Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Операционные системы и среды

**ПОЯСНИТЕЛЬНАЯ** **ЗАПИСКА**

к курсовому проекту

на тему

**«Сжатие данных без потерь»**

Студент Ермолович Дмитрий Сергеевич

Гр. 053504

 Ассистент кафедры информатики

Руководитель Давыдчик А.В.

Минск 2023

**СОДЕРЖАНИЕ**

[Введение 3](#_Toc132394257)

[1 Теоретические сведения 4](#_Toc132394258)

[1.1 Виды сжатия данных 4](#_Toc132394259)

[1.2 Универсальные методы сжатия без потерь 4](#_Toc132394260)

[1.3 Общие принципы, на которых основано сжатие данных 5](#_Toc132394261)

[1.4 Энтропия 5](#_Toc132394262)

[1.5 Критерии оценки методов сжатия 6](#_Toc132394263)

[2 Платформа программного обеспечения 8](#_Toc132394264)

[3 Кодирование без памяти 9](#_Toc132394265)

[3.1 Основные понятия 9](#_Toc132394266)

[3.2 Алгоритм Хаффмана 10](#_Toc132394267)

[4 Кодирование Шеннона-Фано 16](#_Toc132394268)

[5 Преобразование Барроуза-Уилира 19](#_Toc132394269)

[6 Сжатие данных LZW 23](#_Toc132394270)

[Список источников 28](#_Toc132394271)

[Приложение А (обязательное) Листинг кода алгоритма Шеннона-Фано 29](#_Toc132394272)

[Приложение Б (обязательное) Листинг кода алгоритма LZW 32](#_Toc132394273)

[Приложение В (обязательное) Листинг кода алгоритма Хаффмана 34](#_Toc132394274)

[Приложение Г (обязательное) Листинг кода алгоритма BWT 37](#_Toc132394275)

# **ВВЕДЕНИЕ**

# 

Сжатие данных - это процесс, при котором исходные данные сжимаются для уменьшения их объема, тем самым упрощая их хранение и передачу. Существует два основных типа сжатия данных: сжатие с потерями и сжатие без потерь. В данном контексте рассмотрим сжатие данных без потерь. В моей курсовой работе я буду рассматривать сжатие данных без потерь.

Сейчас нам доступны носители информации большого объема, и высокоскоростные каналы передачи данных. Однако, одновременно с этим растут и объемы передаваемой информации. Если несколько лет назад можно было смотреть 700-мегабайтные фильмы, умещающиеся на одну флешку, то сегодня фильмы в HD-качестве могут занимать десятки гигабайт.

Конечно, пользы от сжатия всего, и вся не так много. Но все же существуют ситуации, в которых сжатие крайне полезно, если не необходимо.

1 Пересылка документов по электронной почте (особенно больших объемов документов с использованием мобильных устройств)

2 При публикации документов на сайтах, потребность в экономии трафика

3 Экономия дискового пространства в тех случаях, когда замена или добавление средств хранения затруднительно.

Цель моей работы исследовать разные алгоритмы сжатия данных без потерь, сравнить их, выделить их плюсы и минусы.

## **1 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

## **1.1 Виды сжатия данных**

Все методы сжатия можно разделить на две большие группы: сжатие с потерями и сжатие без потерь. Сжатие без потерь применяется в тех случаях, когда информацию нужно восстановить с точностью до бита. Такой подход является единственно возможным при сжатии, например, текстовых данных.

В некоторых случаях, однако, не требуется точного восстановления информации и допускается использовать алгоритмы, реализующие сжатие с потерями, которое, в отличие от сжатия без потерь, обычно проще реализуется и обеспечивает более высокую степень архивации.

Виды сжатий:

1 Сжатие с потерями:

При таком сжатии мы теряем часть информации. Но смысл алгоритмов сжатия в том, чтобы мы этого не замечали: сжатие должно происходить так, чтобы всё важное передалось, а неважное — нет. Лучшие степени сжатия, при сохранении «достаточно хорошего» качества данных. Применяются в основном для сжатия аналоговых данных — звука, изображений. В таких случаях распакованный файл может очень сильно отличаться от оригинала на уровне сравнения «бит в бит», но практически неотличим для человеческого уха или глаза в большинстве практических применений.

2 Сжатие без потерь:

Данные восстанавливаются с точностью до бита, что не приводит к каким-либо потерям информации. Однако, сжатие без потерь показывает обычно худшие степени сжатия.

## **1.2 Универсальные методы сжатия без потерь**

В общем случае можно выделить три базовых варианта, на которых строятся алгоритмы сжатия.

Первая группа методов – преобразование потока. Это предполагает описание новых поступающих несжатых данных через уже обработанные. При этом не вычисляется никаких вероятностей, кодирование символов осуществляется только на основе тех данных, которые уже были обработаны, как например в LZ – методах (названных по имени Абрахама Лемпеля и Якоба Зива). В этом случае, второе и дальнейшие вхождения некой подстроки, уже известной кодировщику, заменяются ссылками на ее первое вхождение.

Вторая группа методов – это статистические методы сжатия. В свою очередь, эти методы делятся на адаптивные (или поточные), и блочные.

В первом (адаптивном) варианте, вычисление вероятностей для новых данных происходит по данным, уже обработанным при кодировании. К этим методам относятся адаптивные варианты алгоритмов Хаффмана и Шеннона-Фано.

Во втором (блочном) случае, статистика каждого блока данных высчитывается отдельно, и добавляется к самому сжатому блоку. Сюда можно отнести статические варианты методов Хаффмана, Шеннона-Фано, и арифметического кодирования.

Третья группа методов – это так называемые методы преобразования блока. Входящие данные разбиваются на блоки, которые затем трансформируются целиком. При этом некоторые методы, особенно основанные на перестановке блоков, могут не приводить к существенному (или вообще какому-либо) уменьшению объема данных. Однако после подобной обработки, структура данных значительно улучшается, и последующее сжатие другими алгоритмами проходит более успешно и быстро.

## **1.3 Общие принципы, на которых основано сжатие данных**

Все методы сжатия данных основаны на простом логическом принципе. Если представить, что наиболее часто встречающиеся элементы закодированы более короткими кодами, а реже встречающиеся – более длинными, то для хранения всех данных потребуется меньше места, чем если бы все элементы представлялись кодами одинаковой длины.

Точная взаимосвязь между частотами появления элементов, и оптимальными длинами кодов описана в так называемой теореме Шеннона о источнике шифрования (Shannon's source coding theorem), которая определяет предел максимального сжатия без потерь и энтропию Шеннона.

## **1.4 Энтропия**

Если вероятность появления элемента равна p(), то наиболее выгодно будет представить этот элемент — p() битами. Если при кодировании удается добиться того, что длина всех элементов будет приведена к p() битам, то и длина всей кодируемой последовательности будет минимальной для всех возможных методов кодирования. При этом, если распределение вероятностей всех элементов F = { p()} неизменно, и вероятности элементов взаимно независимы, то средняя длина кодов может быть рассчитана как показано на формуле 1:

, (1)

Это значение называют энтропией распределения вероятностей F, или энтропией источника в заданный момент времени.

Однако обычно вероятность появления элемента не может быть независимой, напротив, она находится в зависимости от каких-то факторов. В этом случае, для каждого нового кодируемого элемента si распределение вероятностей F примет некоторое значение Fk, то есть для каждого элемента F= Fk и H= Hk.

Иными словами, можно сказать, что источник находится в состоянии k, которому соответствует некий набор вероятностей для всех элементов

Поэтому, учитывая эту поправку, можно выразить среднюю длину кодов как показано на формуле 2:

, (2)

где  — вероятность нахождения источника в состоянии k.

## **1.5 Критерии оценки методов сжатия**

Основными свойствами какого-либо алгоритма сжатия данных являются:

1 качество (коэффициент или степень) сжатия, т. е. отношение длины (в битах) сжатого представления данных к длине исходного представления;

2 скорость кодирования и декодирования, определяемые временем, затрачиваемым на кодирование и декодирование данных;

3 объем требуемой памяти.

В области сжатия данных, как это часто случается, действует закон рычага: алгоритмы, использующие больше ресурсов (времени и памяти), обычно достигают лучшего [качества сжатия](http://mf.grsu.by/UchProc/livak/po/comprsite/theory_classification_01.html#ratio), и наоборот: менее ресурсоемкие алгоритмы по [качеству сжатия](http://mf.grsu.by/UchProc/livak/po/comprsite/theory_classification_01.html#ratio), как правило, уступают более ресурсоемким.

Понятно, что критерии оценки методов сжатия с практической точки зрения сильно зависят от предполагаемой области применения. Например, при использовании сжатия в системах реального времени необходимо обеспечить высокую скорость кодирования и декодирования; для встроенных систем критический параметр – объем требуемой памяти; для систем долговременного хранения данных – качество сжатия и/или скорость декодирования и т. д.

**1.6 Надежность программ и сложность алгоритмов**

Надежность программных систем и комплексов очень важна и обеспечивается как безошибочностью программирования и дизайна, так и характеристиками использованных алгоритмов.

Если количество ошибок в основном определяется полнотой и качеством тестирования (а также квалификацией и культурой программирования) и мало зависит от воли разработчика, то выбор алгоритмов – вполне управляемый и контролируемый процесс.

Для обеспечения конечного и заранее известного времени сжатия (в наихудшем случае), необходимо, чтобы алгоритм обладал хорошо детерминированным временем работы (желательно, мало зависящим от кодируемых данных) и заранее известным объемом требуемой памяти. В частности, выполнение этих требований необходимо при разработке встроенных систем, систем реального времени, файловых систем со сжатием данных и других систем с жесткими ограничениями на разделяемые различными процессами ресурсы.

Если с теоретической точки зрения полиномиальные алгоритмы, обладающие полиномиальной или экспоненциальной сложностью, считаются хорошим решением проблемы, то на практике приемлемы только алгоритмы с линейной или линейно-логарифмической временной сложностью, причем крайне желательно, чтобы среднее время работы (на типичных данных) было линейным.

# **2 ПЛАТФОРМА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ**

PyCharm — это кроссплатформенная интегрированная среда разработки для языка программирования Python, разработанная компанией JetBrains на основе IntelliJ IDEA. Предоставляет пользователю комплекс средств для написания кода и визуальный отладчик.

Продукт доступен в двух версиях: PyCharm Community Edition — бесплатная версия, находится под лицензией Apache License, и PyCharm Professional Edition — расширенная версия продукта, обладающая дополнительной функциональностью, является проприетарным ПО.

**История**

PyCharm был выпущен на рынок интегрированных сред разработки для создания конкуренции с PyDev (однако, на данный момент PyCharm использует PyDev для отладки кода) и более распространённой среды разработки Komodo IDE. Бета-версия была выпущена в июле 2010 года, версия 1.0 была выпущена тремя месяцами позже.

Версия 2.0 вышла 13 декабря 2011 года.

Версия 3.0 была выпущена 24 сентября 2013 года.

PyCharm Community Edition, бесплатная версия с открытым исходным кодом, была опубликована 22 октября 2013 года.

В марте 2016 года JetBrains перешла на подписную модель лицензирования, а вместе с этим изменилась и нумерация версий. Теперь номер версии выглядит как YYYY.R, где YYYY — год выпуска, а R — выпуск в течение этого года.

**Возможности**

1 Отладка кода при помощи PyDev;

2 Рефакторинг кода;

3 Поддержка Git, SVN, Mercurial и других систем контроля версиями;

4 Автодополнение кода.

# **3 КОДИРОВАНИЕ БЕЗ ПАМЯТИ**

## **3.1 Основные понятия**

Коды без памяти являются простейшими кодами, на основе которых может быть осуществлено сжатие данных. В коде без памяти каждый символ в кодируемом векторе данных заменяется кодовым словом из префиксного множества двоичных последовательностей или слов.

Пусть задан некоторый алфавит состоящий из некоторого (конечного) числа букв. Назовем каждую конечную последовательность символов из этого алфавита словом, а число n — длиной этого слова.

Пусть задан также другой алфавит Аналогично, обозначим слово в этом алфавите как B.

Введем еще два обозначения для множества всех непустых слов в алфавите. Пусть  — количество непустых слов в первом алфавите, а  — во втором.

Пусть также задано отображение F, которое ставит в соответствие каждому слову A из первого алфавита некоторое слово B=F(A) из второго. Тогда слово B будет называться кодом слова A, а переход от исходного слова к его коду будет называться кодированием.

Поскольку слово может состоять и из одной буквы, то мы можем выявить соответствие букв первого алфавита и соответствующих им слов из второго:

a1 <->B1  
a2 <->B2  
…  
an <-> Bn

Это соответствие называют схемой, и обозначают ∑.

В этом случае слова B1, B2,…, Bn называют элементарными кодами, а вид кодирования с их помощью — алфавитным кодированием. Конечно, большинство из нас сталкивались с таким видом кодирования, пусть даже и не зная всего того, что я описал выше.

Итак, мы определились с понятиями алфавит, слово, код, и кодирование. Теперь введем понятие префикс.

Пусть слово B имеет вид B=B'B''. Тогда B' называют началом, или префиксом слова B, а B'' — его концом. Это довольно простое определение, но нужно отметить, что для любого слова B, и некое пустое слово ʌ («пробел»), и само слово B, могут считаться и началами, и концами.

Итак, мы подошли вплотную к пониманию определения кодов без памяти. Последнее определение, которое нам осталось понять — это префиксное множество. Схема ∑ обладает свойством префикса, если для любых 1≤i, j≤r, i≠j, слово Bi не является префиксом слова Bj. Проще говоря, префиксное множество – это такое конечное множество, в котором ни один элемент не является префиксом (или началом) любого другого элемента. Простым примером такого множества является, например, обычный алфавит.

Итак, мы разобрались с основными определениями. Так как же происходит само кодирование без памяти?

Оно происходит в три этапа:

1 Составляется алфавит Ψ символов исходного сообщения, причем символы алфавита сортируются по убыванию их вероятности появления в сообщении.

2 Каждому символу ai из алфавита Ψ ставится в соответствие некое слово Bi из префиксного множества Ω.

3 Осуществляется кодирование каждого символа, с последующим объединением кодов в один поток данных, который будет являться результатам сжатия.

Одним из канонических алгоритмов, которые иллюстрируют данный метод, является алгоритм Хаффмана.

## **3.2 Алгоритм Хаффмана**

Каждый символ представляет собой последовательность 0's а также 1's и хранится с использованием 8-бит. Это известно, как “кодирование с фиксированной длиной”, так как каждый символ использует одинаковое количество фиксированных битов памяти.

Как уменьшить количество места, необходимое для хранения символа?

Идея состоит в том, чтобы использовать “кодирование переменной длины”. Мы можем использовать тот факт, что одни символы встречаются в тексте чаще, чем другие для разработки алгоритма, который может представлять тот же фрагмент текста, используя меньшее количество битов. При кодировании с переменной длиной мы присваиваем символам переменное количество битов в зависимости от их частоты в данном тексте. Таким образом, некоторые символы могут в конечном итоге занимать один бит, а некоторые — два бита, некоторые могут быть закодированы с использованием трех битов и так далее. Проблема с кодированием переменной длины заключается в его декодировании [1].

Рассмотрим строку aabacdab. Оно имеет 8 символов в нем и использует 64-битное хранилище (с использованием кодирования фиксированной длины). Если принять во внимание, что частота символов a, b, c, а также d находятся 4, 2, 1, 1, соответственно. Попробуем представить aabacdab используя меньшее количество битов, используя тот факт, что a встречается чаще, чем b, а также b встречается чаще, чем c а также d. Начнем со случайного присвоения однобитового кода 0 к a, 2-битный код 11 к b, и 3-битный код 100 а также 011 к персонажам c а также d, соответственно.

На рисунке 1 приведен пример кодирования символов.



Рисунок 1 – Кодирования символов

Итак, строка aabacdab будет закодирован в 00110100011011 (0|0|11|0|100|011|0|11) используя приведенные выше коды. Но настоящая проблема заключается в расшифровке. Если мы попытаемся декодировать строку 00110100011011, это приведет к неоднозначности, так как его можно декодировать.

На рисунке 2 приведен пример декодирования строк.

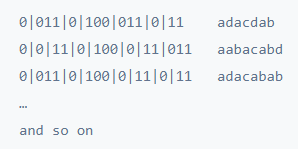


Рисунок 2 – Декодирования строк

Чтобы предотвратить двусмысленность при декодировании, мы обеспечим соответствие нашего кодирования “правилу префикса”, что приведет к “уникально декодируемым кодам”. Правило префикса гласит, что ни один код не является префиксом другого кода. Под кодом мы подразумеваем биты, используемые для определенного символа. В приведенном выше примере 0 является префиксом 011, что нарушает правило префикса. Если наши коды удовлетворяют префиксному правилу, декодирование будет однозначным (и наоборот) [2].

Давайте снова рассмотрим приведенный выше пример. На этот раз мы присваиваем символам коды, удовлетворяющие правилу префикса. 'a', 'b', 'c', а также 'd'.

На рисунке 3 приведен пример кодирования символов в соответствии с префиксами.



Рисунок 3 – Кодирования символов в соответствии с префиксами

Используя приведенные выше коды, строка aabacdab будет закодирован в 00100110111010 (0|0|10|0|110|111|0|10). Теперь мы можем однозначно декодировать 00100110111010 вернуться к нашей исходной строке aabacdab.

Теперь, когда мы разобрались с кодированием переменной длины и правилом префиксов, давайте поговорим о кодировании Хаффмана.

**Кодирование Хаффмана**

Техника работает, создавая бинарное дерево узлов. Узел может быть листовым узлом или внутренним узлом. Изначально все узлы являются листовыми узлами, которые содержат сам персонаж, вес (частоту появления) персонажа. Внутренние узлы содержат вес символов и ссылки на два дочерних узла. По общему соглашению, бит 0 представляет следующий левый дочерний элемент, и немного 1 представляет следующий правильный ребенок. Готовое дерево имеет n листовые узлы и n-1 внутренние узлы. Рекомендуется, чтобы дерево Хаффмана отбрасывало неиспользуемые символы в тексте, чтобы получить наиболее оптимальную длину кода [3].

Мы будем использовать приоритетная очередь для построения дерева Хаффмана, где узел с наименьшей частотой имеет наивысший приоритет. Ниже приведены полные шаги [1]:

1. Создайте конечный узел для каждого символа и добавьте их в очередь приоритетов.

2. Пока в queue больше одного узла:

2.1 Удалите из queue два узла с наивысшим приоритетом (самой низкой частотой).

2.2 Создайте новый внутренний узел с этими двумя узлами в качестве дочерних элементов и частотой, равной сумме частот обоих узлов.

2.3 Добавьте новый узел в очередь приоритетов.

3. Оставшийся узел является корневым узлом, и дерево завершено.

Рассмотрим некоторый текст, состоящий только из 'A', 'B', 'C', 'D', а также 'E' символов, а их частота 15, 7, 6, 6, 5, соответственно. Рисунке 4,5 иллюстрируют шаги, за которыми следует алгоритм.

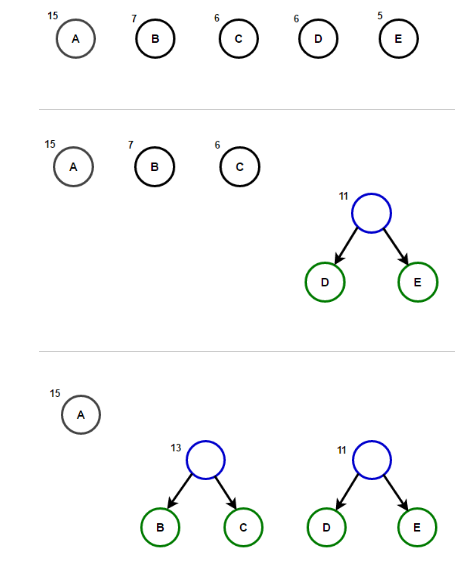
****

Рисунок 4 – Алгоритм Хаффмана

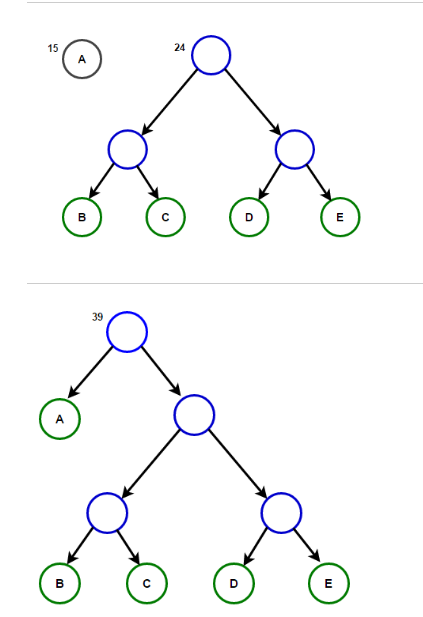
****

Рисунок 5 – Алгоритм Хаффмана

Путь от корня к любому конечному узлу хранит оптимальный код префикса (также называемый кодом Хаффмана), соответствующий символу, связанному с этим конечным узлом.

На рисунке 6 изображено дерева с помощью которого можно закодировать символ в соответствии с его частотой.

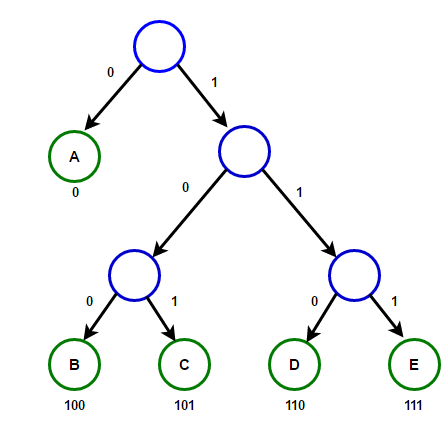
****

Рисунок 6 – Конечное дерево

Рассмотрим еще один пример пусть на вход у нас была строка из 1000 символов, в которой символ a1 встречался 500 раз, a2 — 240, a3 — 150, и a4 — 110 раз. После кодирования мы получим a1 = 0, a2 = 11, a3 = 100, a4 = 101.

Если предположить, что изначально для хранения каждого символа использовался один байт, то можно посчитать, насколько нам удалось уменьшить данные.

Изначально данная строка занимала 8000 бит. После кодирования мы получим строку длинной в ∑pili = 500 \* 1 + 240 \* 2 + 150 \* 3 + 110 \* 3 = 1760 бит. Итак, нам удалось сжать данные в 4,54 раза, потратив в среднем 1,76 бита на кодирование каждого символа потока.

Cогласно Шеннону, средняя длина кодов составляет .

Подставив в это уравнение наши значения вероятностей, мы получим среднюю длину кодов равную 1.75496602732291, что весьма и весьма близко к полученному нами результату.

# **4 КОДИРОВАНИЕ ШЕННОНА-ФАНО**

Кодирование Шеннона-Фано является одним из самых первых алгоритмов сжатия, который впервые сформулировали американские учёные Шеннон (Shannon) и Фано (Fano). Данный метод сжатия имеет большое сходство с кодированием Хаффмана, которое появилось на несколько лет позже. Главная идея этого метода - заменить часто встречающиеся символы более короткими кодами, а редко встречающиеся последовательности более длинными кодами. Таким образом, алгоритм основывается на кодах переменной длины. Для того, чтобы декомпрессор впоследствии смог раскодировать сжатую последовательность, коды Шеннона-Фано должны обладать уникальностью, то есть, не смотря на их переменную длину, каждый код уникально определяет один закодированный символ и не является префиксом любого другого кода [4].

Рассмотрим алгоритм вычисления кодов Шеннона-Фано (для наглядности возьмём в качестве примера последовательность 'aa bbb cccc ddddd'). Для вычисления кодов, необходимо создать таблицу уникальных символов сообщения c(i) и их вероятностей p(c(i)), и отсортировать её в порядке невозрастания вероятности символов.

На рисунке 7 приведена таблица вероятностей символов.



Рисунок 7 – Таблица вероятностей символов

Далее, таблица символов делится на две группы таким образом, чтобы каждая из групп имела приблизительно одинаковую частоту по сумме символов. Первой группе устанавливается начало кода в '0', второй в '1'. Для вычисления следующих бит кодов символов, данная процедура повторяется рекурсивно для каждой группы, в которой больше одного символа. Таким образом для нашего случая получаем следующие коды символов [5].

На рисунке 8 приведена таблица кодирования символов.

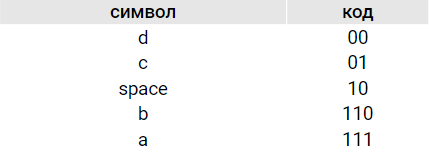


Рисунок 8 – Таблица кодирования символов

Длина кода s(i) в полученной таблице равна int(-lg p(c(i))), если символы удалость разделить на группы с одинаковой частотой, в противном случае, длина кода равна int(-lg p(c(i))) + 1. Это можно записать в виде отношения: int(-lg p(c(i))) <= s(i) <= int(-lg p(c(i))) + 1.

Используя полученную таблицу кодов, кодируем входной поток - заменяем каждый символ соответствующим кодом. Естественно для расжатия полученной последовательности, данную таблицу необходимо сохранять вместе со сжатым потоком, что является одним из недостатков данного метода. В сжатом виде, наша последовательность принимает вид: 111111101101101101001010101100000000000. Ее длина в 39 бит. Учитывая, что оригинал имел длину равную 136 бит, получаем коэффициент сжатия ~28% - не так уж и плохо [5].

Мы не можем, как в случае кодирования, заменять каждые 8 бит входного потока, кодом переменной длины. При расжатии нам необходимо всё сделать наоборот - заменить код переменной длины символом длиной 8 бит. В данном случае, лучше всего будет использовать бинарное дерево, листьями которого будут является символы (аналог дерева Хаффмана) [4].

Кодирование Шеннона-Фано является достаточно старым методом сжатия, и на сегодняшний день оно не представляет особого практического интереса (разве что как упражнение по курсу структур данных). В большинстве случаев, длина сжатой последовательности, по данному методу, равна длине сжатой последовательности с использованием кодирования Хаффмана. Но на некоторых последовательностях всё же формируются не оптимальные коды Шеннона-Фано, поэтому сжатие методом Хаффмана принято считать более эффективным. Для примера, рассмотрим последовательность с таким содержанием символов: 'a' - 14, 'b' - 7, 'c' - 5, 'd' - 5, 'e' - 4. Метод Хаффмана сжимает её до 77 бит, а вот Шеннона-Фано до 79 бит.

На рисунке 9 приведена таблица кодирования символов методами Хаффмана и Шеннона-Фано.

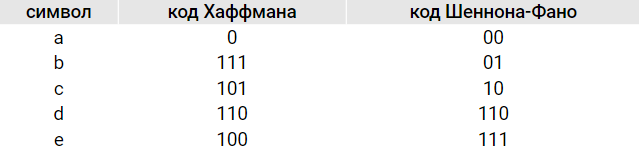


Рисунок 9 – Таблица кодирования символов разными способами

# **5 ПРЕОБРАЗОВАНИЕ БАРРОУЗА-УИЛИРА**

BWT-преобразование (Burrows-Wheeler Transform) – техника для сжатия информации (в особенности текстов), основанная на преобразовании, открытом в 1983 г. и описанная в 1994 г. BWT является удивительным алгоритмом. Во-первых, необычно само преобразование, открытое в научной области, далекой от архиваторов. Во-вторых, даже зная BWT, не совсем ясно, как его применить к сжатию информации. В-третьих, BWT преобразование чрезвычайно просто. И, наконец, сам BWT компрессор состоит из последовательности нескольких рассмотренных ранее алгоритмов и требует, поэтому, для своей реализации самых разнообразных программных навыков [6].

BWT не сжимает данные, но преобразует блок данных в формат, исключительно подходящий для компрессии.

Прежде всего, следует отметить одну из его особенностей. BWT оперирует сразу целым блоком данных. То есть, ему заранее известны сразу все элементы входного потока или, по крайней мере, достаточно большого блока. Это делает затруднительным использование алгоритма в тех областях применения, где требуется сжатие данных “на лету”, символ за символом. В этом отношении BWT даже более требователен, чем методы семейства LZ, использующие для сжатия скользящее окно.

Следует отметить, что возможна реализация сжатия данных на основе BWT, обрабатывающая данные последовательно по символам, а не по блокам. Но скоростные характеристики программ, использующих такую реализацию, будут очень далеки от совершенства [7].

Первая фаза преобразования – выделение из непрерывного потока блока данных. Далее, нужно из полученного блока данных создать матрицу всех возможных его циклических перестановок. Первой строкой матрицы будет исходная последовательность, второй строкой – она же, сдвинутая циклически на один символ влево и т.д. Рассмотрим преобразование на примере строки "абракадабра". Таким образом, получим следующую матрицу:

0 абракадабра

1 бракадабраа

2 ракадабрааб

3 акадабраабр

4 кадабраабра

5 адабраабрак

6 дабраабрака

7 абраабракад

8 браабракада

9 раабракадаб

10 аабракадабр

Пометим в этой матрице исходную строку и отсортируем все строки в соответствии с лексикографическим порядком символов. Будем считать, что одна строка должна находиться в матрице выше другой в том случае, если в самой левой из позиций, начиная с которой строки отличаются, в этой строке находится символ лексикографически меньший, чем у другой строки. Другими словами, следует отсортировать строки сначала по первому символу, затем строки, у которых первые символы равны – по второму и т. д.

0 аабракадабр

1 абраабракад

2 абракадабра – исходная строка

3 адабраабрак

4 акадабраабр

5 браабракада

6 бракадабраа

7 дабраабрака

8 кадабраабра

9 раабракадаб

10 ракадабрааб

Теперь остался последний шаг – выписать символы последнего столбца и запомнить номер исходной строки среди отсортированных. В нашем примере "рдакраааабб", 2 – это результат, полученный в результате преобразования Барроуза-Уилера.

**Обратное преобразование**

Пусть нам дано: BWT(s)=(«BCABAAA», 3). Тогда выпишем в столбик нашу преобразованную последовательность символов «BCABAAA». Запишем её как последний столбик предыдущей матрицы (при прямом преобразовании Барроуза — Уилера), при этом все предыдущие столбцы оставляем пустыми. Далее построчно отсортируем матрицу, затем в предыдущий столбец запишем «BCABAAA». Опять построчно отсортируем матрицу. Продолжая таким образом, можно восстановить полный список всех циклических перестановок строки, которую нам надо найти. Выстроив полный отсортированный список перестановок, выберем строку с номером, который нам был изначально дан. В итоге мы получим искомую строку [6].

Алгоритм обратного преобразования представлен на рисунке 10.

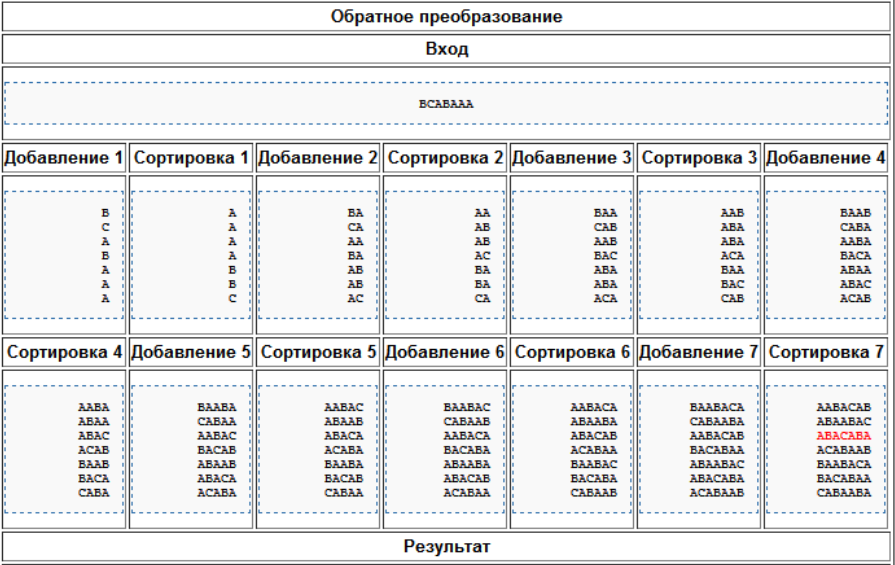


Рисунок 10 – Алгоритм обратного преобразования

**Использование BWT в сжатии данных**

После того, как выяснилось, что наши действия вполне обратимы, и данные мы не исказим, можно перейти к вопросу рассмотрения полезности преобразования. Главная задача преобразования Барроуза-Уилера заключается в том, чтобы переставить символы таким образом, чтобы их можно было легко сжать.

Для понимания этого процесса достаточно представить поток данных состоящим из набора некоторых стабильных сочетаний символов. Сочетание символов, позволяющее предсказать некоторый неизвестный символ, называется контекстом. Например, если нам известна последовательность символов "реобразование", то скорее всего ей предшествует символ "п". Назовем устойчивым (стабильным) такой контекст, для которого распределение частот символов, непосредственно примыкающих к нему слева и справа, меняется незначительно в пределах блока [7].

Если нам потребуется подвергнуть преобразованию текст данной главы, можно с уверенностью сказать, что строки, начинающиеся с символов "реобразование", будут располагаться рядом в отсортированной матрице. И в соответствующих этим строкам позициях последнего столбца матрицы будет находится символ "п".

Таким образом, главное свойство преобразования в том, что оно собирает вместе символы, соответствующие похожим контекстам. Чем больше стабильных контекстов в блоке данных, тем лучше будет сжиматься полученный в результате преобразования блок. Практика показывает, что в результате преобразования обычных текстов более половины из всех символов следует за такими же.

**Эффективность**

Можно отметить характерные особенности архиваторов, использующих описанное преобразование, по сравнению с другими. Скорость сжатия – на уровне архиваторов, применяющих словарные методы, например, LZ77. Декомпрессия, как правило в 3-4 раза быстрее сжатия. Степень сжатия сильно зависит от типа данных.

Наиболее эффективно применение BWT-архиваторов для текстов и любых данных со стабильными контекстами. Расходы памяти в режиме максимального сжатия довольно близки у всех современных архиваторов. По расходу памяти при декодировании архиваторы на основе BWT занимают промежуточное положение.

# **6 СЖАТИЕ ДАННЫХ LZW**

**Обзор**

Если бы вы взглянули почти на любой файл данных в компьютере, просматривая символ за символом, то наверняка обратили бы внимание на множество повторяющихся элементов. LZW — это метод сжатия данных, который воспользовался этим повторением. Оригинальная версия метода была создана Лемпелем и Зивом в 1978 году (LZ78) и доработана Уэлчем в 1984 году, отсюда и аббревиатура LZW (Lempel, Ziv and Welch). Как и в любом адаптивном/динамическом методе сжатия, идея заключается в том, чтобы (1) начать с исходной модели, (2) читать данные по частям, (3) обновлять модель и кодировать данные по мере продвижения. LZW — алгоритм сжатия на основе "словаря". Это означает, что вместо сведения в таблицу количества символов и построения деревьев (как при кодировании по Хаффману), LZW кодирует данные, обращаясь к словарю. Таким образом, чтобы закодировать подстроку, в выходной файл нужно записать только одно кодовое число, соответствующее индексу этой подстроки в словаре. Хотя LZW часто рассматривается в контексте сжатия текстовых файлов, его можно использовать для любого типа файлов. Однако, как правило, он лучше всего справляется с файлами где есть повторяющиеся подстроки, например, с текстовыми файлами [8].

**Сжатие**

LZW начинает со словаря из 256 символов (в случае 8 бит) и использует их в качестве "стандартного" набора символов. Затем он считывает данные по 8 бит за раз (например, 't', 'r' и т.д.) и кодирует их в виде числа, которое представляет собой индекс в словаре. Всякий раз, встречая новую подстроку (скажем, "tr"), он добавляет ее в словарь; каждый раз, когда ему попадается подстрока, которая ранее уже встречалась, он просто считывает новый символ и выполняет его конкатенацию с текущей строкой, чтобы получить новую подстроку. В следующий раз, когда LZW вновь обратится к подстроке, она будет закодирована с помощью одного числа. Обычно для словаря задается максимальное количество записей (скажем, 4096), чтобы процесс не исчерпал память. Таким образом, коды, занимающие место подстрок в данном примере, имеют длину 12 бит (2^12 = 4096). Это необходимо, чтобы коды были длиннее в битах, чем символы (12 против 8 бит), но поскольку вместо большого количества часто встречающихся подстрок будет использоваться только один код, в конечном итоге достигается сжатие [9].

На рисунке 13 приведен пример кодирования LZW.

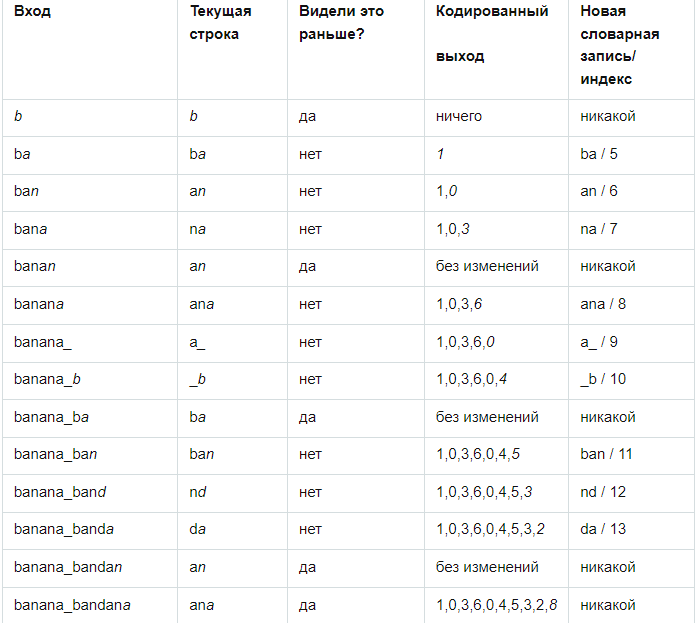


Рисунок 13 – Пример кодирования LZW

**Распаковка**

Процесс декомпрессии (распаковки) для LZW также прост. Кроме того, он имеет преимущество перед статическими методами сжатия, поскольку для алгоритма декодирования не требуется словарь или другая дополнительная информация - в процессе работы словарь, идентичный тому, который был создан во время сжатия, восстанавливается. Программы кодирования и декодирования должны начинаться с одного и того же начального словаря, в данном случае со всех 256 символов ASCII [8].

Вот как это работает. Декодер LZW сначала считывает индекс (целое число), ищет этот индекс в словаре и выводит подстроку, связанную с этим индексом. Первый символ этой подстроки конкатенируется с текущей рабочей строкой. Эта новая конкатенация добавляется в словарь (подобно тому, как подстроки были добавлены во время сжатия). Затем декодированная строка становится текущей рабочей строкой (текущий индекс, т.е. подстрока, запоминается), и процесс повторяется.

Есть исключение, когда алгоритм не работает; это происходит, когда код вызывает индекс, который еще не был введен (например, вызов индекса 31, когда индекс 31 в настоящее время обрабатывается и поэтому его еще нет в словаре). Пример из Sayood поможет проиллюстрировать этот момент. Предположим, у вас есть строка ababababab..... и начальный словарь, состоящий только из a и b с индексами 0 и 1 соответственно.

На рисунке 14 приведен пример кодирования LZW.

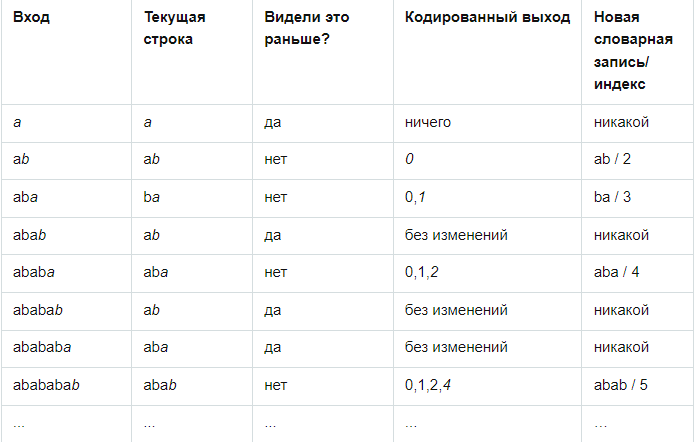


Рисунок 14 – Пример кодирования LZW

Итак, кодированный выход начинается с 0,1,2,4,... . Когда мы начинаем пытаться декодировать, возникает проблема (в приведенной ниже таблице следует помнить, что текущая строка — это просто подстрока, которая была декодирована/переведена в последнем проходе цикла. Кроме того, новая словарная запись создается путем конкатенации текущей строки с первым символом нового преобразования словаря):

На рисунке 15 приведен пример декодирования LZW.

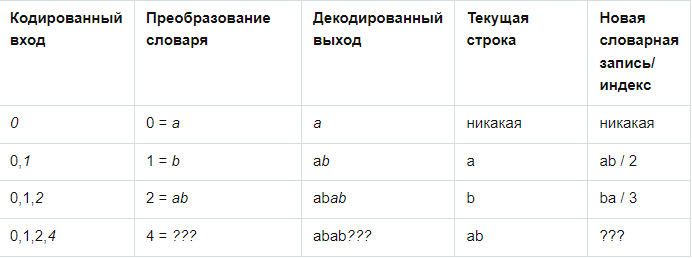


Рисунок 15 – Пример декодирования LZW

Как видите, декодер сталкивается с индексом 4, в то время как принадлежащая ему запись в данный момент обрабатывается. Чтобы понять, почему это происходит, взгляните на таблицу кодирования. Сразу же после того, как "aba" (с индексом 4) заносится в словарь, следующей кодируемой подстрокой является "aba" (т.е. следующим кодом, записанным в кодированный выходной файл, будет 4). Таким образом, этот особый сценарий может возникнуть только в том случае, если подстрока начинается и заканчивается одним и тем же символом ("aba" имеет вид <char><string><char>. <знак><строка><знак>). Поэтому, чтобы справиться с этим исключением, вы просто берете подстроку, которую уже получили, "ab", и конкатенируете ее первый символ с самим собой, "ab "+"a" = "aba", вместо того, чтобы следовать процедуре как обычно. Поэтому приведенный выше псевдокод должен быть немного изменен, чтобы справиться со всеми случаями [9].

**ЗАКЛЮЧЕНИЕ**

Сжатие данных без потерь - это процесс сжатия данных, при котором оригинальные данные могут быть восстановлены из сжатого представления без потерь в качестве информации. В этой работе мы рассмотрели несколько методов сжатия данных без потерь, таких как Алгоритм Хаффмана, кодирование Шеннона-Фано, преобразование Барроуза-Уилера и сжатие данных LZW.

Алгоритм Хаффмана используется для построения оптимального префиксного кода, который может быть использован для сжатия данных без потерь. Он основывается на частоте использования каждого символа в сообщении. Он создает двоичное дерево с префиксным кодом, где часто используемые символы имеют короткий код, а редко используемые символы имеют более длинный код.

Кодирование Шеннона-Фано - это другой метод построения префиксного кода, который также использует частоту символов в сообщении. Он строит код таким образом, что более часто встречающиеся символы имеют более короткий код, чем менее часто встречающиеся символы.

Преобразование Барроуза-Уилера - это метод, который переставляет символы в исходном сообщении, создавая новую последовательность символов, которая может быть сжата лучше, чем исходное сообщение. Он используется в сочетании с другими методами сжатия данных для повышения эффективности сжатия.

Сжатие данных LZW - это алгоритм сжатия данных, который используется для сжатия последовательности символов, создавая словарь кодов, которые представляют серии повторяющихся символов. Он обычно используется для сжатия текстовых файлов, где повторяющиеся фразы могут быть сжаты до одного кода.

В ходе курсовой работы было проведено сравнение различных алгоритмов сжатия данных, выявлены плюсы и минусы каждого.

В качестве программного продукта были разработаны алгоритмы сжатия данных

# **СПИСОК ИСТОЧНИКОВ**

1 Алгоритм Хаффмана на пальцах [Электронный ресурс]. – Режим доступа : <https://habr.com/ru/articles/144200/>. – Дата доступа: 01.04.2023.

2 Алгоритм сжатия кода Хаффмана [Электронный ресурс]. – Режим доступа : <https://www.techiedelight.com/ru/huffman-coding/>. – Дата доступа: 02.04.2023.

3 Алгоритм сжатия Хаффмана [Электронный ресурс]. – Режим доступа : <https://habr.com/ru/companies/otus/articles/497566/>. – Дата доступа: 04.04.2023.

4 Кодирование Шеннона-Фано [Электронный ресурс]. – Режим доступа : [http://www.compression.ru/download/articles/huff/tiger\_shannon-fano.html#:~:text=%D0%9A%D0%BE%D0%B4%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%20%D0%A8%D0%B5%D0%BD%D0%BD%D0%BE%D0%BD%D0%B0%2D%D0%A4%D0%B0%D0%BD%D0%BE%20%D1%8F%D0%B2%D0%BB%D1%8F%D0%B5%D1%82%D1%81%D1%8F%20%D0%BE%D0%B4%D0%BD%D0%B8%D0%BC,%D0%BF%D0%BE%D1%8F%D0%B2%D0%B8%D0%BB%D0%BE%D1%81%D1%8C%20%D0%BD%D0%B0%20%D0%BD%D0%B5%D1%81%D0%BA%D0%BE%D0%BB%D1%8C%D0%BA%D0%BE%20%D0%BB%D0%B5%D1%82%20%D0%BF%D0%BE%D0%B7%D0%B6%D0%B5](http://www.compression.ru/download/articles/huff/tiger_shannon-fano.html" \l ":~:text=%D0%9A%D0%BE%D0%B4%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%20%D0%A8%D0%B5%D0%BD%D0%BD%D0%BE%D0%BD%D0%B0%2D%D0%A4%D0%B0%D0%BD%D0%BE%20%D1%8F%D0%B2%D0%BB%D1%8F%D0%B5%D1%82%D1%81%D1%8F%20%D0%BE%D0%B4%D0%BD%D0%B8%D0%BC,%D0%BF%D0%BE%D1%8F%D0%B2). – Дата доступа: 06.04.2023.

5 Алгоритм Шеннона-Фано [Электронный ресурс]. – Режим доступа : <https://habr.com/ru/articles/137766/>. – Дата доступа: 06.04.2023.

6 Алгоритмы, используемые при сжатии данных [Электронный ресурс]. – Режим доступа : <https://habr.com/ru/articles/132289/>. – Дата доступа: 08.04.2023.

7 BWT-преобразование и компрессор [Электронный ресурс]. – Режим доступа : <https://mf.grsu.by/UchProc/livak/po/comprsite/theory_bwt.html>. – Дата доступа: 10.04.2023.

8 Сжатие данных LZW [Электронный ресурс]. – Режим доступа : <https://habr.com/ru/companies/otus/articles/581728/>. – Дата доступа: 08.04.2023.

9 Алгоритм LZW [Электронный ресурс]. – Режим доступа : <https://mf.grsu.by/UchProc/livak/po/comprsite/theory_lzw.html>. – Дата доступа: 10.04.2023.

# **ПРИЛОЖЕНИЕ А**

**(обязательное)**

**Листинг кода алгоритма Шеннона-Фано**

class node :

def \_\_init\_\_(self) -> None:

self.sym=''

self.pro=0.0

self.arr=[0]\*20

self.top=0

p=[node() for \_ in range(20)]

def shannon(l, h, p):

pack1 = 0; pack2 = 0; diff1 = 0; diff2 = 0

if ((l + 1) == h or l == h or l > h) :

if (l == h or l > h):

return

p[h].top+=1

p[h].arr[(p[h].top)] = 0

p[l].top+=1

p[l].arr[(p[l].top)] = 1

return

else :

for i in range(l,h):

pack1 = pack1 + p[i].pro

pack2 = pack2 + p[h].pro

diff1 = pack1 - pack2

if (diff1 < 0):

diff1 = diff1 \* -1

j = 2

while (j != h - l + 1) :

k = h - j

pack1 = pack2 = 0

for i in range(l, k+1):

pack1 = pack1 + p[i].pro

for i in range(h,k,-1):

pack2 = pack2 + p[i].pro

diff2 = pack1 - pack2

if (diff2 < 0):

diff2 = diff2 \* -1

if (diff2 >= diff1):

break

diff1 = diff2

j+=1

k+=1

for i in range(l,k+1):

p[i].top+=1

p[i].arr[(p[i].top)] = 1

for i in range(k + 1,h+1):

p[i].top+=1

p[i].arr[(p[i].top)] = 0

shannon(l, k, p)

shannon(k + 1, h, p)

def sortByProbability(n, p):

temp=node()

for j in range(1,n) :

for i in range(n - 1) :

if ((p[i].pro) > (p[i + 1].pro)) :

temp.pro = p[i].pro

temp.sym = p[i].sym

p[i].pro = p[i + 1].pro

p[i].sym = p[i + 1].sym

p[i + 1].pro = temp.pro

p[i + 1].sym = temp.sym

def display(n, p):

print("\n\n\n\tSymbol\tProbability\tCode",end='')

for i in range(n - 1,-1,-1):

print("\n\t", p[i].sym, "\t\t", p[i].pro,"\t",end='')

for j in range(p[i].top+1):

print(p[i].arr[j],end='')

# **ПРИЛОЖЕНИЕ Б**

**(обязательное)**

**Листинг кода алгоритма LZW**

def compress(uncompressed):

"""Compress a string to a list of output symbols."""

# Build the dictionary.

dict\_size = 256

dictionary = dict((chr(i), chr(i)) for i in range(dict\_size))

# in Python 3: dictionary = {chr(i): chr(i) for i in range(dict\_size)}

w = ""

result = []

for c in uncompressed:

wc = w + c

if wc in dictionary:

w = wc

else:

result.append(dictionary[w])

# Add wc to the dictionary.

dictionary[wc] = dict\_size

dict\_size += 1

w = c

# Output the code for w.

if w:

result.append(dictionary[w])

return result

def decompress(compressed):

"""Decompress a list of output ks to a string."""

# Build the dictionary.

dict\_size = 256

dictionary = dict((chr(i), chr(i)) for i in range(dict\_size))

# in Python 3: dictionary = {chr(i): chr(i) for i in range(dict\_size)}

w = result = compressed.pop(0)

for k in compressed:

if k in dictionary:

entry = dictionary[k]

elif k == dict\_size:

entry = w + w[0]

else:

raise ValueError('Bad compressed k: %s' % k)

result += entry

# Add w+entry[0] to the dictionary.

dictionary[dict\_size] = w + entry[0]

dict\_size += 1

w = entry

return result

# **ПРИЛОЖЕНИЕ В**

**(обязательное)**

**Листинг кода алгоритма Хаффмана**

import heapq

from heapq import heappop, heappush

def isLeaf(root):

return root.left is None and root.right is None

class Node:

def \_\_init\_\_(self, ch, freq, left=None, right=None):

self.ch = ch

self.freq = freq

self.left = left

self.right = right

def \_\_lt\_\_(self, other):

return self.freq < other.freq

def encode(root, s, huffman\_code):

if root is None:

return

if isLeaf(root):

huffman\_code[root.ch] = s if len(s) > 0 else '1'

encode(root.left, s + '0', huffman\_code)

encode(root.right, s + '1', huffman\_code)

def decode(root, index, s):

if root is None:

return index

if isLeaf(root):

print(root.ch, end='')

return index

index = index + 1

root = root.left if s[index] == '0' else root.right

return decode(root, index, s)

def buildHuffmanTree(text):

if len(text) == 0:

return

freq = {i: text.count(i) for i in set(text)}

pq = [Node(k, v) for k, v in freq.items()]

heapq.heapify(pq)

while len(pq) != 1:

left = heappop(pq)

right = heappop(pq)

total = left.freq + right.freq

heappush(pq, Node(None, total, left, right))

root = pq[0]

huffmanCode = {}

encode(root, '', huffmanCode)

print('Huffman Codes are:', huffmanCode)

print('The original string is:', text)

s = ''

for c in text:

s += huffmanCode.get(c)

print('The encoded string is:', s)

print('The decoded string is:', end=' ')

if isLeaf(root):

while root.freq > 0:

print(root.ch, end='')

root.freq = root.freq - 1

else:

index = -1

while index < len(s) - 1:

index = decode(root, index, s)

# **ПРИЛОЖЕНИЕ Г**

**(обязательное)**

**Листинг кода алгоритма BWT**

Input = 'banana'

assert "$" not in Input

Input = Input + "$"

table = [Input[i:] + Input[:i] for i in range(len(Input))]

table = sorted(table)

last\_column = [row[-1:] for row in table]

bwt = ''.join(last\_column)

print(bwt)

table = [""] \* len(bwt)

for i in range(len(bwt)):

table = [bwt[i] + table[i] for i in range(len(bwt))]

print('unsorted = ', table)

table = sorted(table)

print('sorted =', table)

inverse\_bwt = [row for row in table if row.endswith("$")][0] # Find the correct row (ending in $)

inverse\_bwt = inverse\_bwt.rstrip("$") # Get rid of start and end markers

print(inverse\_bwt)