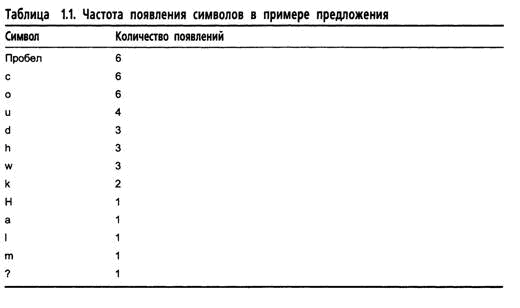
Теперь, когда в нашем распоряжении имеется класс потока битов, им можно воспользоваться при рассмотрении алгоритмов сжатия и восстановления данных. Мы начнем с исследования алгоритмов кодирования с минимальной избыточностью, а затем рассмотрим более сложное сжатие с применением словаря.

Мы приведем подробное описание трех алгоритмов кодирования с минимальной избыточностью: кодирование Шеннона-Фано (Shannon-Fano), кодирование Хаффмана (Haffman) и сжатие с применением скошенного дерева (splay tree compression), однако рассмотрим реализации только последних двух алгоритмов (алгоритм кодирования Хаффмана ни в чем не уступает, а кое в чем даже превосходит алгоритм кодирования Шеннона Фано). При использовании каждого из этих алгоритмов входные данные анализируются как поток байтов, и различным значениям байтов тем или иным способом присваиваются различные последовательности битов.

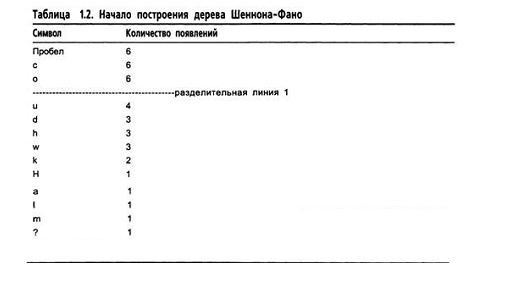
### 2.1.Кодирование Шеннона-Фано

Первый алгоритм сжатия, который мы рассмотрим - кодирование Шеннона-Фано, названное так по имени двух исследователей, которые одновременно и независимо друг от друга разработали этот алгоритм: Клода Шеннона (Claude Shannon) и Р. М. Фано (R. М. Fano). Алгоритм анализирует входные данные и на их основе строит бинарное дерево минимального кодирования. Используя это дерево, затем можно выполнить повторное считывание входных данных и закодировать их.

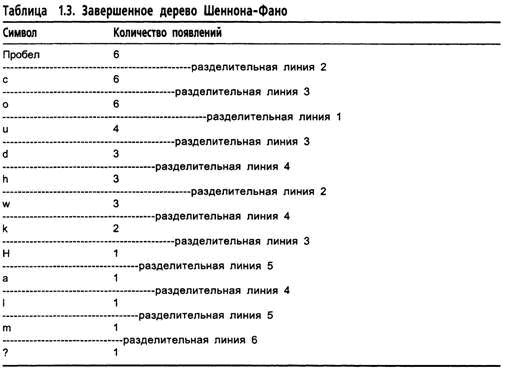
Чтобы проиллюстрировать работу алгоритма, выполним сжатие предложения "How much wood could a woodchuck chuck?" ("Сколько дров мог бы заготовить дровосек?") Прежде всего, предложение необходимо проанализировать. Просмотрим данные и вычислим, сколько раз в предложении встречается каждый символ. Занесем результаты в таблицу (см. таблицу 1.1).



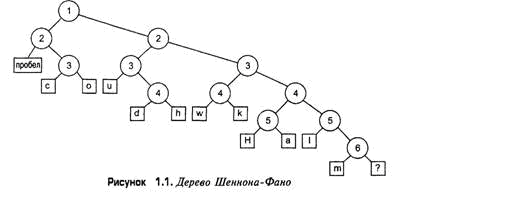
Теперь разделим таблицу на две части, чтобы общее число появлений символов в верхней половине таблицы приблизительно равнялось общему числу появлений в нижней половине. Предложение содержит 38 символов, следовательно, верхняя половина таблицы должна отражать приблизительно 19 появлений символов. Это просто: достаточно поместить разделительную линию между строкой o и строкой u. В результате этого верхняя половина таблицы будет отражать появление 18 символов, а нижняя - 20. Таким образом, мы получаем таблицу 1.2.



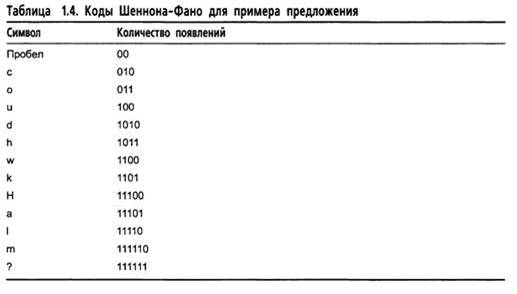
Теперь проделаем то же с каждой из частей таблицы: вставим линию между строками так, чтобы разделить каждую из частей. Продолжим этот процесс, пока все буквы не окажутся разделенными одна от другой. Результирующее дерево Шеннона-Фано представлено в таблице 1.3.



Я намеренно изобразил разделительные линии различными по длине, чтобы разделительная линия 1 была самой длинной, разделительная линия 2 немного короче и так далее, вплоть до самой короткой разделительной линии 6. Этот подход обусловлен тем, что разделительные линии образуют повернутое на 90° бинарное дерево (чтобы убедиться в этом, поверните таблицу на 90° против часовой стрелки). Разделительная линия 1 является корневым узлом дерева, разделительные линии 2 - двумя его дочерними узлами и т.д. Символы образуют листья дерева. Результирующее дерево в обычной ориентации показано на рис.1.1



Все это очень хорошо, но как оно помогает решить задачу кодирования каждого символа и выполнения сжатия? Что ж, чтобы добраться до символа пробела, мы начинаем с коневого узла, перемещаемся влево, а затем снова влево. Чтобы добраться до символа c, мы смещаемся влево из корневого узла, затем вправо, а затем влево. Для перемещения к символу o потребуется сместиться влево, а затем два раза вправо. Если принять, что перемещение влево эквивалентно нулевому биту, а вправо - единичному, можно создать таблицу кодирования, приведенную в таблице 11.4.



Cодержит всего 131 бит. Если мы предполагаем, что исходная фраза закодирована кодом ASCII, т.е. один байт на символ, то оригинальная фраза заняла бы 256 байт, т.е. мы получаем коэффициент сжатия 54%.

Для декодирования сжатого потока битов мы строим то же дерево, которое было построено на этапе сжатия. Мы начинаем с корневого узла и выбираем из сжатого потока битов по одному биту. Если бит является нулевым, мы перемещаемся влево, если единичным - вправо. Мы продолжаем этот процесс до тех пор, пока не достигнем листа, т.е. символа, после чего выводим символ в поток восстановленных данных. Затем мы снова начинаем процесс с корневого узла дерева с целью извлечения следующего бита. Обратите внимание, что поскольку символы расположены только в листьях дерева, код одного символа не образует первую часть кода другого символа. Благодаря этому, неправильное декодирование сжатых данных невозможно. (Бинарное дерево, в котором данные размещены только в листьях, называется префиксным деревом (prefix tree).)

Однако при этом возникает небольшая проблема: как распознать конец потока битов? В конце концов, внутри класса мы будем объединять восемь битов в байт, после чего выполнять запись байта. Маловероятно, чтобы поток битов содержал количество битов строго кратное 8. Существует два возможных решения этой дилеммы. Первое - закодировать специальный символ, отсутствующий в исходных данных, и назвать его символом конца файла. Второе - записать в сжатый поток длину несжатых данных перед тем, как приступить к сжатию самих данных. Первое решение вынуждает нас найти отсутствующий в исходных данных символ и использовать его (это предполагает передачу этого символа в составе сжатых данных программе восстановления, чтобы она знала, что следует искать). Или же можно было бы принять, что хотя символы данных имеют размер, равный размеру одного байта, символ конца файла имеет длину, равную длину слова (и заданное значение, например 256). Однако мы будем использовать второе решение. Перед сжатыми данными мы будем сохранять длину несжатых данных, и таким образом во время восстановления будет в точности известно, сколько символов нужно декодировать.

Еще одна проблема применения кодирования Шеннона-Фано, на которую до сих пор мы не обращали внимания, связана с деревом. Обычно сжатие данных выполняется в целях экономии объема памяти или уменьшения времени передачи данных. Как правило, сжатие и восстановление данных разнесено во времени и пространстве. Однако алгоритм восстановления требует использования дерева. В противном случае невозможно декодировать закодированный поток. Нам доступны две возможности. Первая - сделать дерево статическим. Иначе говоря, одно и то же дерево будет использоваться для сжатия всех данных. Для некоторых данных результирующее сжатие будет достаточно оптимальным, для других весьма далеким от приемлемого. Вторая возможность состоит в том, чтобы тем или иным способом присоединить само дерево к сжатому потоку битов. Конечно, присоединение дерева к сжатым данным ведет к снижению коэффициента сжатия, но с этим ничего нельзя поделать.

Листинг программы осуществляющей сжатие данных методом Шеннона приведён в приложении 1.

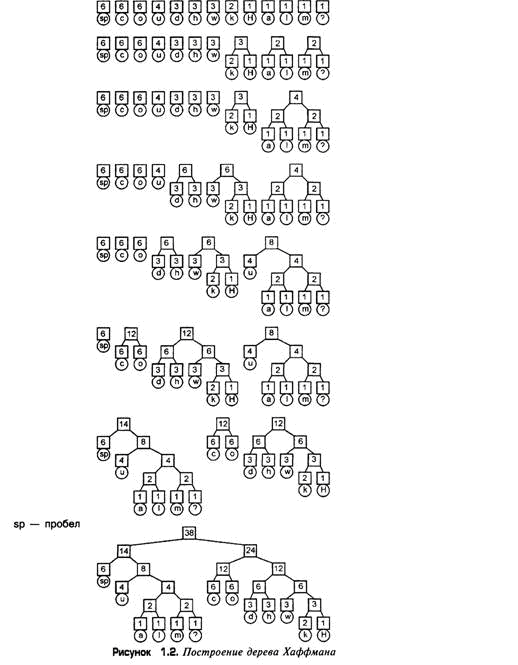
### 2.2.Кодирование Хаффмана

Алгоритм кодирования Хаффмана очень похож на алгоритм сжатия Шеннона-Фано. Этот алгоритм был изобретен Девидом Хаффманом (David Huffman) в 1952 году ("A method for the Construction of Minimum-Redundancy Codes" ("Метод создания кодов с минимальной избыточностью")), и оказался еще более удачным, чем алгоритм Шеннона-Фано. Это обусловлено тем, что алгоритм Хаффмана математически гарантированно создает наименьший по размеру код для каждого из символов исходных данных.

Аналогично применению алгоритма Шеннона-Фано, нужно построить бинарное дерево, которое также будет префиксным деревом, где все данные хранятся в листьях. Но в отличие от алгоритма Шеннона-Фано, который является нисходящим, на этот раз построение будет выполняться снизу вверх. Вначале мы выполняем просмотр входных данных, подсчитывая количество появлений значений каждого байта, как это делалось и при использовании алгоритма Шеннона-Фано. Как только эта таблица частоты появления символов будет создана, можно приступить к построению дерева.

Будем считать эти пары символ-количество "пулом" узлов будущего дерева Хаффмана. Удалим из этого пула два узла с наименьшими значениями количества появлений. Присоединим их к новому родительскому узлу и установим значение счетчика родительского узла равным сумме счетчиков его двух дочерних узлов. Поместим родительский узел обратно в пул. Продолжим этот процесс удаления двух узлов и добавления вместо них одного родительского узла до тех пор, пока в пуле не останется только один узел. На этом этапе можно удалить из пула один узел. Он является корневым узлом дерева Хаффмана.

Описанный процесс не очень нагляден, поэтому создадим дерево Хаффмана для предложения "How much wood could a woodchuck chuck?" Мы уже вычислили количество появлений символов этого предложения и представили их в виде таблицы 11.1, поэтому теперь к ней потребуется применить описанный алгоритм с целью построения полного дерева Хаффмана. Выберем два узла с наименьшими значениями. Существует несколько узлов, из которых можно выбрать, но мы выберем узлы "m" и "?". Для обоих этих узлов число появлений символов равно 1. Создадим родительский узел, значение счетчика которого равно 2, и присоединим к нему два выбранных узла в качестве дочерних. Поместим родительский узел обратно в пул. Повторим цикл с самого начала. На этот раз мы выбираем узлы "a" и "1", объединяем их в мини-дерево и помещаем родительский узел (значение счетчика которого снова равно 2) обратно в пул. Снова повторим цикл. На этот раз в нашем распоряжении имеется единственный узел, значение счетчика которого равно 1 (узел "H") и три узла со значениями счетчиков, равными 2 (узел "к" и два родительских узла, которые были добавлены перед этим). Выберем узел "к", присоединим его к узлу "Н" и снова добавим в пул родительский узел, значение счетчика которого равно 3. Затем выберем два родительских узла со значениями счетчиков, равными 2, присоединим их к новому родительскому узлу со значением счетчика, равным 4, и добавим этот родительский узел в пул. Несколько первых шагов построения дерева Хаффмана и результирующее дерево показаны на рис. 1.2.



Используя это дерево точно так же, как и дерево, созданное для кодирования Шенона-Фано, можно вычислить код для каждого из символов в исходном предложении и построить таблицу 11.5.

Следует обратить внимание на то, что таблица кодов - не единственная возможная. Каждый раз, когда имеется три или больше узлов, из числа которых нужно выбрать два, существуют альтернативные варианты результирующего дерева и, следовательно, результирующих кодов. Но на практике все эти возможные варианты деревьев и кодов будут обеспечивать максимальное сжатие. Все они эквивалентны.

Повторим снова, что, как и при применении алгоритма Шеннона-Фано, необходимо каким-то образом сжать дерево и включить его в состав сжатых данных.

Восстановление выполняется совершенно так же, как при использовании кодирования Шеннона-Фано: необходимо восстановить дерево из данных, хранящихся в сжатом потоке, и затем воспользоваться им для считывания сжатого потока битов.

Листинг программы осуществляющей сжатие данных методом Хаффмана приведён в приложении 2.

На рис.2.1. Показан вид окна работающей программы.

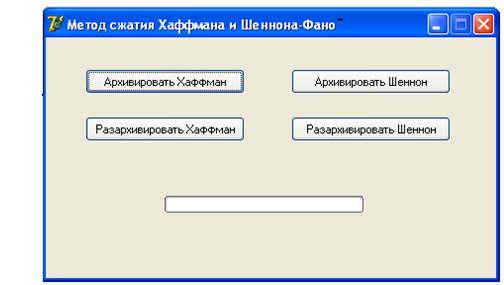


Рис.2.1 Вид окна работающей программы

Выводы

В задании к курсовой работе была задана проверка работы программы по сжатию файлов формата .bmp и .xls. Сжав файлы этих форматов получил следующие результаты. Для .bmp формата рисунок 2.2. Для .xsl формата рисунок 2.3. Отсюда можно сделать вывод, что используя метод Хаффмана можно достичь большего коэффициента сжатия, чем по методу Шеннона. Для файлов типа .bmp коэффициент сжатия выше чем для .xls.

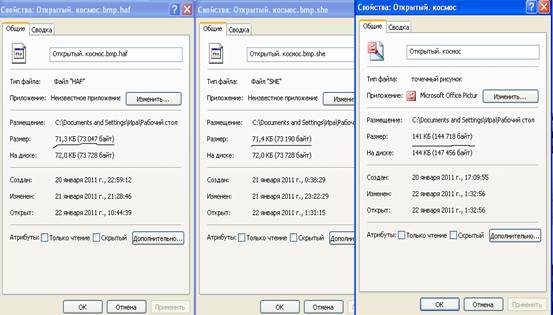


Рис.2.2. Результаты по сжатию одного и того же .bmp файла

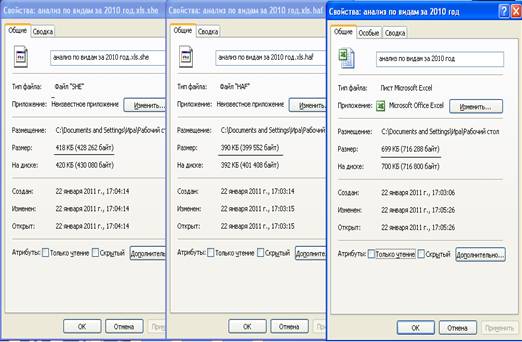


Рис.2.2 Результаты по сжатию одного и того же .xls файла

Литература

1. Фундаментальные алгоритмы с структуры данных в Delphi: Пер. с англ. /Джулиан М. Бакнел. – СПб: ООО «ДиаСофтЮП», 2003.- 560 с.

2. Искусство дизассемблирования К.Касперски Е.Рокко, БХВ-Петербург 2008. -780 с.

3. Win32 API. Эффективная разработка приложений. – СПб.: Питер, 2007 – 572 с.: ил.

4. Жоголев Е.А. Ж.78 Технология программирования. – М., Научный Мир, 2004, 216 с.

5. Фундаментальные алгоритмы на C++. Анализ/Структуры данных/Сортировка/Поиск: Пер. с англ./Роберт Седжвик. - К.: Издательство «ДиаСофт», 2001.- 688 с.

6. Искусство программирования на Ассемблере. Лекции и упражнения: Голубь Н.Г. – 2-е изд., испр. и доп. – СПб: ООО «ДиаСофтЮП». 2002. – 656 с.

Приложение 1

Реализация на Delphi алгоритма сжатия Шеннона

Листинг программы с комментариями

unit Shannon;

interface

Uses

 Forms, Dialogs;

const

 Count=4096;

 ArchExt='she';

 dot='.';

//две файловые переменные для чтения исходного файла и для

//записи архива

var

 FileToRead,FileToWrite: File;

 Str1:String='';

// Процедуры для работы с файлом

// Первая - кодирование файла

procedure RunEncodeShan(FileName\_: string);

// Вторая - декодирование файла

procedure RunDecodeShan(FileName\_: string);

implementation

Type

 //тип элемета для динамической обработки статистики байтов

 TByte=^PByte;

 PByte=Record

 //Символ (один из символв ASCII)

 Symbol: Byte;

 //статистика символа

 SymbolStat: Integer;

 //последовательность битов, в которые преобразуется текущий

 //элемент после работы древа (Кодовое слово) (в виде строки из "0" и "1")

 CodWord: String;

 //ссылки на левое и правое поддеревья (ветки)

 left, right: TByte;

 End;

//массив из символов со статистикой , т.е. частотой появления их

//в архивируемом файле

 BytesWithStat = Array [0..255] of TByte;

 //объект, включающий в себя:

 // массив элементов содержащий в себе количество элементов,

 // встречающихся в файле хотя бы один раз

 // процедура инициализации объекта

 // процедура для увеличения частоты i-го элемента

 TStat =Object

 massiv: BytesWithStat;

 CountByte: byte;

 Procedure Create;//процера инициализации обьекта

 Procedure Inc(i: Byte);

 End;

//процедура инициализации объекта вызввается из

 Procedure TStat.Create;

 var

 i: Byte;

 Begin

 CountByte:=255;

 For i:=0 to CountByte do

 Begin

 New(massiv[i]);//создаём динамическую переменную

 //и устанавливаем указатель на неё

 massiv[i]^.Symbol:=i;

 massiv[i]^.SymbolStat:=0;

 massiv[i]^.left:=nil;

 massiv[i]^.right:=nil;

 Application.ProcessMessages;//Высвобождаем ресурсы

 //чтобы приложение не казалось зависшим, иначе все ресуры процессора

 //будт задействованы на обработку кода приложения

 End;

 End;

// процедура для для вычисления частот появления

// i-го элемента в сжимаемом файле. Вызывается из

 Procedure TStat.Inc(i: Byte);

 Begin

 massiv[i]^.SymbolStat:=massiv[i]^.SymbolStat+1;

 End;

Type

 //объект включающий в себя:

 //имя и путь к архивируемому файлу

 //размер архивируемого файла

 //массив статистики частот байтов

 //дерево частот байтов

 //функцию генерации по имени файла имени архива

 //функцию генерации по имени архива имени исходного файла

 //функцию для определения размера файла без заголовка

 //иными словами возвращающую смещение в архивном файле

 //откуда начинаются сжатые данные

 File\_=Object

 Name: String;

 Size: Integer;

 Stat: TStat;

 Tree: TByte;

 Function ArcName: String;

 Function DeArcName: String;

 Function FileSizeWOHead: Integer;

 End;

 // генерация по имени файла имени архива

Function File\_.ArcName: String;

 Var

 i: Integer;

 name\_: String;

 Const

 PostFix=ArchExt;

 Begin

 name\_:=name;

 i:=Length(Name\_);

 While (i>0) And not(Name\_[i] in ['/','\','.']) Do

 Begin

 Dec(i);

 Application.ProcessMessages;

 End;

 If (i=0) or (Name\_[i] in ['/','\'])

 Then

 ArcName:=Name\_+'.'+PostFix

 Else

 If Name\_[i]='.'

 Then

 Begin

 Name\_[i]:='.';

 //Name\_[i]:='!';

 ArcName:=Name\_+'.'+PostFix;

 End;

 End;

 // генерация по имени архива имени исходного файла

Function File\_.DeArcName: String;

 Var

 i: Integer;

 Name\_: String;

 Begin

 Name\_:=Name;

 if pos(dot+ArchExt,Name\_)=0

 Then

 Begin

 ShowMessage('Неправильное имя архива,'#13#10'оно должно заканчиваться на ".'+ArchExt+'"');

 Application.Terminate;

 End

 Else

 Begin

 i:=Length(Name\_);

 While (i>0) And (Name\_[i]<>'!') Do

 Begin

 Dec(i);

 Application.ProcessMessages;

 End;

 If i=0

 Then

 Begin

 Name\_:=copy(Name\_,1,pos(dot+ArchExt,Name\_)-1);

 If Name\_=''

 Then

 Begin

 ShowMessage('Неправильное имя архива');

 Application.Terminate;

 End

 Else

 DeArcName:=Name\_;

 End

 Else

 Begin

 Name\_[i]:='.';

 Delete(Name\_,pos(dot+ArchExt,Name\_),4);

 DeArcName:=Name\_;

 End;

 End;

 End;

Function File\_.FileSizeWOHead: Integer;

 Begin

 FileSizeWOHead:=FileSize(FileToRead)-4-1-

 (Stat.CountByte+1)\*5;

 //размер исходного файла записывается в 4 байтах

 //количество оригинальных байт записывается в 1байте

 //количество байтов со статистикой - величина массива

 End;

 //процедура сортировки массива с байтами (сортировка производится

 //по убыванию частоты байта

procedure SortMassiv(var a: BytesWithStat; length\_mass: byte);

 var

 i,j: Byte;

 b: TByte;

 Begin

 if length\_mass<>0

 Then

 for j:=0 to length\_mass-1 do

 Begin

 for i:=0 to length\_mass-1 do

 Begin

 If a[i]^.SymbolStat < a[i+1]^.SymbolStat

 Then

 Begin

 b:=a[i]; a[i]:=a[i+1]; a[i+1]:=b;

 End;

 Application.ProcessMessages;

 End;

 Application.ProcessMessages;

 End;

 End;

 {Процедура построения древа частот Shennon}

procedure CreateTree(var Root: TByte;massiv: BytesWithStat;

 last: byte);

//процедуа деления группы

procedure DivGroup(i1, i2: byte);

{процедура создания кодовых слов. Вызывается после того как отработала процедура деления массива на группы. В полученном первом массиве мы ко всем одовым словам добавляем символ '0' во втором символ единицы}

procedure CreateCodWord(i1, i2: byte;Value:string);

 var

 i:integer;

begin

 for i:=i1 to i2 do

 massiv[i]^.CodWord:=massiv[i]^.CodWord+Value;

end;

//Процедуа деления массива

var

k, i : byte;

c, oldc, s, g1, g2 :Single;

begin

 //Пограничное условие, чтобы рекурсия у нас

 // не была бесконечной

 if (i1<i2) then

 begin

 s := 0;

 for i := i1 to i2 do

 s := s + massiv[i]^.SymbolStat;//Суммируем статистику частот

//появления символов в файле

 k := i1; //Далее инициализируем переменные

 g1 := 0;

 g2 := s;

 c := g2 - g1;

{Алгоритм: Переменные i1 и i2 это индексы начального и соответственно конечного элемента массива в k будем вырабатывать индекс пограничного элемента массива по которому мы его будем делить. с переменная в кторой будет хранится разность между g2 и g1. Потребуется для определения k. Сначала суммируем статистику появления символов в файле (Она как ни странно будет равна размеру файла =: т.е. количеству байт в нём)). Далее инициализируем переменные.

Затем цикл в котором происходит следующее к g1 нулевая статистика прибавляем статстику massiv[k] элемента массива massiv[k], а из g2 вычитаем ту же статистику. Далее oldc:=c это нам надо для определения дошли мы до значения k где статистика обойх частей массива равна. c := abs(g2-g1) именно по модулю иначе у нас не выполнится условие (c >= oldc) в том случае когда (g2<g1). Далее проверяется условие c > oldc, если оно верно то мы уменьшаем k на единицу, если не то оставляем k какое есть это и будет значение элемента в котором сумм статистик масивов примерно равны. Далее просто рекурсивно вызываем Эту же процедуру пока массивы полностью не разделятся на одиночные элементы или листья }

repeat

 g1 := g1 + massiv[k]^.SymbolStat;

 g2 := g2 - massiv[k]^.SymbolStat;

 oldc := c;

 c := abs(g2-g1);

 Inc(k);

 until (c >= oldc) or (k = i2);

 if c > oldc then

 begin

 Dec(k); //вырабатываем значение k2

 end;

 CreateCodWord(i1, k-1,'0'); //Заполняем первый массив

 //элементами

 CreateCodWord(k, i2,'1'); //Заполняем второй массив

 //элементами

 DivGroup(i1, k-1);//снова вызываем процедуру

 //деления массива (первой части)

 DivGroup(k, i2);// вызываем процедуру

 end;

end;

begin

DivGroup(0,last);

end;

var

 //экземпляр объекта для текущего сжимаемого файла

 MainFile: file\_;

//процедура для полного анализа частот байтов встречающихся хотя бы

//один раз в исходном файле

procedure StatFile(Fname: String);

var

 f: file; //переменная типа file в неё будем писать

 i,j: Integer;

 buf: Array [1..count] of Byte;//массив=4кБ содержащий в

 //себе часть архивируемого файла до 4кБ делается это для ускорения

 //работы програмы

 countbuf, lastbuf: Integer;//countbuf переменная которая показывает

 //какое целое количество буферов=4кБ содержится в исходном файле

 //для анализа частот символов встречающих в исходнлм файле

 //lastbuf остаток байт которые неободимо будет проанализировать

Begin

 AssignFile(f,fname);//связываем файловую переменню f

 //с архивируемым файлом

 Try

 Reset(f,1);//открываем файл

 MainFile.Stat.create;//вызываем метод инициализации объекта

 //для архивируемого файла

 MainFile.Size:=FileSize(f);//метод определения размера

 // архивируемого файла

 ///////////////////////

 countbuf:=FileSize(f) div count;//столько целых буферов

 //по 4096 байт содержится в исходном файле

 lastbuf:=FileSize(f) mod count; // остаток (последий буфер)разница

 //в байтах до 4096

 ////////////

 For i:=1 to countbuf do

 Begin

 BlockRead(f,buf,count);

 for j:=1 to count do

 Begin

 MainFile.Stat.inc(buf[j]);

 Application.ProcessMessages;

 End;

 Application.ProcessMessages;

 End;

 /////////////

 If lastbuf<>0 //просчитываем статистику для оставшихся

 //байт

 Then

 Begin

 BlockRead(f,buf,lastbuf);

 for j:=1 to lastbuf do

 Begin

 MainFile.Stat.inc(buf[j]);

 Application.ProcessMessages;

 End;

 Application.ProcessMessages;

 End;

 CloseFile(f);

 Except

 ShowMessage('ошибка доступа к файлу!')

 End;

End;

//процедура записи сжатого потока битов в архив

Procedure WriteInFile(var buffer: String);

var

 i,j: Integer;

 k: Byte;

 buf: Array[1..2\*count] of byte;

Begin

 i:=Length(buffer) div 8; // узнаем сколько получится

 //байт в каждой последовательности

 //////////////////////////

 For j:=1 to i do // работаем с байтами

 Begin

 buf[j]:=0;// обнуляем тот элемент мссива в

 //который будем писать

 ///////////////////////////

 For k:=1 to 8 do //работаем с битами

{находим в строке тот элемент который будем записывать в виде последовательности бит (будем просматривать с (j-1) элемента строки buffer восемь элментов за ним тем самым сформируется строка из восьми '0' и '1'. Эту строку мы будем преобразовывать в байт, который должен будет содержать такуюже последовательность бит)}

 Begin

 If buffer[(j-1)\*8+k]='1'

 Then

{Преобразование будем производить с помощью операции битового сдвига влево shl и логической опереоции или (or). Делается это так поверяется условие buffer[(j-1)\*8+k]='1' если в выделенной строке из восьми символов (мы просматриваем её по циклу от первого элемента до восьмого), элемент, индекс которого равен счётчику цикла к, равен единице, то к соответствующему биту (номер которого в байте равен переменной цикла к) будет применена операция or (0 or 1=1) т.е. это бит примет значение 1. Если в строке будет ноль то и соответствующий бит будет равен нулю. (нам его не требуется устанавливать т.к. в начале работы с каждым байтом мы его обнуляем)}

 buf[j]:=buf[j] or (1 shl (8-k));

 Application.ProcessMessages;

 End;

 Application.ProcessMessages;

 End;

 BlockWrite(FileToWrite,buf,i);

 Delete(buffer,1,i\*8);

End;

//процедура для окончательной записи остаточной цепочки битов в архив

Procedure WriteInFile\_(var buffer: String);

var

 a,k: byte;

Begin

 WriteInFile(buffer);

 If length(buffer)>=8

 Then

 ShowMessage('ошибка в вычислении буфера')

 Else

 If Length(buffer)<>0

 Then

 Begin

 a:=$FF;

 for k:=1 to Length(buffer) do

 If buffer[k]='0'

 Then

 a:=a xor (1 shl (8-k));

 BlockWrite(FileToWrite,a,1);

 End;

End;

Type

 Integer\_=Array [1..4] of Byte;

//перевод целого числа в массив из четырех байт.

Procedure IntegerToByte(i: Integer; var mass: Integer\_);

var

 a: Integer;

 b: ^Integer\_;

Begin

 b:=@a;

 a:=i;

 mass:=b^;

End;

//перевод массива из четырех байт в целое число.

Procedure ByteToInteger(mass: Integer\_; var i: Integer);

var

 a: ^Integer;

 b: Integer\_;

Begin

 a:=@b;

 b:=mass;

 i:=a^;

End;

//процедура создания заголовка архива

Procedure CreateHead;

var

 b: Integer\_;

 //a: Integer;

 i: Byte;

Begin

 //Размер несжатого файла

 IntegerToByte(MainFile.Size,b);

 BlockWrite(FileToWrite,b,4);

 //Количество оригинальных байт

 BlockWrite(FileToWrite,MainFile.Stat.CountByte,1);

//Байты со статистикой

 For i:=0 to MainFile.Stat.CountByte do

 Begin

 BlockWrite(FileToWrite,MainFile.Stat.massiv[i]^.Symbol,1);

 IntegerToByte(MainFile.Stat.massiv[i]^.SymbolStat,b);

 BlockWrite(FileToWrite,b,4);

 End;

End;

const

 MaxCount=4096;

type

 buffer\_=object

 ArrOfByte: Array [1..MaxCount] of Byte;

 ByteCount: Integer;

 GeneralCount: Integer;

 Procedure CreateBuf;

 Procedure InsertByte(a: Byte);

 Procedure FlushBuf;

 End;

 /////////////////////////////

 Procedure buffer\_.CreateBuf;

 Begin

 ByteCount:=0;

 GeneralCount:=0;

 End;

 ////////////////////////////////////////

Procedure buffer\_.InsertByte(a: Byte); //в а передаём уже

// раскодированный символ котрый надо записать в файл

 Begin

 if GeneralCount<MainFile.Size

 Then

 Begin

 inc(ByteCount);

 inc(GeneralCount);

 ArrOfByte[ByteCount]:=a;

 //////////////////////////

 if ByteCount=MaxCount

 Then

 Begin

 BlockWrite(FileToWrite,ArrOfByte,ByteCount);

 ByteCount:=0;

 End;

 End;

 End;

 ////////////////////////////

Procedure Buffer\_.FlushBuf; //сброс буфера

 Begin

 If ByteCount<>0

 Then

 BlockWrite(FileToWrite,ArrOfByte,ByteCount);

 End;

//создание деархивированного файла

Procedure CreateDeArc;

var

 i,j: Integer;

 k: Byte;

 //////////////

 Buf: Array [1..Count] of Byte;

 CountBuf, LastBuf: Integer;

 MainBuffer: buffer\_;

 BufSearch:string;

{Процедура поиска символа, кторый соотвествуеткодовому слову которое передаётся вызывающей функцией как параметр.

Алгоритм: Вызывающая ф-ия CreateDeArc вырабатывает значение символа из разархивируемого файла и вызывает ф-ию SearchSymbol (Str:string); с параметром Str в котором находится выработанны символ. Ф-ия SearchSymbol прибавляет этот символ к строке Str1 в которой формируется кодовое слово}

Procedure SearchSymbol (Str:string);

var

v:integer;

SearchStr:String;//вспомогательная переменная в которую

//загоняются кодовые слова для сравнения их с Str1

a:byte;//переменная в которой будет находится найденный

//символ

begin

 Str1:=Str1+Str;//растим кодовое слово

 For v:=0 to MainFile.Stat.CountByte do

 begin //производим поиск в массиве

 SearchStr:=MainFile.Stat.massiv[v]^.CodWord ;

 If (SearchStr=Str1) Then

 begin

//если нашли то в а загоняем значение символа

 a:=MainFile.Stat.massiv[v]^.Symbol;

//вызываем процедуру записи символа

 MainBuffer.InsertByte(a);

//обнуляем строковую переменную

 Str1:='';

//выходим из цикла

 Break;

 end;

 end;

end;

Begin

 BufSearch:='';{переменная в которой хранится выработанный символ, который будет передаватся в процедуру SearchSymbol}

 CountBuf:=MainFile.FileSizeWOHead div count;

 LastBuf:=MainFile.FileSizeWOHead mod count;

 MainBuffer.CreateBuf;

 For i:=1 to CountBuf do

 Begin

 BlockRead(FileToRead,buf,count);

 for j:=1 to Count do

 Begin

{Выделяем байт в массиве. По циклу от 1 до 8 просматриваем значения его бит c 8 до 1. Для этого используется операция битового сдвига влево shl и логиеская операция and.

В цикле всё происходит следующим образом: Сначала просматривается старший бит (8-к)=7 и производится логическая операция and, если бит равен 1 то (1 and 1)=1 и в BufSearch:='1', если же бит равен 0 и (0 and 1)=0 и в BufSearch:='1' }

 for k:=1 to 8 do

 Begin

 If ((Buf[j] and (1 shl (8-k)))<>0 ) Then

 begin

 BufSearch:='1';

//вызываем процедуру SearchSymbol

 SearchSymbol (BufSearch);

//обнуляем поисковую переменную

 BufSearch:='';

 end

 Else

 begin

 BufSearch:=BufSearch+'0';

 SearchSymbol (BufSearch);

 BufSearch:='';

 Application.ProcessMessages;

 End;

 Application.ProcessMessages;

 End;

 Application.ProcessMessages;

 End;

 Application.ProcessMessages;

 End;

 If LastBuf<>0

 Then //аналогично вышесказанному

 Begin

 BlockRead(FileToRead,Buf,LastBuf);

 for j:=1 to LastBuf do

 Begin

 for k:=1 to 8 do

 Begin

 If ((Buf[j] and (1 shl (8-k)))<>0 )

 Then

 begin

 BufSearch:=BufSearch+'1';

 SearchSymbol (BufSearch);

 BufSearch:='';

 end

 Else

 begin

 BufSearch:=BufSearch+'0';

 SearchSymbol (BufSearch);

 BufSearch:='';

 end;

 Application.ProcessMessages;

 End;

 Application.ProcessMessages;

 End;

 End;

 MainBuffer.FlushBuf;

End;

//процедура чтения заголовка архива

Procedure ReadHead;

var

 b: Integer\_;

 SymbolSt: Integer;

 count\_, SymbolId, i: Byte;

Begin

 try

 //узнаем исходный размер файла

 BlockRead(FileToRead,b,4);

 ByteToInteger(b,MainFile.size);

 //узнаем количество оригинальных байтов

 BlockRead(FileToRead,count\_,1);

 {}{}{}

 MainFile.Stat.create;

 MainFile.Stat.CountByte:=count\_;

 //загоняем частоты в массив

 for i:=0 to MainFile.Stat.CountByte do

 Begin

 BlockRead(FileToRead,SymbolId,1);

 MainFile.Stat.massiv[i]^.Symbol:=SymbolId;

 BlockRead(FileToRead,b,4);

 ByteToInteger(b,SymbolSt);

 MainFile.Stat.massiv[i]^.SymbolStat:=SymbolSt;

 End;

 CreateTree(MainFile.Tree,MainFile.stat.massiv,MainFile.Stat.CountByte);

 /////////////

 CreateDeArc;

 //////////////

 // DeleteTree(MainFile.Tree);

 except

 ShowMessage('архив испорчен!');

 End;

End;

//процедура извлечения архива

Procedure ExtractFile;

Begin

 AssignFile(FileToRead,MainFile.Name);

 AssignFile(FileToWrite,MainFile.DeArcName);

 try

 Reset(FileToRead,1);

 Rewrite(FileToWrite,1);

//процедура чтения шапки файла

 ReadHead;

 Closefile(FileToRead);

 Closefile(FileToWrite);

 Except

 ShowMessage('Ошибка распаковки файла');

 End;

End;

//вспомогательная процедура для создания архива

Procedure CreateArchiv;

var

 buffer: String;

 ArrOfStr: Array [0..255] of String;

 i,j: Integer;

 //////////////

 buf: Array [1..count] of Byte;

 CountBuf, LastBuf: Integer;

Begin

 Application.ProcessMessages;

 AssignFile(FileToRead,MainFile.Name);

 AssignFile(FileToWrite,MainFile.ArcName);

 Try

 Reset(FileToRead,1);

 Rewrite(FileToWrite,1);

 For i:=0 to 255 Do ArrOfStr[i]:='';

 For i:=0 to MainFile.Stat.CountByte do

 Begin

 ArrOfStr[MainFile.Stat.massiv[i]^.Symbol]:=

 MainFile.Stat.massiv[i]^.CodWord;

 Application.ProcessMessages;

 End;

 CountBuf:=MainFile.Size div Count;

 LastBuf:=MainFile.Size mod Count;

 Buffer:='';

 /////////////

 CreateHead;

 /////////////

 for i:=1 to countbuf do

 Begin

 BlockRead(FileToRead,buf,Count);

 //////////////////////

 For j:=1 to count do

 Begin

 buffer:=buffer+ArrOfStr[buf[j]];

 If Length(buffer)>8\*count

 Then

 WriteInFile(buffer);

 Application.ProcessMessages;

 End;

 End;

 If lastbuf<>0

 Then

 Begin

 BlockRead(FileToRead,buf,LastBuf);

 For j:=1 to lastbuf do

 Begin

 buffer:=buffer+ArrOfStr[buf[j]];

 If Length(buffer)>8\*count

 Then

 WriteInFile(buffer);

 Application.ProcessMessages;

 End;

 End;

 WriteInFile\_(buffer);

 CloseFile(FileToRead);

 CloseFile(FileToWrite);

 Except

 ShowMessage('Ошибка создания архива');

 End;

End;

//главная процедура для создания архивного файла

Procedure CreateFile;

var

 i: Byte;

Begin

 With MainFile do

 Begin

 {сортировка массива байтов с частотами}

 SortMassiv(Stat.massiv,stat.CountByte);

 {поиск числа задействованных байтов из таблицы возмжных символов. В count\_byte будем хранить количество этох самых байт }

 i:=0;//обнуляем счётчик

 While (i<Stat.CountByte) //до тех пор пока счётчик

 //меньше количества задействовнных байт CountByte

 //и статистика байта (частота появления в файле)

 //не равна нулю делаем

 and (Stat.massiv[i]^.SymbolStat<>0) do

 Begin

 Inc(i); //увеличиваем счётчик на единицу

 End;

 //////////////////////

 If Stat.massiv[i]^.SymbolStat=0 //если дошли до символа

 //с нулевой встречаемостью в файле то

 Then

 Dec(i); //уменьшаем счётчик на единицу тоесть возвращаемся

 //назад это будет последний элемент

 //////////////////////

 Stat.CountByte:=i;//присваиваем значение счётчика

 //count\_byte. Это означает что в архивируемом файле

 //используется такое количество из 256 возможных

 //символов. Будет исползоватся для построения древа частот

 {создание дерева частот.

Передаём в процедуру начальные параметры Tree=nil-эта переменная будет содержать после работы процедуры древо ,Stat.massiv-массив с символами и соответствующей им статистикой,а так же указанием на правое и левой дерево, Stat. CountByte-количество используемых символов в архивирумом файле }

 CreateTree(Tree,Stat.massiv,Stat.CountByte);

 //пишем сам файл

 CreateArchiv;

 //Удаляем уже ненужное дерево

 //DeleteTree(Tree);

 //Инициализируем статистику файла

 MainFile.Stat.Create;

 End;

End;

procedure RunEncodeShan(FileName\_: string);

begin

 MainFile.Name:=FileName\_;//передаём имя

 //архивируемого файла в программу

 StatFile(MainFile.Name); //запускем процедуру создания

 //статистики (частоты появления того или иного символа)для файла

 CreateFile; //вызов процедуры созданя архивного файла

end;

procedure RunDecodeShan(FileName\_: string);

begin

 MainFile.name:=FileName\_;//передаём имя

 //архивируемого файла в программу

 ExtractFile;//Вызываем процедуру извлечения архива

end;

end.

Приложение 2.

Реализация на Delphi алгоритма сжатия Хафмана

unit Haffman;

interface

Uses

 Forms,ComCtrls, Dialogs;

const

 Count=4096;

 ArchExt='haf';

 dot='.';

//две файловые переменные для чтения исходного файла и для

//записи архива

var

 FileToRead,FileToWrite: File;

 ProgressBar1:TProgressBar;

// Процедуры для работы с файлом

// Первая - кодирование файла

procedure RunEncodeHaff(FileName\_: string);

// Вторая - декодирование файла

procedure RunDecodeHaff(FileName\_: string);

implementation

Type

{тип элемета для динамической обработки статистики символов

встречающихся в файле}

 TByte=^PByte;

 PByte=Record

 //Символ (один из символв ASCII)

 Symbol: Byte;

 //частота появления символа в сжимаемом файле

 SymbolStat: Integer;

 //последовательность битов, в которые преобразуется текущий

 //элемент после работы древа (Кодовое слово) (в виде строки из "0" и "1")

 CodWord: String;

 //ссылки на левое и правое поддеревья (ветки)

 left, right: TByte;

 End;

{массив из символов со статистикой , т.е. частотой появления их в архивируемом файле}

 BytesWithStat = Array [0..255] of TByte;

 {объект, включающий в себя:

 массив элементов содержащий в себе количество элементов,

 встречающихся в файле хотя бы один раз

 процедура инициализации объекта

 процедура для увеличения частоты i-го элемента}

 TStat =Object

 massiv: BytesWithStat;

 CountByte: byte;

 Procedure Create;//процедура инициализации обьекта

 Procedure Inc(i: Byte);

 End;

// процедура инициализации объекта вызывается из процедуры StatFile

 Procedure TStat.Create; //(291)

 var

 i: Byte;

 Begin //создаём массив симолв (ASCII), обнуляем статистику

 //и ставим указатели в положение не определено

 CountByte:=255;

 For i:=0 to CountByte do

 Begin

 New(massiv[i]);//создаём динамическую переменную

 //и устанавливаем указатель на неё

 massiv[i]^.Symbol:=i;

 massiv[i]^.SymbolStat:=0;

 massiv[i]^.left:=nil;

 massiv[i]^.right:=nil;

 Application.ProcessMessages;//Высвобождаем ресурсы

 //чтобы приложение не казалось зависшим, иначе все ресуры процессора

 //будут задействованы на обработку кода приложения

 End;

 End;

{процедура для вычисления частот появления

i-го элемента в сжимаемом файле вызывается строка(310)}

 Procedure TStat.Inc(i: Byte);

 Begin //увеличиваем значение статистики символа [i] наединицу

 massiv[i]^.SymbolStat:=massiv[i]^.SymbolStat+1;

 End;

Type

 //объект включающий в себя:

 //имя и путь к архивируемому файлу

 //размер архивируемого файла

 //массив статистики частот байтов

 //дерево частот байтов

 //функцию генерации по имени файла имени архива

 //функцию генерации по имени архива имени исходного файла

 //функцию для определения размера файла без заголовка

 //иными словами возвращающую смещение в архивном файле

 //откуда начинаются сжатые данные

 File\_=Object

 Name: String;

 Size: Integer;

 Stat: TStat;

 Tree: TByte;

 Function ArcName: String;

 Function DeArcName: String;

 Function FileSizeWOHead: Integer;

 End;

 // генерация по имени файла имени архива

Function File\_.ArcName: String;

 Var

 i: Integer;

 name\_: String;

 Const

 PostFix=ArchExt;

 Begin

 name\_:=name;

 i:=Length(Name\_);

 While (i>0) And not(Name\_[i] in ['/','\','.']) Do

 Begin

 Dec(i);

 Application.ProcessMessages;

 End;

 If (i=0) or (Name\_[i] in ['/','\'])

 Then

 ArcName:=Name\_+'.'+PostFix

 Else

 If Name\_[i]='.'

 Then

 Begin

 Name\_[i]:='.';

 // Name\_[i]:='!';

 ArcName:=Name\_+'.'+PostFix;

 End;

 End;

 // генерация по имени архива имени исходного файла

Function File\_.DeArcName: String;

 Var

 i: Integer;

 Name\_: String;

 Begin

 Name\_:=Name;

 if pos(dot+ArchExt,Name\_)=0

 Then

 Begin

 ShowMessage('Неправильное имя архива,'#13#10'оно должно заканчиваться на ".'+ArchExt+'"');

 Application.Terminate;

 End

 Else

 Begin

 i:=Length(Name\_);

 While (i>0) And (Name\_[i]<>'.') Do //до тех пор пока

 //не встритится '.' !

 Begin

 Dec(i); //уменьшаем счётчик на единицу

 Application.ProcessMessages;

 End;

 If i=0

 Then

 Begin

 Name\_:=copy(Name\_,1,pos(dot+ArchExt,Name\_)-1);

 If Name\_=''

 Then

 Begin

 ShowMessage('Неправильное имя архива');

 Application.Terminate;

 End

 Else

 DeArcName:=Name\_;

 End

 Else

 Begin

 Name\_[i]:='.';

 Delete(Name\_,pos(dot+ArchExt,Name\_),4);

 DeArcName:=Name\_;

 End;

 End;

 End;

Function File\_.FileSizeWOHead: Integer;

 Begin

 FileSizeWOHead:=FileSize(FileToRead)-4-1-

 (Stat.CountByte+1)\*5;

 //размер исходного файла записывается в 4 байтах

 //количество оригинальных байт записывается в 1байте

 //количество байтов со статистикой - величина массива

 End;

 //процедура сортировки массива с байтами (сортировка производится

 //по убыванию частоты байта (743)

procedure SortMassiv(var a: BytesWithStat; LengthOfMass: byte);

 var

 i,j: Byte; //счётчики циклов

 b: TByte;

 Begin //сортировка перестановкой

 if LengthOfMass<>0

 Then

 for j:=0 to LengthOfMass-1 do

 Begin

 for i:=0 to LengthOfMass-1 do

 Begin

 If a[i]^.SymbolStat < a[i+1]^.SymbolStat

 Then

 Begin

 b:=a[i]; a[i]:=a[i+1]; a[i+1]:=b;

 End;

 Application.ProcessMessages;

 End;

 Application.ProcessMessages;

 End;

 End;

 //процедура удаления динамической структуры частотного дерева

 //из памяти

 Procedure DeleteTree(Root: TByte);

 Begin

 Application.ProcessMessages;

 If Root<>nil

 Then

 Begin

 DeleteTree(Root^.left);

 DeleteTree(Root^.right);

 Dispose(Root);

 Root:=nil;

 End;

 End;

 //создание дерева частот для архивируемого файла Haffman (777)

 Procedure CreateTree(var Root: TByte; massiv: BytesWithStat;

 last: byte);

var

Node: TByte;//узел

Begin

 //sort\_mass(massiv, last);

 If last<>0 //если не 0 то начинаем строить дерево

 Then

 Begin

 SortMassiv(massiv, last);//сортируем по убыванию

 //частоты появления символа

 new(Node);//создаёмо новый узел

 //присваиваем ему вес двух самых лёгких эементов

 //т.е. складываем статистику этих элементов

 Node^.SymbolStat:=massiv[last-1]^.SymbolStat + massiv[last]^.SymbolStat;

 Node^.left:=massiv[last-1];//от узла делаем ссылку на левую

 Node^.right:=massiv[last];//и правую ветки

 massiv[last-1]:=Node;// удаляем два последних элемента

 //из массива на место предпоследнего из них ставим

 //сформированный узел

 ///////////////// проверяем не достигли ли корня

 if last=1//если =1 то да

 Then

 Begin

 Root:=Node; //устанавливаем корневой узел

 End

 Else

 Begin

 CreateTree(Root,massiv,last-1);//если нет то строим

 //древо дальше

 End;

 End

 Else//если значение last в самом начале =0 т.е. файл

 //содержит один и тот же символ (если файл состоит или

 //из одного байта или из чередования одного итогоже символа)

 Root:=massiv[last];//то вершина дерева будет от last

 Application.ProcessMessages;

End;

var

 //экземпляр объекта для текущего сжимаемого файла

 MainFile: file\_;

//процедура для полного анализа частот байтов встречающихся хотя бы

//один раз в исходном файле

procedure StatFile(fname: String);

var

 f: file; //переменная типа file в неё будем писать

 i,j: Integer;

 buf: Array [1..count] of Byte;//массив=4кБ содержащий в

 //себе часть архивируемого файла до 4кБ делается это для ускорения

 //работы програмы

 countbuf, lastbuf: Integer;//countbuf переменная которая показывает

 //какое целое количество буферов=4кБ содержится в исходном файле

 //для анализа частот символов встречающих в исходнлм файле

 //lastbuf остаток байт которые неободимо будет проанализировать

Begin

 AssignFile(f,fname);//связываем файловую переменню f

 //с архивируемым файлом

 Try //на всякий случай

 Reset(f,1);//открываем файл для чтения

 MainFile.Stat.create;//вызываем метод инициализации объекта

 //для архивируемого файла (58)

 MainFile.Size:=FileSize(f);//метод определения размера

 // архивируемого файла. Стандартная функция FileSize

 //возвращает начение в байтах

 ///////////////////////

 countbuf:=FileSize(f) div count;//столько целых буферов

 //по 4096 байт содержится в исходном файле

 lastbuf:=FileSize(f) mod count; // остаток от целочисленного

 // деления=(последий буфер)разница в байтах до 4096

 //////////// Создаём статистику для каждого символа в файле

 For i:=1 to countbuf do //сначала прогоняем все целые буферы(на )

 Begin

 BlockRead(f,buf,count);

 for j:=1 to count do

 Begin //мы берём из буфера элемент от 1 до 4096 и с этими

 //параметрами вызываем функцию Stat.inc(элемент)

 //он же будет являтся и указателем на самого себя в

 //в массиве символов там мы просто увеличиваем значение

 //SymbolStat(частоты появления) на единицу

 MainFile.Stat.inc(buf[j]);//(строка 80)

 Application.ProcessMessages;

 End;

 Application.ProcessMessages;

 End;

 /////////////

 If lastbuf<>0 //далее просчитываем статистику для оставшихся

 //байт

 Then

 Begin

 BlockRead(f,buf,lastbuf);

 for j:=1 to lastbuf do

 Begin

 MainFile.Stat.inc(buf[j]);//(80)

 Application.ProcessMessages;

 End;

 Application.ProcessMessages;

 End;

 CloseFile(f);//Закрываем файл

 Except //Если чтото не так то выводим сообщение

 ShowMessage('ошибка доступа к файлу!')

 End;

End;

{функция поиска в дереве Found(Tree: TByte; i: byte): Boolean;

 параметры Tree:корень дерева или его узел, i:символ кодовое слово которого ищем; возвращает булево значение в функцию HSymbolToCodWord.

Алгоритм работы:

 функция HSymbolToCodWord вызывает функцию Found(Tree^.left,i) т.е c параметром поиска в левой ветке дерева начиная от корня. Функция Found будет рекурсивно вызывать сама себя двигаясь по узлам дерева пока не дойдёт до искомого символа. Если там окажется искомый символ то Found вернёт true и в HSymbolToCodWord запишется первый нолик если Found(Tree^.left,i):true или единичка если Found(Tree^.right,i):true далее HSymbolToCodWord вызывает Found, но уже в параметрах указывается не корень, а седующий за ним узел, находящийся слева или справа, в зависимости от пред идущего результата поиска (в какой ветви от корня был найден символ(если слева его не было зачем там искать)) так будет продолжатся до тех пор пока HSymbolToCodWord не будет достигнут символ т.е. параметры функции будут Tree=узлу где находится символ (т.е. указатели на левую и правую ветви =nil)далее при выполнении функции она выработает значение для Tree=nil. Далее Found вернёт значение

Tree= узлу где нахоится искомый символ, выработает значение Found=True  и вернётся в вызывающую функцию HSymbolToCodWord где в значение

HSymbolToCodWord в конец запишется '+'-означающий что кодовое слово найдено. Псле этого HSymbolToCodWord вернёт в вызвавшую её функциюSymbolToCodWord значение кодового слова+'+'на конце где произойдё проверка и символ '+' будет удалён, в вызывающий метод Stat.massiv[i]^.CodWord будет возвращено значение кодового слова}Function Found(Tree: TByte; i: byte): Boolean;

Begin

Application.ProcessMessages;

 if (Tree=nil)//если древо nil то

 Then

 Found:=False //функция прекращает работу

 Else //иначе

 Begin //если указатель на левую часть древа или

 //на правую nil, и указатель на символ равен счётчику

 if ((Tree^.left=nil) or (Tree^.right=nil))

 and (Tree^.Symbol=i)

 Then

 Found:=True {то функция возвращает флаг, что найден символ

и прекращает работу и возвращает в вызвавшую её функцию }

 Else //иначе функция продолжает поиск от других узлов

 //т.е.рекурсивно вызывает сама себя с другими параметрами

 Found:=Found(Tree^.left, i) or Found(Tree^.right, i);

 End;

End;

//функция для определения строкового представления сжатой последовательности

//битов для исходного байта i

Function HSymbolToCodWord(Tree: TByte; i: Byte): String;

Begin

 Application.ProcessMessages;

 if (Tree=nil)

 Then

 HSymbolToCodWord:='+=='

 Else

 Begin

 if (Found(Tree^.left,i))//если символ находится в левой ветви

 //в зависимости от того что вернула Found

 Then //то в строку добавляем символ нуля и вызываем HSymbolToCodWord

 //от ниже лежащего левого узла

 HSymbolToCodWord:='0'+HSymbolToCodWord(Tree^.left,i)

 Else

 Begin

 if Found(Tree^.right,i)//если символ находится в правой ветви

 Then //то в строку добавляем символ единицы и вызываем HSymbolToCodWord

 //от ниже лежащего правого узла

 HSymbolToCodWord:='1'+HSymbolToCodWord(Tree^.right,i)

 Else //иначе

 Begin //если найден символ

 If (Tree^.left=nil) and (Tree^.right=nil)

 and (Tree^.Symbol=i)

 Then //HSymbolToCodWord //помечаем символ найден

 HSymbolToCodWord:='+'

 Else //иначе

 HSymbolToCodWord:=''; //символа нет

 End;

 End;

 End;

End;

//вспомогательная функция для определения Кодового слова

//сжатой последовательности битов для исходного байта i (с учетом

//того экстремального случая, когда исходный файл состоит всего из одного

//и того же символа)

Function SymbolToCodWord(Tree: TByte; i: Byte): String;

var

 s: String;

Begin //Вызыаем ф-ию поиска кодовых слов

 s:=HSymbolToCodWord(Tree, i);

 s:=s;

 If (s='+'){если функция HSymbolToCodWord вернула строку

 содержащую '+' т.е. исходный файл состоит из одного и того же

 символа то кодовому слову присваиваем строку из '0' }

 Then

 SymbolToCodWord:='0'

 Else {иначе уменьшаем строку на один символ т.е. убираем '+'

 признак того что символ найден}

 SymbolToCodWord:=Copy(s,1,length(s)-1);

End;

//процедура записи сжатого потока битов в архив

Procedure WriteInFile(var buffer: String);

var

 i,j: Integer;

 k: Byte;

 buf: Array[1..2\*count] of byte;

Begin

 i:=Length(buffer) div 8; // узнаем сколько получится

 //байт в каждой последовательности

 //////////////////////////

 For j:=1 to i do //работаем с байтами от превого элемента

 //массива до последнего

 Begin

 buf[j]:=0;//обнуляем тот элемент мссива в

 //который будем писать

 ///////////////////////////

 For k:=1 to 8 do//работаем с битами

 Begin

 If buffer[(j-1)\*8+k]='1'{находим в строке тот элементкоторый будем записывать в виде последовательности бит(будем просматривать с (j-1) элемента строки buffer восемь элментов за ним тем самым сформируется строка из восьми '0' и '1'. Эту строку мы будем преобразовывать в байт,который должен будет содержать такуюже последовательность бит)} Then {Преобразование будем производить с помощью операции битового сдвига влево shl и логической опереоции или (or). Делается это так поверяется условие buffer[(j-1)\*8+k]='1' если в выделенной строке из восьми символов (мы просматриваем её по циклу от первого элемента до восьмого), элемент, индекс которого равен счётчику цикла к, равен единице, то к соответствующему биту (номер которого в байте равен переменной цикла к) будет применена операция or (0 or 1=1) т.е. это бит примет значение 1. Если в строке будет ноль то и соответствующий бит будет равен нулю. (нам его не требуется устанавливать т.к. в начале работы с каждым байтом мы его обнуляем)}

 buf[j]:=buf[j] or (1 shl (8-k));

 Application.ProcessMessages;

 End;

 Application.ProcessMessages;

 End; //записываем в файл получивийся буфер

 BlockWrite(FileToWrite,buf,i);

 Delete(buffer,1,i\*8);//удаляем из входного буфера те элементы

 //которые уже записаны()

End;

//процедура для окончательной записи остаточной цепочки битов в архив

Procedure WriteInFile\_(var buffer: String);

var

 a,k: byte;

Begin

{Так как эту процедуру вызывает процедура которая передаёт в буфереотнюдь не один последний байт, то срау вызываем процедуруобычной записи в файл. После работы которой в buffer должнаостася последвательность из не более 8 символов. По этомумы производим проверку и если что то не так то выводим сообщение.

Иначе устанавливаем в переменной а все биты в 1 и далее производимследующие действия: Просматриваем по циклу всё что осталось вbuffer и если найдётся символ '0' то к сответтвующему биту переменной априменяем операцию xor (т.е. 1 xor 1 что даст 0) т.е. оответствующийбит установится в 0 все остальные биты переменной а останутся в том жесостоянии что и были. Оставшиеся биты будут единицами}

 WriteInFile(buffer);

 If length(buffer)>=8

 Then

 ShowMessage('ошибка в вычислении буфера')

 Else

 If Length(buffer)<>0

 Then

 Begin

 a:=$FF;

 for k:=1 to Length(buffer) do

 If buffer[k]='0'

 Then

 a:=a xor (1 shl (8-k));

 BlockWrite(FileToWrite,a,1);

 End;

End;

Type

 Integer\_=Array [1..4] of Byte;

//перевод числа типа Integer в массив из четырех байт.

Procedure IntegerToByte(i: Integer; var mass: Integer\_);

var

 a: Integer;

 b: ^Integer\_;

Begin

 b:=@a;// соединяем адресс переменной а с b

 a:=i;//в а перегоняем наше значение типа integer

 mass:=b^;{разименовываем b и соединяем результат с massв результате работы этого кода число типа Integerперейд в массив из 4 байт. Это требуется для того что ,бы мызапись в файл производим по байтно}

End;

//перевод массива из четырех байт в число типа Integer.

Procedure ByteToInteger(mass: Integer\_; var i: Integer);

var

 a: ^Integer;

 b: Integer\_;

Begin

 a:=@b;// соединяем адресс переменной b с а

 b:=mass;//b присваиваем значение mass

 i:=a^;{разименовываем а и соединяем результат с i

в результате работы этого кода массив из 4 байтперейд в число типа Integer. Это требуется для того что бы мымогли узнать наши значения типа Integer}

End;

//процедура создания заголовка архива

Procedure CreateHead;

var

 b: Integer\_;

 //a: Integer;

 i: Byte;

Begin

//Записываем размер несжатого файла

 IntegerToByte(MainFile.Size,b);

 BlockWrite(FileToWrite,b,4);

//Записываем количество оригинальных байт

 BlockWrite(FileToWrite,MainFile.Stat.CountByte,1);

{зисываем байты со статистикой (на каждую запись требуется по пять байт. Первый байт содержит сам символ далее идут 4 байта со статистикой (Intege занимает 4 байта)}

 For i:=0 to MainFile.Stat.CountByte do

 Begin

 BlockWrite(FileToWrite,MainFile.Stat.massiv[i]^.Symbol,1);

 IntegerToByte(MainFile.Stat.massiv[i]^.SymbolStat,b);

 BlockWrite(FileToWrite,b,4);

 End;

End;

const

 MaxCount=4096;

type

{buffer\_ это объект включающий в себя массив из байт ArrOfByteсчётчик байт ByteCount (необходим для учёта промежуточнойзапися разархивируемых байт в файл)и основной счётчик (необходимдля отслеживани какое количество байт должно быть разархивированокак только он стнет равным размеру сжимаемого файла то процессразархивирования первётся)}

 buffer\_=object

 ArrOfByte: Array [1..MaxCount] of Byte;

 ByteCount: Integer;

 GeneralCount: Integer;

 Procedure CreateBuf;//процедура инициализации

 Procedure InsertByte(a: Byte);//процедура вставки

//разархивированных байтов в файл

 Procedure FlushBuf;

 End;

 /////////////////////////////

 Procedure buffer\_.CreateBuf;

 Begin

 ByteCount:=0;//иициализируем переменные

 GeneralCount:=0;

 End;

 ////////////////////////////////////////

Procedure buffer\_.InsertByte(a: Byte);

{В переменной а мы передаём значение разархивированного байта,которое получили в вызывающей процедуре}

 Begin //до тех пор пока GeneralCount меньше

 //размера сжимаемого файла деаем

 if GeneralCount<MainFile.Size

 Then

 Begin

 inc(ByteCount); //увеличиваем соответствующие

 //счётчики на единицу

 inc(GeneralCount);

 ArrOfByte[ByteCount]:=a;//загоняем в массив ArrOfByte

//значение полученное в переменной а

 //////////////////////////

 if ByteCount=MaxCount //если ByteCount=MaxCount

//то записываем содержимое массива в разархивируемый файл

 Then

 Begin

 BlockWrite(FileToWrite,ArrOfByte,ByteCount);

 ByteCount:=0;

 //Form1.ProgressBar1.Position:=form1.ProgressBar1.Position+1;

 End;

 End;

 End;

 ////////////////////////////

Procedure Buffer\_.FlushBuf;

//Процедура записи остаточной цепочки байт

 Begin

 If ByteCount<>0

 Then

 BlockWrite(FileToWrite,ArrOfByte,ByteCount);

 End;

//создание деархивированного файла

Procedure CreateDeArc;

var

 i,j: Integer;

 k: Byte;

 //////////////

 Buf: Array [1..Count] of Byte;

 CountBuf, LastBuf: Integer;

 MainBuffer: buffer\_;

 CurrentPoint: TByte;

Begin

//определяем сколько целых буферов по 4 кбайт в сжатом

//файле без заголовка

 CountBuf:=MainFile.FileSizeWOHead div count;

//определяем сколько останеся байт не вошедших

//в целые буферы по 4 кбайт в сжатом файле без заголовка

 LastBuf:=MainFile.FileSizeWOHead mod count;

 MainBuffer.CreateBuf;//иициализируем переменные

 CurrentPoint:=MainFile.Tree;//присваиаем текущую

//позицию на корень дерева

//начинаем расаковку

 For i:=1 to CountBuf do

 Begin//считываем из сжатого файла данные в буфер

 BlockRead(FileToRead,buf,count);

 for j:=1 to Count do //по байтно начинаем

 //просматривать буфер

 Begin

 for k:=1 to 8 do//просматриваем биты от 1 до 8

 //выеленного байта

 Begin {Выделяем байт в массиве. По циклу от 1 до 8просматриваем значения его бит с 7 до 0. Для этого используетсяоперация битового сдвига влево shl и логиеская операция and.

В цикле всё происходит следующим образом: Сначала просматриваетсястарший бит (8-к)=1 и производится логическая операция and,если бит равен 1 то (1 and 1)=1 и программа установит текущую позицию поиска в дереве на правый узел, если же бит равен 0 то (0 and 1)=0 и программа установит текущую позицию поиска в дереве на левый узел. так будет продолжатся до тех пор пока не выполнится условие, которое ознчает нахождение искомого символа ((CurrentPoint^.left=nil) or (CurrentPoint^.right=nil))

После этого будет вызвана процедура вставки байта, после возвращения из которой мы текущую точку опять устанавливаем на корень}

 If (Buf[j] and (1 shl (8-k)))<>0

 Then

 CurrentPoint:=CurrentPoint^.right

 Else

 CurrentPoint:=CurrentPoint^.left;

 if (CurrentPoint^.left=nil) or (CurrentPoint^.right=nil)

 Then

 Begin

 MainBuffer.InsertByte(CurrentPoint^.Symbol);

 CurrentPoint:=MainFile.Tree;

 End;

 Application.ProcessMessages;

 End;

 Application.ProcessMessages;

 End;

 End;

 If LastBuf<>0

 Then

 Begin//работа этого блока программы аналогична предидущему

 BlockRead(FileToRead,Buf,LastBuf);

 for j:=1 to LastBuf do

 Begin

 for k:=1 to 8 do

 Begin

 If (Buf[j] and (1 shl (8-k)))<>0

 Then

 CurrentPoint:=CurrentPoint^.right

 Else

 CurrentPoint:=CurrentPoint^.left;

 if (CurrentPoint^.left=nil) or (CurrentPoint^.right=nil)

 Then

 Begin

 MainBuffer.InsertByte(CurrentPoint^.Symbol);

 CurrentPoint:=MainFile.Tree;

 End;

 Application.ProcessMessages;

 End;

 Application.ProcessMessages;

 End;

 End;

 MainBuffer.FlushBuf;

End;

//процедура чтения заголовка архива

Procedure ReadHead;

var

 b: Integer\_; // исходный размер файла

 SymbolSt: Integer;//статистика символа

 count\_, SymbolId, i: Byte;//SymbolId=Symbol просто чтобы

 // не путать глобальную переменную с локальной

Begin

 try

//узнаем исходный размер файла

 BlockRead(FileToRead,b,4);

 ByteToInteger(b,MainFile.size);

//узнаем количество оригинальных байтов

 BlockRead(FileToRead,count\_,1);

 {}{}{Вызываем процедуру инициализации объекта}

 MainFile.Stat.create;

 MainFile.Stat.CountByte:=count\_;

//загоняем частоты в массив

 for i:=0 to MainFile.Stat.CountByte do

 Begin

 BlockRead(FileToRead,SymbolId,1);

 MainFile.Stat.massiv[i]^.Symbol:=SymbolId;

 BlockRead(FileToRead,b,4);

 ByteToInteger(b,SymbolSt);

 MainFile.Stat.massiv[i]^.SymbolStat:=SymbolSt;

 End;

//вызываем процедуру создания дерева

 CreateTree(MainFile.Tree,MainFile.stat.massiv,MainFile.Stat.CountByte);

 /////////////

//Вызываем процедуру распаковки файла

 CreateDeArc;

 //////////////

//Вызываем процедуру уничтожения дерева

 DeleteTree(MainFile.Tree);

 except

 ShowMessage('архив испорчен!');

 End;

End;

//процедура извлечения архива

Procedure ExtractFile;

Begin

 AssignFile(FileToRead,MainFile.Name);

 //соединяем наш файл файловй переменой передэтим

 //вызываем метод получения имени разархивированого файла

 AssignFile(FileToWrite,MainFile.DeArcName);

 try

 Reset(FileToRead,1);

 Rewrite(FileToWrite,1);

//процедура чтения шапки файла

 ReadHead;

 Closefile(FileToRead);

 Closefile(FileToWrite);

 Except

 ShowMessage('Ошибка распаковки файла');

 End;

End;

//вспомогательная процедура для создания архива

Procedure CreateArchiv;

var

 buffer: String;//строка в которой будет формироватся

//последовательность из кодовых слов

 ArrOfStr: Array [0..255] of String;

 i,j: Integer;

 //////////////

 buf: Array [1..count] of Byte;//массив в который

//будем считывать данные из архивируемого файла

 CountBuf, LastBuf: Integer;

Begin

Application.ProcessMessages;

 AssignFile(FileToRead,MainFile.Name);

 AssignFile(FileToWrite,MainFile.ArcName);

 Try

 Reset(FileToRead,1);

 Rewrite(FileToWrite,1);

//Инициализируем массив строк в котором будут

//хранится кодовые слова

 For i:=0 to 255 Do ArrOfStr[i]:='';

//Загоням в массив строк кодовые слова соответсвующие

//своим символам

 For i:=0 to MainFile.Stat.CountByte do

 Begin

 ArrOfStr[MainFile.Stat.massiv[i]^.Symbol]:=

 MainFile.Stat.massiv[i]^.CodWord;

 Application.ProcessMessages;

 End;

//узнаём какое целое количество буферов по 4 кбайт будет содержатся в

//сжимаемом файле

 CountBuf:=MainFile.Size div Count;

//Сколько останется байт для записи не вошедших в ранее

//определённое значение CountBuf

 LastBuf:=MainFile.Size mod Count;

 Buffer:='';//обнуляем буфер

 /////////////

 CreateHead; //вызываем процедуру создания заголовка файла

 /////////////

 //фрмируем буфер кодовых слов

 for i:=1 to countbuf do

 Begin

//считываем из файла по 4 кбайт

 BlockRead(FileToRead,buf,Count);

 //////////////////////

 For j:=1 to count do

 Begin

//растим буфер из кодовых слов

 buffer:=buffer+ArrOfStr[buf[j]];

//если длина buffer превысит значеие 8\*4096 (это означает

//превысит размер выходного буфера размер которого 4096байт)

//мы вызываем процедуру записи в файл

 If Length(buffer)>8\*count

 Then

 WriteInFile(buffer);

 Application.ProcessMessages;

 End;

 // ProgressBar1.Position:=100 div countbuf;

 End;

//Запись оставшейся цепочки байт

 If lastbuf<>0

 Then

 Begin

//считываем в массив из файла оставшиеся байты

 BlockRead(FileToRead,buf,LastBuf);

//растим buffer строку из кодовых слов

 For j:=1 to lastbuf do

 Begin

 buffer:=buffer+ArrOfStr[buf[j]];

 If Length(buffer)>8\*count

//если его размер превысит значение 8\*4096 (а это может иметь

//место), то вызываем процедуру записи в файл

 Then

 WriteInFile(buffer);

 Application.ProcessMessages;

 End;

 End;

//выываем процедуру записи оставшейся цепочки кодовых слов

 WriteInFile\_(buffer);

 CloseFile(FileToRead);

 CloseFile(FileToWrite);

 Except

 ShowMessage('Ошибка создания архива');

 End;

End;

//главная процедура для создания архивного файла

Procedure CreateFile; //(802)

var

 i: Byte;

Begin

 With MainFile do

 Begin

 {сортировка массива байтов с частотами (192)}

 SortMassiv(Stat.massiv,stat.CountByte);

 {поиск числа задействованных байтов из массива

 (ACSII) возмжных символов. В CountByte будем хранить

 количество этох самых символов }

 i:=0;//обнуляем счётчик

 While (i<Stat.CountByte) //до тех пор пока счётчик

 //меньше количества задействовнных байт CountByte

 //и статистика байта (частота появления в файле)

 //не равна нулю делаем

 and (Stat.massiv[i]^.SymbolStat<>0) do

 Begin

 Inc(i); //увеличиваем счётчик на единицу

 End;

 //////////////////////

 If Stat.massiv[i]^.SymbolStat=0 //если дошли до символа

 //с нулевой встречаемостью в файле то

 Then

 Dec(i); //уменьшаем счётчик на единицу тоесть возвращаемся

 //назад это будет последний элемент

 //////////////////////

 Stat.CountByte:=i;{присваиваем значение счётчика

CountByte. Это означает что в архивируемом файле используется такое количество из 256 возможных символов. Будет исползоватся для построения древа частот} {создание дерева частот. Передаём в процедуру начальные параметры Tree=nil-эта переменная будет содержать после работы процедуры древо ,Stat.massiv-массив с символами и соответствующей им статистикой,а так же указанием на правое и левой дерево,Stat. CountByte количество используемых символов в архивирумом файле (230)} CreateTree(Tree,Stat.massiv,Stat.CountByte); {запускаем в работу дерево с помощью его нахадим соответствующие кодовые слова. Суть алгоритма вызываем функцию SymbolToCodWord(Tree:TByte(указатель на корень дерева. Он у нас выработался в результате работы процедуры CreateTree, Symbol:byte):

String функция вернёт нам строку содержащую кодовое слово ()}

for i:=0 to Stat.CountByte do

Stat.massiv[i]^.CodWord:=SymbolToCodWord(Tree,stat.massiv[i]^.Symbol);

//пишем сам файл

 CreateArchiv;

//Удаляем уже ненужное дерево

 DeleteTree(Tree);

//Инициализируем статистику файла

 MainFile.Stat.Create;

 End;

End;

//Основная процедура сжатия файла

procedure RunEncodeHaff(FileName\_: string);

begin

 MainFile.Name:=FileName\_;//передаём имя

//архивируемого файла в программу

 StatFile(MainFile.Name); //запускем процедуру создания

//статистики (частоты появления того или иного символа)

//для файла (строка 274)

 CreateFile; //вызов процедуры созданя архивного файла (737)

end;

//Основная процедура разархивирования файла

procedure RunDecodeHaff(FileName\_: string);

begin

 MainFile.name:=FileName\_;//передаём имя

 //архивируемого файла в программу

 ExtractFile;//Вызываем процедуру извлечения архива

end;

end.