Е. Р. ПАНТЕЛЕЕВ, А. Л. АЛЫКОВА

АЛГОРИТМЫ СЖАТИЯ ДАННЫХ БЕЗ ПОТЕРЬ

Учебное пособие
Издание второе, стереотипное

ЛАНЬ
• САНКТ-ПЕТЕРБУРГ • МОСКВА • КРАСНОДАР •
• 2022 •

УДК 004.627
ББК 32.9я73

П 16 **Пантелеев Е. Р.** Алгоритмы сжатия данных без потерь : учебное пособие для вузов / Е. Р. Пантелеев, А. Л. Алыкова. — 2-е изд., стер. — Санкт-Петербург : Лань, 2022. — 172 с. : ил. — Текст : непосредственный.

ISBN 978-5-507-45062-6

Учебное пособие содержит описание алгоритмов сжатия данных без потерь, включающее классификацию этих алгоритмов, их обсуждение на концептуальном уровне и на уровне программной реализации, сравнительный анализ результатов их практического применения, рекомендации по выполнению курсового проекта по данной теме. Также обсуждаются смежные вопросы: особенности работы с двоичными данными, формирования заголовочной части сжатого файла, применение вспомогательных алгоритмов, повышающих эффективность сжатия, и объектно-ориентированного подхода к реализации алгоритмов сжатия.

Пособие предназначено для бакалавров направления «Программная инженерия».

УДК 004.627
ББК 32.9я73

Обложка
П. И. ПОЛЯКОВА

Содержание

Предисловие

Учебное пособие охватывает содержание раздела «Алгоритмы сжатия данных» дисциплины «Алгоритмы и структуры данных», которая читается для бакалавров, обучающихся по направлению подготовки 09.03.04 «Программная инженерия» по профилю «Разработка программно-информационных систем». Дисциплина «Алгоритмы и структуры данных» разработана в Ивановском государственном энергетическом университете (ИГЭУ) на кафедре программного обеспечения компьютерных систем. Рабочая программа дисциплины предусматривает выполнение курсового проекта по разделу «Алгоритмы сжатия данных».

Освоение материала учебного пособия предполагает знание алгоритмических языков, наличие навыков написания объектно-ориентированных программ, а также формируемых в ходе изучения предшествующих разделов дисциплины «Алгоритмы и структуры данных» умений определять и использовать абстрактные типы данных и строить порядковые оценки вычислительной эффективности алгоритмов.

Цель учебного пособия — сформировать теоретическую базу, необходимую для реализации алгоритмов сжатия данных без потерь информации, и продемонстрировать возможности практического применения этих знаний для самостоятельной реализации алгоритмов сжатия в ходе выполнения курсового проекта по дисциплине. Для достижения этих целей в пособии обсуждается классификация алгоритмов сжатия, модели кодирования, лежащие в основе алгоритмов сжатия без потерь, сами алгоритмы, как на концептуальном уровне, так и на уровне их программной реализации, а также практические рекомендации по выполнению курсового проекта.

Убедиться в том, что алгоритм сжатия выполняет свою главную функцию, то есть уменьшает объем входных данных — естественное желание разработчика. Поэтому в структуре пособия предусмотрены разделы, содержащие результаты испытаний и их обсуждение. Для чистоты эксперимента использовались файлы, заведомо не являющиеся продуктом применения алгоритмов сжатия. Это монохромные растровые файлы формата bmp (последовательность точек экранного изображения), текстовые файлы (последовательность символьных строк переменной

длины) и файлы исполняемого формата (управляющая информация для загрузчика и образ памяти задачи). Приведенное обсуждение результатов испытаний не претендует на общность выводов, касающихся эффективности того или иного алгоритма — для этого объем испытаний явно недостаточен. Авторы и не ставили перед собой такой задачи. Скорее, эти испытания являются поводом для рассуждений о влиянии тех или иных параметров настройки на эффективность алгоритмов сжатия.

В основу учебного пособия положены материалы читаемого в ИГЭУ одним из авторов лекционного курса по дисциплине «Алгоритмы и структуры данных», а также целый ряд отечественных и зарубежных публикаций по этой теме. В их числе — электронный ресурс «Всё о сжатии данных, изображений и видео» (**http://compression.ru/**), специализирующийся на сборе и публикации материалов по теме. Хотелось бы особо выделить монографию [1] «The Data Compression Book» — книга о сжатии данных, которая удачно сочетает доступное изложение концепций алгоритмов сжатия с глубоким обсуждением тонкостей программной реализации этих концепций на языке С. Однако с момента выхода этой книги технологии программирования заметно усовершенствовались, а аппаратные возможности компьютеров выросли на порядок. Это делает актуальной попытку увидеть старые проблемы с новых позиций. Круг рассматриваемых в учебном пособии вопросов ограничен обсуждением алгоритмов сжатия данных без потерь информации. Это ограничение, а также отбор алгоритмов для детального рассмотрения, продиктовано ограниченным объемом учебного курса и не нарушает логику изучения темы.

Все программы, сопровождающие обсуждение алгоритмов сжатия, написаны на языке C++. Этот выбор обусловлен тем, что языки C/C++ являются базой для целого семейства C-подобных языков программирования. В их число входят Java, C#, Python и др. Следовательно, программный код должен быть понятен широкому кругу читателей.

В тексте пособия приводятся только ключевые фрагменты программного кода, демонстрирующие особенности реализации алгоритмов. Необходимые для полноценной реализации вспомогательные алгоритмы приведены в Приложении в разделе П.2, в разделах П.3 и П.4 представлено описание программного кода. Весь программный код реализован в среде Visual Studio 2019.

1. Введение

Words are like leaves; and where they most abound,
Much fruit of sense beneath is rarely found[1].

Alexander Pope

В течение последних десятилетий человечество создало столько же данных, сколько за всю предыдущую историю цивилизации. При этом темпы роста объемов хранимых данных продолжают увеличиваться: их удвоение происходит, по разным оценкам, каждые 4 или 10 лет. Одновременно с ростом объемов хранения увеличиваются и объемы передачи данных: количество интернет-сайтов, основных потребителей сетевой информации, с 1995 по 2007 гг. ежегодно удваивалось [2]. Разумеется, аппаратные возможности хранения и передачи данных также растут. Емкость устройств внешней памяти домашних компьютеров измеряется не мегабайтами, как было 20 лет назад, и не гигабайтами, как это было 10 лет назад, а терабайтами. Скорость обмена с внешними устройствами за этот же период выросла на два-три порядка. Скорость передачи данных по сети Интернет сегодня составляет десятки и сотни мегабит в секунду, тогда как еще 10 лет назад она была на порядок меньше. Однако рост аппаратных возможностей компьютеров и сетей передачи не поспевает за ростом лавины данных. Для борьбы с этой проблемой необходимы алгоритмы и программное обеспечение сжатия (уменьшения объема) сохраняемых данных и распаковки сжатых данных для последующего использования[2].

Количественный эффект сжатия можно оценивать по-разному (количеством сжатых бит на исходный байт, отношением исходного объема к сжатому). Будем называть степенью сжатия *R* разность исходного *S* и результирующего *D* объемов, отнесенную к исходному объему и выраженную в процентах [1]:

$$R = \frac{S - D}{S} * 100. \quad (1)$$

---

[1] Слова подобны листьям: где они изобильны, там мало плодов смысла.

[2] К сожалению, существует и деструктивный вариант использования алгоритмов сжатия для создания так называемой zip-бомбы (decompression bomb): сжатый по специальной схеме файл при распаковке заполняет всю свободную память и вызывает крах системы.

Анализ этого выражения показывает, что степень сжатия может быть как положительной, в случае уменьшения объема входного потока, так и отрицательной, в случае его увеличения.

Возможность сжатия обусловлена двумя свойствами данных. Во-первых, они обладают информационной избыточностью. Простой пример избыточности — представление изображения в формате bmp. Этот формат представляет изображение в виде линейной последовательности пикселей (растра). Цветовые атрибуты следующих друг за другом пикселей могут совпадать, что создает соблазн кодирования такой цепочки счетчиком ее длины с указанием значения повторяющегося цветового атрибута. В результате объем хранимых (передаваемых) данных может существенно уменьшиться. Например, если длина цепочки повторяющихся пикселей равна 100, а каждый пиксель кодируется тремя байтами цветовых атрибутов, то 300 байт исходного представления можно заменить четырьмя, один из которых будет хранить значение счетчика в формате байта, а три оставшихся — значение повторяющегося цветового атрибута. Бытовая аналогия сжатия данных — хранение сезонной одежды в вакуумных пакетах. И в том, и в другом случае мы увеличиваем «плотность» хранимых объектов и уменьшаем затраты на хранение «воздуха»[3].

Во-вторых, часть содержащейся в данных информации просто-напросто не нужна потребителю. Например, ухо среднего человека не воспринимает звуки с частотой менее 20 Гц и более 20 КГц, а также пространственную структуру звука (стереоэффект) в области низких частот. Следовательно, соответствующую информацию при сжатии аудиофайла можно игнорировать.

Разумеется, за кодирование, приводящее к уменьшению объема данных, придется заплатить увеличением вычислительных затрат на их декодирование, но эти затраты практически всегда остаются на приемлемом уровне. Достаточно вспомнить, например, про видео- и аудиоданные, передаваемые по сети Интернет. Эти данные возникли в результате применения весьма изощренных алгоритмов сжатия, однако при их просмотре (прослушивании) в браузере не возникает ощущения сколько-нибудь заметной потери качества и/или задержек в воспроизведении.

---

[3] Эффект уменьшения объема хранения может возникнуть не только в результате увеличения плотности объектов хранения, но и в результате их рациональной упаковки (с этой же целью, например, культивируются «квадратные» арбузы). Соответственно, различают алгоритмы сжатия и алгоритмы упаковки данных, хотя в коммерческих продуктах они, как правило, используются совместно.

В силу всего вышесказанного, изучение алгоритмов сжатия данных является важным элементом подготовки специалиста по информационным технологиям. Знание этих алгоритмов обеспечит, как минимум, осмысленное использование готовых решений в этой области, а опыт их практической разработки позволит оценить расстояние от сравнительно простой базовой идеи алгоритма до ее эффективной реализации, а также, возможно, пробудит интерес к самостоятельным экспериментам в этой области.

Понятно, что масштабы проблемы компактного представления данных порождают гигантское многообразие подходов к ее решению, поэтому попытка рассказать обо всем и сразу заранее обречена на неудачу. Однако в любом случае следует начать разговор с определения оснований, по которым это многообразие можно классифицировать, с тем, чтобы выбрать обоснованную «точку входа» в пространство алгоритмов сжатия и использовать структуру, полученную в результате классификации, в качестве навигатора при дальнейшем самостоятельном освоении этого пространства.

Главный классификационный признак — это обратимость процедуры сжатия. Ответ на вопрос, существует ли процедура (декодер), обратная процедуре сжатия (кодеру), которая возвращает сжатые данные в их исходное состояние, делит множество алгоритмов сжатия на те, для которых этот ответ положительный, и те, для которых ответ отрицателен. Первое подмножество — это алгоритмы сжатия без потерь информации. Отсутствие потерь — необходимое и достаточное условие обратимости сжатия. Второе подмножество — алгоритмы сжатия данных с потерями. Приемлемость потерь при сжатии данных определяется тем, кто или что является конечным потребителем результатов их декодирования. Если конечным потребителем является компьютер — потери недопустимы. Представьте себе ситуацию, когда сжатый файл исполняемого кода программы декодирован с потерями. В результате компьютер не сможет запустить его на выполнение. Это полная потеря функциональности, для которой данный файл изначально и создавался. Если мы допускаем сжатие с потерями для файлов бухгалтерской базы данных, последствия декодирования окажутся еще более катастрофическими: вся информация о финансовой деятельности предприятия будет безвозвратно утеряна! Другое дело, если объектом сжатия являются оцифрованные аналоговые данные, а в качестве потребителя декодированных данных выступает человек. В силу физиологических особенностей своего восприятия, чело-

век, в частности, «не слышит» звуков вне диапазона 20–20 000 Гц, «не видит» плавных переходов цвета в плоскости изображения и вообще лучше воспринимает поступающую от изображения яркостную составляющую сигнала, нежели цветовую — «в темноте все кошки серы». Если эта информация остается невостребованной потребителем — значит, ей можно безболезненно пожертвовать ради уменьшения объема хранимых (передаваемых) данных. Именно это и происходит в результате применения алгоритмов сжатия с потерями, которые, в силу названных выше причин, также называют алгоритмами *перцептивного кодирования* (perceptual coding), от английского «perception», что значит «восприятие»[4]. Есть несколько причин отложить детальное знакомство с этими алгоритмами на потом. Во-первых, их математическая основа чрезвычайно сложна. Во-вторых, каждый из них имеет дело с тонкостями физиологического восприятия конкретного вида сигналов (аудио-, видео-, статических изображений). И, наконец, в-третьих, алгоритмы сжатия данных с потерями для исключения информационной избыточности используют алгоритмы сжатия без потерь. Например, при сжатии изображений с потерями в формате jpg используются алгоритмы сжатия без потерь RLE и Хаффмана, алгоритм сжатия без потерь LZW использован при создании формата обмена графикой GIF. Именно поэтому в рамках данного учебного пособия в дальнейшем будут обсуждаться алгоритмы сжатия данных без потерь (далее — просто алгоритмы сжатия). Во-первых, они концептуально проще. Во-вторых, эти алгоритмы более универсальны, так как не привязаны к конкретному типу потока данных. Хотя знание свойств этого потока полезно для достижения более высокой степени сжатия, оно не является обязательным условием реализации алгоритмов. И, наконец, в-третьих, без них невозможна полноценная реализация алгоритмов сжатия с потерями.

Если алгоритмы сжатия данных не допускают потерь информации, то единственным способом уменьшения объема данных при сжатии является уменьшение их информационной избыточности. По способу

---

[4] Сжатие информации, предназначенной для восприятия человеком, не всегда сопровождается потерями. Так, например, сжатый формат изображений jpg уменьшает исходный объем данных за счет частичной потери информации в результате дискретного квантования яркости изображения, в то время как предназначенный для той же цели формат png обеспечивает сжатие без потерь. То же самое можно сказать по поводу форматов сжатия звука mp3 (сжатие с потерями) и flac (сжатие без потерь).

решения этой задачи алгоритмы сжатия подразделяются на частотные и словарные[5]. Обсудим эти подходы, приняв для удобства допущение, состоящее в том, что входной поток данных кодера, если явно не оговорено иное, — это поток символов в формате байта. Это допущение не снижает общности рассуждений, так как в компьютерном представлении символ — это его цифровой код. Для кодера безразлично, будет ли этот код интерпретирован как символ, число или какой-либо другой объект более высокого уровня абстракции. С другой стороны, принятие этого допущения делает обсуждение алгоритмов кодирования и декодирования более наглядным.

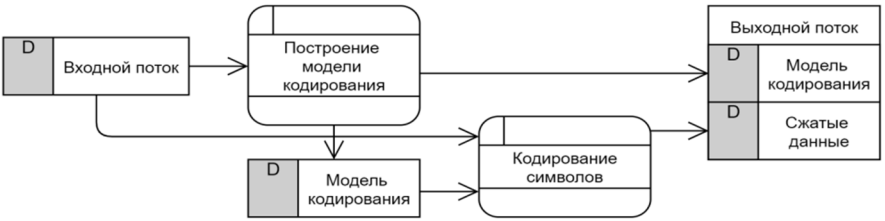
Принцип уменьшения информационной избыточности, реализованный в частотных алгоритмах, состоит в том, что чем выше частота символа во входном потоке (следовательно, чем меньше информации этот символ несет), тем меньше должна быть длина кода этого символа в выходном (сжатом) потоке. И наоборот, чем ниже частота символа во входном потоке, тем больше должна быть длина кода этого символа в выходном потоке. Таким образом, эффект сжатия возникает из-за того, что на хранении в выходном потоке коротких кодов часто встречающихся символов мы выигрываем больше, чем теряем на хранении длинных кодов редко встречающихся символов.

Иной подход к уменьшению информационной избыточности реализуют алгоритмы сжатия словарной группы. Эффект сжатия здесь достигается в результате замены цепочки символов произвольной длины во входном потоке уникальным кодом фиксированной длины для этой цепочки в выходном потоке. Понятно, что чем чаще цепочка повторяется во входном потоке, тем большей экономии за счет использования замещающего ее в выходном потоке кода можно достигнуть.

Разумеется, между символами (цепочками символов) входного потока кодера и кодами выходного потока должно существовать взаимно-однозначное соответствие, которое известно как кодеру, так и декодеру. Наличие этого соответствия обеспечивает обратимость сжатия и отсутствие информационных потерь. Будем называть это соответствие, а также правила его интерпретации парой кодер/декодер в рамках конкретного алгоритма *моделью кодирования*.

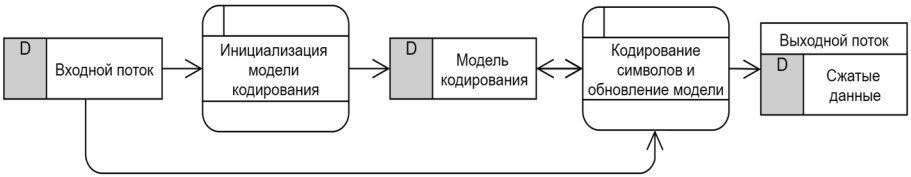
---

[5] Это не означает строгого подразделения алгоритмов сжатия на два класса по данному признаку. Существуют «гибриды». Например, алгоритм сжатия Deflate, используемый архиватором PKZIP, комбинирует словарный принцип сжатия с частотным.

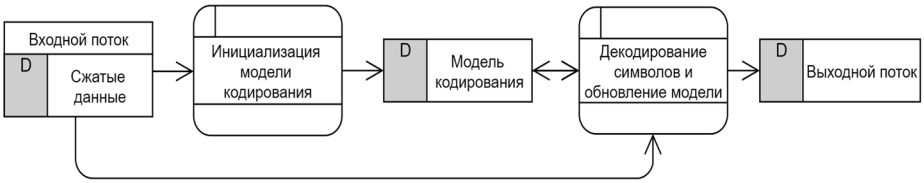


а) Статический кодер

б) Статический декодер



в) Динамический кодер



г) Динамический декодер

Рис. 1
Классификация алгоритмов сжатия по принципу формирования модели кодирования

По способу формирования этой модели алгоритмы кодирования/декодирования подразделяются на *статические* и *динамические*. Различия между этими двумя способами показаны на рис. 1 в формате диаграмм потоков данных (DFD). Кодер статического алгоритма формиру-

ет модель кодирования путем сканирования входного потока *до начала* процесса кодирования (рис. 1*а*) и передает ее декодеру в заголовке файла сжатых данных (рис. 1*б*). Кодер и декодер динамического алгоритма (рис. 1*в, г*) строят модель кодирования «на лету», т. е. непосредственно в процессе сжатия или декомпрессии. Они используют для этого согласованные процедуры инициализации и обновления модели кодирования. Процедура инициализации формирует начальное состояние модели кодирования, а процедура обновления адаптирует эту модель к текущему состоянию входного потока данных кодирования/декодирования. По этой причине динамическое кодирование также называют адаптивным. При таком подходе передача модели кодирования от кодера декодеру в заголовке файла сжатых данных уже не нужна. Поскольку в конечном счете целью сжатия является уменьшение занимаемого файлом объема памяти на внешнем носителе, отсутствие необходимости передавать модель кодирования, которая увеличивает объем файла сжатых данных, — безусловное преимущество динамических алгоритмов. С другой стороны, тот факт, что модель кодирования, учитывающая все существенные для сжатия свойства входного потока, известна кодеру и декодеру априори, обеспечивает достижение более высокой степени сжатия по сравнению с динамическими алгоритмами, которым информация о свойствах входного потока становится доступной по мере работы кодера и декодера.

1.1. Вопросы для самоконтроля

1. Какими свойствами данных обусловлена возможность их сжатия?
2. Назовите основной признак, по которому классифицируются алгоритмы сжатия.
3. Почему алгоритмы сжатия данных с потерями информации иногда называют алгоритмами перцептивного кодирования?
4. Какие группы методов уменьшения информационной избыточности используют алгоритмы сжатия данных без потерь?
5. Что такое модель кодирования?
6. Сформулируйте принцип уменьшения информационной избыточности, лежащий в основе словарных методов сжатия данных.
7. Сформулируйте принцип уменьшения информационной избыточности, лежащий в основе частотных методов сжатия данных.

8. В чем отличие статических и динамических алгоритмов сжатия данных?
9. В чем заключаются преимущества и недостатки статических алгоритмов сжатия данных?
10. В чем заключаются преимущества и недостатки динамических алгоритмов сжатия данных?

2. Словарные алгоритмы сжатия

Прочтя в черноморской вечерке объявление «Сд. пр. ком. в. уд. в. н. м. од. ин. хол.» и мигом сообразив, что объявление это означает — «Сдается прекрасная комната со всеми удобствами и видом на море одинокому интеллигентному холостяку», Остап подумал: «Сейчас я, кажется, холост».

И. Ильф, Е. Петров. «Золотой теленок»

Подход к минимизации информационной избыточности, реализованный алгоритмами словарной группы, основан на замене повторяющихся цепочек символов во входном потоке кодера кодами этих цепочек в выходном потоке. Декодер выполняет обратную операцию. Эффект сжатия достигается за счет того, что длина кода цепочки фиксирована и, как правило, меньше длины самой цепочки. Отличие между обсуждаемыми далее алгоритмами заключается в трактовке понятия «цепочка» и используемой модели кодирования.

2.1. Алгоритм кодирования длин серий (RLE)

Алгоритм RLE (Run-Length Encoding — кодирование длин серий) — самый старый и простейший в группе словарных алгоритмов сжатия. Еще в 1967 году он применялся для сжатия телевизионных сигналов при передаче[6]. При этом степень сжатия, которую обеспечивает RLE, в некоторых случаях может быть достаточно высокой. Под цепочкой алгоритм RLE понимает серию повторяющихся символов, а под моделью кодирования (кодом цепочки) — пару «длина серии — символ серии». Правила построения этой модели кодером и ее интерпретации декодером известны до начала кодирования/декодирования, сама же модель формируется непосредственно в ходе этих процессов. Поэтому можно утверждать, что алгоритм RLE использует динамическую модель кодиро-

[6] В соответствии с ГОСТ Р 54998-2012 «Цифровая система телевидения высокой четкости. Кодирование цифровых телевизионных сигналов для сжатия цифрового потока», RLE используется в составе системы кодирования видеосигнала по стандарту MPEG-2.

вания. Согласно этой модели, кодер превращает входной поток «АААААААААААББББББВВВГ» в «12А6Б3В1Г», а декодер возвращает сжатый поток в исходное состояние. Этот пример показывает, что чем длиннее серии, тем выше степень сжатия входного потока кодера. Если все серии имеют единичную длину, объем сжатого файла превышает объем исходного. Наличие длинных серий наиболее характерно для двухцветных (например, черно-белых) изображений растрового формата bmp. В этом формате изображение хранится в виде последовательности точек растра, каждая из которых содержит либо значение цвета, либо его индекс в палитре — структуре данных в заголовочной части файла. Напротив, если серии короткие, что характерно для bmp-файлов полноцветных изображений с плавными цветовыми переходами, текстовых или двоичных файлов, применение алгоритма RLE может привести к нежелательному эффекту увеличения объема.

2.1.1. Байт-ориентированный алгоритм

Все, что возможно считать, – считайте!
Фрэнсис Гальтон

Алгоритм очень прост в реализации. Кодер в цикле считывает очередной байт byte_curr из входного потока fin и сравнивает его с предыдущим байтом byte_pred. Если эти два значения совпадают, текущая длина цепочки в счетчике count увеличивается на 1.

Для хранения длины цепочки кодер использует формат, согласованный с декодером — например байт, слово, двойное слово. Выбор формата зависит от свойств входного потока: чем длиннее цепочки — тем больше памяти требуется для хранения длины. Статистику длин цепочек можно собрать путем предварительного анализа входного потока. Ясно, что обоснованный выбор формата положительно влияет на степень его сжатия, но эта тема не входит в круг задач данного пособия. В обсуждаемой реализации кодер и декодер используют формат байта. При необходимости это допущение можно легко изменить.

Как только цепочка обрывается или ее длина превышает значение, которое может быть записано в формате байта, значения count и

byte_pred записываются в выходной поток fout. Затем счетчик длины сбрасывается в 1, а значение предыдущего байта обновляется значением текущего. По завершении цикла остается только записать информацию о последней цепочке.

```
fread(&byte_pred, 1, 1, fin);
while (fread(&byte_curr, 1, 1, fin))
{
  if (byte_pred == byte_curr && count < 255)
    count++;
  else
  {
    fwrite(&count, sizeof(count), 1, fout);
    fwrite(&byte_pred, sizeof(byte_pred), 1, fout);
    count = 1;
    byte_pred = byte_curr;
  }
}
```

Обратный процесс декодирования файла также несложен. В цикле считывается длина цепочки count и байт, образующий цепочку, byte. Затем цепочка байтов записывается в выходной поток.

```
while (fread(&count, 1, 1, fin))
  {
    fread(&byte, 1, 1, fin);
    for (unsigned char i = 0; i < count; i++)
      fwrite(&byte, sizeof(byte), 1, fout);
  }
```

К сожалению, зависимость степени сжатия, которую обеспечивает алгоритм RLE, от длины серий серьезно ограничивает область его эффективного применения. Поэтому поиск способов уменьшения этой зависимости является актуальной задачей. Проблему можно решить, если найти такое обратимое преобразование, прямое применение которого к входному потоку способствовало бы увеличению длины серий для кодера, а обратное — обеспечивало восстановление первоначальных серий в декодированном потоке. Этим требованиям отвечает преобразование Бэрроуза — Уилера (Burrows–Wheeler Transform: BWT) [3], описанное в разделе П.2.3 Приложения.

2.1.2. Бинарный алгоритм

… сумма нулей — грозная цифра!
Станислав Ежи Лец

Алгоритм RLE-кодирования можно настроить на обработку элементов входного потока, имеющих разрядность битов, байтов или слов[7]. RLE-кодирование, настроенное на обработку слов, принципиально не отличается от рассмотренной выше байт-ориентированной версии. Побитовое кодирование, которое называют двоичным, или бинарным, имеет дело с потоком нулей и единиц и может позволить себе строить цепочки только для одного двоичного значения, а цепочки элементов, имеющих противоположное значение, кодировать неявно. Это существенное отличие от версий, работающих с элементами входного потока большей разрядности, поэтому далее будет обсуждаться программная реализация бинарного RLE-кодирования.

Учитывая упомянутую особенность двоичной системы счисления, кодер руководствуется следующими правилами.

1. Кодируются только серии нулей.
2. Серии единиц декодеру не передаются.
3. Последовательные значения единиц разделяются кодами серий нулевой длины.

Для программной реализации алгоритма необходимо определиться с разрядностью хранения длины серии *k*. Как уже говорилось, выбор разрядности влияет на обеспечиваемую алгоритмом степень сжатия. Предположим, для определенности, что *k* = 4 битам (серии, длина кото-

---

[7] **Машинное слово** — машинно-зависимая и платформо-зависимая величина, измеряемая в битах или байтах (тритах или трайтах), равная разрядности регистров процессора и/или разрядности шины данных (обычно некоторая степень двойки). В ранних ЭВМ размер слова совпадал с минимальным размером адресуемой информации. В современных машинах минимальным адресуемым блоком информации называется байт, а слово состоит из нескольких байтов (Википедия — свободная энциклопедия, https://ru.wikipedia.org/wiki/).

рых больше $L = 2^4 - 1 = 15$, следует разбить на группы). Тогда кодер превращает входной поток в структуру данных, показанную на рис. 2[8].

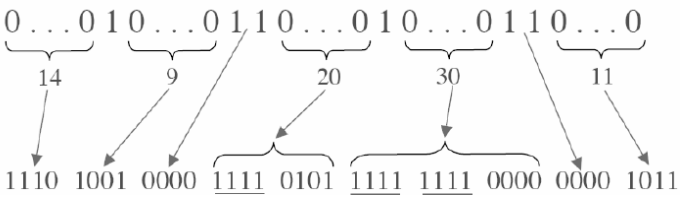


Рис. 2. Пример работы двоичного кодера RLE

Обратите внимание, что код группы 15 $(1111)_2$ означает, что эта группа не последняя в серии. Если длина серии кратна $2^k - 1$, последовательность кодов входящих в нее групп завершается нулевым кодом. Преимуществом двоичного RLE является то, что ввиду «бинарности» входного потока нет необходимости передавать декодеру значения символов, образующих серию: согласно правилам 1–2, кодируются только серии нулей. Это способствует уменьшению выходного потока и позволяет легко построить гипотетические оценки степени сжатия для «лучшего» и «худшего», с точки зрения двоичного RLE, случая.

В лучшем случае входной поток состоит только из нулей («черный квадрат», если речь идет о растровом изображении). Тогда, обозначив длину (количество бит) входного потока через *S*, легко определить длину *D* выходного потока:

$$D = \frac{Sk}{2^k - 1}.$$

Применив формулу (1), получим выражение для степени сжатия:

$$R = \left(1 - \frac{k}{2^k - 1}\right) * 100.$$

Для $k = 4$ степень сжатия $R = 73\%$, что соответствует приблизительно четырехкратному уменьшению объема исходного файла.

В худшем случае («белый квадрат») во входном потоке присутствуют только единицы. Тогда каждая единица на входе кодера замещается *k* нулевыми битами, где *k* — длина кода серии. Таким образом, не-

---

[8] Заимствован на http://web.engr.oregonstate.edu/~thinhq/teaching/ece499/spring06/runlength_turnstall_golomb.pdf.

производительно расходуется *k–1* бит. Следовательно, выходной поток увеличивается в *k* раз по сравнению со входным. Согласно формуле (1), степень сжатия для *k* = 4 в данном случае будет равна –300%. То есть, сжатый файл по отношению к исходному увеличится в объеме в четыре раза.

Очевидно, что с ростом *k* и максимальный проигрыш, и максимальный выигрыш будут возрастать.

2.1.2.1. Бинарный кодер

Несмотря на то что бинарный кодер оперирует битами, наименьшей единицей информации, доступной при чтении входного потока, является байт. Поэтому основной цикл кодирования обеспечивает чтение байта из входного потока fin в переменную byte_in. Каждый байт последовательно разбивается на биты (переменная bit), начиная со старшего. Таким образом, кодер получает непрерывную цепочку битов и анализирует их значения, выделяя серии нулей. При необходимости серии разбиваются на группы. Для подсчета количества подряд идущих нулей используется счетчик count_0. Если его значение становится равным максимальной длине группы (в нашем случае 15), группа добавляется в выходной поток. Если серия нулей прерывается единицей, добавляется группа длиной count_0, причем если предшествующая единице серия закончилась полной группой, то добавляется группа нулевой длины. Нулевая группа добавляется и между единицами. Основной цикл можно представить следующим кодом:

```
while (fread(&byte_in, 1, 1, fin))
  { // байт из входного потока считан
    for (bit_in = 0; bit_in < 8; bit_in++)
    {
      bit = byte_in >> 7;  // выделение бита
      byte_in <<= 1;
      if (!bit)      // если бит = 0
      {
        count_0++;
        if (count_0 == 15)
        // серия нулей достигла максимальной длины
          Addgroup(); // запись группы нулей
      }
      else  // если бит = 1
```

```
      Addgroup(); // запись группы нулей
    }
  }
```

Функция добавления группы Addgroup() обеспечивает добавление очередной группы длиной count_0 к выходному коду. Поскольку длина группы не может превышать 15, значение count_0 можно разместить в четырех битах, а в один байт byte_out упаковать две группы. Если полный байт сформирован, он записывается в выходной поток fout. Ниже приведен код функции Addgroup().

```
void Addgroup()
{
  byte_out = (byte_out << 4) + count_0;
  bit_out += 4;  // количество битов в выходном байте
  count_0 = 0;
  if (bit_out == 8) // сформирован байт кода
  {
    fwrite(&byte_out, 1, 1, fout);
    bit_out = 0;
    byte_out = 0;
  }
}
```

По завершении основного цикла кодирования последняя группа нулей может оказаться незаписанной в выходной поток. Такая ситуация возникает в том случае, если входной поток заканчивается неполной группой нулей. Кроме того, незаписанным остается и не полностью сформированный выходной байт. Чтобы избежать потери данных, в кодер включен следующий код:

```
// запись последней группы нулей
 if (count_0 || bit_out)
 {
   if (bit_out)
     byte_out = (byte_out << 4) + count_0;
   else
   {
     byte_out = count_0;
     byte_out <<= 4;
   }
   fwrite(&byte_out, 1, 1, fout);
 }
```

2.1.2.2. Бинарный декодер

Бинарный декодер считывает текущий байт входного потока, выделяет в нем текущую группу и декодирует ее в соответствии со значением ее длины len. Если длина ненулевая, декодер добавляет в выходной поток соответствующее количество нулевых битов функцией Addbit. Если группа неполная или имеет нулевую длину, то, согласно логике кодирования, декодер добавляет бит, равный единице.

```
while (true)
{
  if (bit_in == 0) // нет необработанных битов
    if (fread(&byte_in, 1, 1, fin))
      bit_in = 8;       // число необработанных битов
    else
      break;
  bit_in -= 4;
  char len = byte_in >> 4;  // выделение из байта длины группы
  byte_in <<= 4;
  if (len)               // обработка группы ненулевой длины
    for (char i = 0; i < len; i++)
      Addbit(0);
  if (len < 15)
      Addbit(1);
}
```

Функция Addbit выполняет операцию добавления бита, передаваемого параметром, к выходному байту и проверяет количество битов в нем. Если байт полностью сформирован, функция записывает его в выходной поток.

```
void Addbit(unsigned char bit)
 {
   byte_out = (byte_out << 1) + bit;
   bit_out++;
   if (bit_out == 8)   // сформирован байт
   {
     fwrite(&byte_out, 1, 1, fout);
     bit_out = 0;
     byte_out = 0;
   }
 }
```

При декодировании в последнем байте входного потока могут присутствовать «лишние» 4 бита с нулевыми значениями (эта ситуация обсуждалась выше), что с учетом принятых правил может привести к добавлению единичного бита. Однако, поскольку в выходной поток записывается только полный байт, лишняя единица никак не скажется на выходном потоке.

2.1.3. Обсуждение результатов

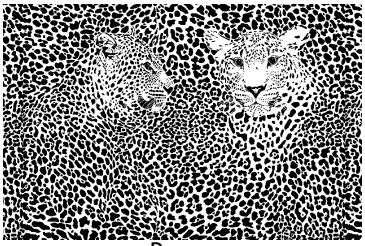
Самая привлекательная черта алгоритма RLE — простота его программной реализации. Оборотной стороной этой простоты является узкая сфера эффективного применения алгоритма. В рамках байт-ориентированной реализации RLE предполагается, что этот недостаток до некоторой степени может быть преодолен за счет использования обратимого преобразования Бэрроуза — Уилера (см. раздел П.2.3). Эксперименты, результаты которых приведены в табл. 1, показывают, что RLE хорош для сжатия контрастных черно-белых изображений (объем сжатого файла уменьшился приблизительно на 60–70% по отношению к исходному), но для изображений с плавными переходами тонов эффект может быть обратным (увеличение объема на 60–80%). Кроме того, выяснилось, что применение преобразования BWT не всегда приводит к желаемому результату.

Ниша эффективного применения бинарного алгоритма RLE, как было показано выше, ограничена сжатием черно-белых изображений с доминирующим черным цветом, тогда как доминирование белого цвета приводит к увеличению объема сжатого файла из-за необходимости использования нулевых блоков для разделения единиц.

Это подтверждают оценки сжатия черно-белых изображений формата bmp, показанных на рис. 3. Так как черный цвет кодируется нулями, а белый — единицами, длина «черных» (нулевых) серий на негативе больше, чем на оригинале. Поэтому негатив, сжатый алгоритмом RLE (рис. 3*б*) уменьшился в объеме приблизительно на треть, в то время как исходное изображение после сжатия увеличилось в объеме почти втрое (рис. 3*а*).

Таблица 1

Результаты применения алгоритма RLE

Формат файла	Степень сжатия, %	
	с BWT	без BWT
 Растр	67,35	57,16
 Растр	–67,26	–83,55
Текст	–120,4	–48,58
Исполняемый формат	19,23	19,17

а) черно-белый рисунок, степень сжатия –245%

б) его негатив, степень сжатия 34%

Рис. 3. Эффективность сжатия черно-белых изображений бинарным алгоритмом RLE

Самые плохие результаты RLE показал для текстового файла. Очевидно, это следствие незначительного количества и малой длины RLE-цепочек в русском тексте (проза Н. М. Карамзина). Уменьшение почти на 20% объема файла исполняемого формата (результат компиляции программы на языке C) частично реабилитирует примитивную схему кодирования RLE.

Приведенные результаты — не приговор той или иной версии алгоритма, так как для вынесения категорических суждений объем испытаний явно недостаточен. Авторы и не ставили перед собой такой цели, чтобы не уйти слишком далеко от первоначальных замыслов. Скорее, это повод для размышлений о том, как в рамках RLE можно повлиять на степень сжатия данных путем параметрической настройки этого алгоритма. Для байт-ориентированного алгоритма в качестве «параметров оптимизации» можно использовать разрешенную длину цепочки и размер блока BWT. Для бинарного алгоритма таким параметром будет разрешенная длина серии нулей. Стоит также заметить, что степень сжатия — это показатель применения *конкретного* алгоритма к *конкретным* данным. Оба фактора влияют на целевой показатель. Например, если бы бинарный алгоритм считал не нули, а единицы, значения степеней сжатия на рис. 3*а, б* поменялись бы местами. Следовательно, «умный» алгоритм сжатия, прежде чем выполнить основную функцию, должен исследовать свойства входного потока, чтобы настроить свои параметры для получения наилучшего результата. Разумеется, об этих настройках он обязан сообщить декодеру, включив их в заголовок сжатого файла.

2.2. Алгоритм Лемпеля — Зива — Велча (LZW)

… словарь есть книга в самом широком значении слова. Все другие книги содержатся в ней: суть в том, чтобы извлечь их из нее.

Вольтер

Алгоритм LZW, названный так по имени разработчиков (Лемпеля, Зива и Велча), опубликован в 1984 году. Его предшественники, LZ77

и LZ78, алгоритмы Лемпеля и Зива 1977 и 1978 годов разработки соответственно, были в основном предметом академического обсуждения. Ситуацию изменила публикация Велча «Средства высокопроизводительного сжатия данных»[9] в журнале IEEE Computer, которая содержала описание алгоритма LZW, усовершенствовавшего алгоритмы Лемпеля и Зива. Выход статьи инициировал работы по созданию утилиты сжатия в операционной системе UNIX [1]. На момент опубликования алгоритм LZW демонстрировал более высокую, по сравнению с существующими аналогами, степень сжатия. Он до сих пор используется в различных модификациях и приложениях (утилиты сжатия, протоколы модемной связи, стандарты графических изображений GIF и TIFF, стандарт PDF) и считается наиболее эффективным алгоритмом словарной группы [11].

Алгоритм LZW понимает под цепочкой произвольную последовательность символов, а модель кодирования LZW представлена в виде таблицы цепочек (словаря), сопоставляющей каждой цепочке ее уникальный код, и правил формирования этой таблицы. Таблица цепочек состоит из корневой части, которая сопоставляет коды цепочкам единичной длины и симметрично инициализируется кодером и декодером до начала кодирования, и динамической части, которая формируется непосредственно в процессе кодирования/декодирования. Таким образом, алгоритм LZW использует динамическую модель кодирования. Если речь идет о байт-ориентированном кодировании, длина корневой части составляет $2^8 = 256$ входов, по числу элементарных цепочек в формате байта. Длина динамической части таблицы цепочек концептуально не ограничена. Но, поскольку размер таблицы определяет длину уникального кода цепочки, в практической реализации его выбирают из ряда степеней двойки, превышающих 8, так как возможности восьмибитового кода исчерпываются при заполнении корневой части таблицы:

Размер таблицы, входов	$2^9 = 512$	$2^{10} = 1024$	$2^{11} = 2048$	$2^{12} = 4096$
Длина кода, бит	9	10	11	12

[9] A Technique for High-Performance Data Compression.

Выбор размера таблицы цепочек — это компромиссное решение. С одной стороны, чем больше таблица, тем больше цепочек может создать кодер и тем большее количество цепочек потенциально может быть заменено кодами фиксированной длины, что уменьшает объем сжатых данных. С другой стороны, длина этих кодов также увеличивается с ростом таблицы, и это работает на увеличение объема. Очевидно, следует увязывать решение о выборе таблицы с размером сжимаемого файла: чем он больше — тем больше должна быть и таблица.

Фиксация размера таблицы порождает еще одну проблему: рано или поздно эта таблица в процессе кодирования/декодирования будет заполнена, и эту исключительную ситуацию должны одинаково обрабатывать и кодер и декодер. Решение этой проблемы состоит в том, что динамическая часть таблицы должна быть полностью или выборочно очищена. При любом способе очистки происходит потеря информации о ранее распознанных цепочках, и это неизбежно должно привести к уменьшению степени сжатия входного потока. Полная очистка динамической части таблицы — самый простой способ решения проблемы, который, однако, приводит к максимальным потерям. Именно он будет обсуждаться далее на уровне программной реализации. Реализация выборочной очистки более сложна — она основана на использовании статистики по распознанным цепочкам (насколько часто или давно они использовались для замены), но способствует минимизации потерь. Эти способы предлагаются читателю для самостоятельной реализации.

Наконец, фиксация размера таблицы порождает проблему работы с кодами, длина которых не кратна байту (если только размер таблицы не равен, например, $2^{16} = 65536$). Эта проблема является общей для многих алгоритмов сжатия, и ее решение будет обсуждаться в разделе П.2.1.

2.2.1. Кодер LZW

Кодер LZW реализует «жадную» стратегию наращивания цепочек: он увеличивает цепочку за счет добавления к ней очередного символа входного потока. Процесс продолжается до тех пор, пока в результате добавления не образуется не известная кодеру (т. е. отсутствующая

в таблице) цепочка. Для понимания его деталей определим понятие префикса цепочки. Под префиксом будем понимать последовательность символов произвольной (в том числе и нулевой) длины, составляющей начальную часть цепочки. С учетом введенного понятия, алгоритм LZW-кодирования выглядит следующим образом:

```
Инициализировать корневую часть Таблицы_Цепочек;
Префикс = Пустая_Цепочка;
ПОКА НЕ Конец(Входной_Поток) ВЫПОЛНЯТЬ
{
  Читать (Входной_Поток, Символ);
  ЕСЛИ (Префикс+Символ) есть в таблице цепочек
    Префикс=Префикс+Символ
  ИНАЧЕ
        {
          Писать(Выходной_Поток, Код_Префикса);
          Цепочка = Префикс+Символ;
          Добавить (Цепочка, Таблица_Цепочек);
          Префикс = Символ
        }
}
Отправить код Префикса в выходной поток.
```

Рассмотрим работу кодера на небольшом примере. Пусть входной поток содержит слово «АНАНАС». Каждый шаг продвижения по входному потоку будем называть итерацией (табл. 2).

Нулевая итерация соответствует шагу инициализации таблицы цепочек. Напомним: таблица цепочек сопоставляет каждой цепочке ее уникальный код. Столбец кода в табл. 2 приведен исключительно по соображениям наглядности: легко заметить, что код цепочки — это индекс строки, содержащей цепочку. В общем случае корневая часть содержит все 256 возможных однобайтовых цепочек, однако для упрощения обсуждения в табл. 2 показаны только те из них, которые присутствуют во

входном потоке. Кроме того, на этой итерации значением «» — пустая строка — инициализируется префикс.

Таблица 2

Пример LZW-кодирования

Итерация	0	1	2	3	4	5	6	7
Таблица цепочек	A 0 H 1 C 2		A 0 H 1 C 2 AH 3	A 0 H 1 C 2 AH 3 HA 4		A 0 H 1 C 2 AH 3 HA 4 AHA 5	A 0 H 1 C 2 AH 3 HA 4 AHA 5 AC 6	
Префикс	«»	«»	«A»	«H»	«A»	«AH»	«A»	«C»
Символ	–	A	H	A	H	A	C	–
Выходной поток	–		0	01		013	0130	01302

Итерации 1–6 выполняются в цикле, условием выхода из которого является достижение конца входного потока. Добавление первого символа входного потока к пустому префиксу образует односимвольную цепочку, которая уже присутствует в корневой части. Поэтому «жадный» кодер продолжает наращивать префикс. В результате первая цепочка добавляется в динамическую часть таблицы на второй итерации, а на выход отправляется код ее префикса «A» — 0. В том же духе кодер действует до достижения конца входного потока. На последней итерации, после выхода из цикла, он записывает в выходной поток код префикса, который не был обработан в теле цикла. В итоге кодером сформирован выходной поток кодов 01302. Он чуть короче входного, так как одну из цепочек в динамической части таблицы удалось заменить ее кодом, но в этом отношении пример не слишком показателен — такая задача и не ставилась. Он иллюстрирует действия кодера в режиме пошаговой интерпретации и, кроме того, потребуется далее для иллюстрации работы декодера.

Приведенная ниже программная реализация практически повторяет описанный процесс. После инициализации корневой части таблицы цепочек функцией Init_table в цикле считываются байты входного потока в переменную byte, значение которой добавляется к префиксу chain, его длина length при этом увеличивается. Далее функцией Chain_found производится поиск обновленной цепочки в таблице и, если он оказывается успешным, ее код code запоминается в переменной outcode для последующего вывода в выходной поток. Если цепочка в таблице не найдена, то код префикса outcode отправляется в выходной поток функцией Write_code, цепочка добавляется в таблицу функцией Add_chain, а последний добавленный байт становится началом очередного префикса, код которого снова ищется в таблице. Однако, как обсуждалось выше, размер таблицы может быть ограничен. Поэтому если таблица достигла своего максимального размера maxcount, динамическая часть таблицы «очищается»: код префикса записывается в выходной поток, а количество элементов в таблице count и префикс получают начальные значения. По завершении цикла в выходной поток записывается код последнего префикса и последнее слово, если в нем остались незаписанные биты кода.

```
Init_table();  // инициализация таблицы
while (fread(&byte, 1, 1, input))
{
  readbytes++; // количество прочитанных байтов
  chain[length] = byte;
  length++;
  if (Chain_found(chain, code, length)) // поиск цепочки
    outcode = code;
  else
  {
    Write_code(outcode, freebits, packcode); // запись кода
    Add_chain(chain, length); // добавление цепочки в таблицу
    chain[0] = chain[length - 1];
    length = 1;
    if (Chain_found(chain, code, length))
      outcode = code;
    if (count == maxcount)
    { // очищение динамической части таблицы
      Write_code(outcode, freebits, packcode);
```

```
        count = 256;
        if (readbytes != filesize) length = 0;
       }
     }
   }
   if (length!= 0) // запись оставшихся данных
     Write_code(outcode, freebits, packcode);
   if (freebits < 16) fwrite(&packcode,sizeof(packcode),1,output);
```

Таблица, используемая в программе, организована как динамический массив структур, каждая из которых содержит цепочку, также размещаемую динамически, и ее длину.

```
 struct chaincode
 {
 unsigned char* chain;
 unsigned char len;
 };
```

Функция инициализации таблицы Init_table не требует пространных объяснений, поскольку содержит простой цикл заполнения массива с операциями выделения памяти.

```
void Init_table()
  {
    table = new chaincode[maxcount];
    memset(table, 0, maxcount * sizeof(chaincode));
    for (int i = 0; i < 256; i++)
    {
      table[i].chain = (unsigned char*)malloc(1);
      table[i].chain[0] = i;
      table[i].len = 1;
    }
    count = 256;
  }
```

Функция Chain_found обеспечивает поиск цепочки в таблице. Для ускорения поиска код однобайтовой цепочки определяется как значение этого байта, для более длинных цепочек производится сравнение искомой цепочки с префиксом в таблице. При успешном поиске функция возвращает значение true.

```
  bool Chain_found(unsigned char *chain, unsigned short &code,
                   unsigned char len)
  {
```

```
  code = 0;
  if (len == 1)
  {
    code = chain[0];
    return true;
  }
  else
  for (int i = 256; i < count; i++)
    if (table[i].len == len &&            // сравнение цепочек
        memcmp(chain, table[i].chain, len) == 0)
    {
      code = i;          // цепочки совпадают
      return true;
    }
  return false;
}
```

Функция Write_code упаковывает код outcode в слово packcode (2 байта) и записывает его в выходной поток. В предложенной реализации длина кода ограничена значением переменной len_code. Как упоминалось выше, если это значение не совпадает с границей байта, приходится прибегать к битовым операциям. Количество свободных битов в выходном слове packcode контролирует переменная freebits.

```
void Write_code(unsigned short outcode, unsigned short &freebits,
                unsigned short &packcode)
{
  if (freebits > len_code) // код помещается целиком
  {
    packcode += outcode << (freebits - len_code);
    freebits -= len_code;
  }
  else
  {
    // в packcode добавляется недостающее число битов
    packcode += outcode >> (len_code - freebits);
    fwrite(&packcode, sizeof(packcode), 1, output);
    // оставшиеся биты перемещаются в packcode
    packcode = outcode << (16 - (len_code - freebits));
    freebits = 16 - (len_code - freebits);
  }
}
```

Функция Add_chain выполняет добавление цепочки chain в таблицу, при этом память для цепочки выделяется динамически в соответствии с ее длиной len.

```
void Add_chain(unsigned char *chain, unsigned short len)
{
  // выделение памяти
  table[count].chain =
             (unsigned char*)realloc(table[count].chain, len);
  memcpy_s(table[count].chain, len, chain, len); // заполнение
  table[count].len = len;
  count++;  // количество цепочек в таблице
}
```

2.2.2. Декодер LZW

Декодер трансформирует полученный от кодера поток кодов в исходный поток символов. Так как процедура сжатия должна быть обратимой, логично предположить, что каждый из кодов, которые декодер извлекает из входного потока, ему известен, то есть на момент декодирования уже присутствует в таблице цепочек. Примем эту гипотезу в качестве рабочей и рассмотрим алгоритм работы декодера:

```
Инициализировать корневую часть Таблицы_Цепочек;
Читать (Входной_Поток, Код);
Цепочка = Декодировать(Код);
Писать (Выходной_Поток, Цепочка);
ПОКА НЕ Конец(Входной_Поток) ВЫПОЛНЯТЬ
{
    Читать(Входной_Поток, Текущий_Код);
    Цепочка = Декодировать(Текущий_Код);
    Писать (Выходной_Поток, Цепочка);
    Цепочка = Декодировать(Код) + ПервыйСимвол(Цепочка);
    Добавить(Цепочка, Таблица_Цепочек);
    Код = Текущий_Код
}
```

Используем для верификации алгоритма декодера входной поток 01302, построенный декодером.

Рассмотренный пример декодирования (табл. 3) подтверждает гипотезу о том, что каждый код, который считывает декодер, ему уже

известен. Однако, прежде чем делать окончательные выводы, рассмотрим еще один пример. На этот раз входной поток кодера представляет собой слово «АНАНАНАС». По причине, которая вскоре станет понятна, рассмотрим лишь начальный этап работы кодера (табл. 4).

Таблица 3

Пример LZW-декодирования

Итерация	0	1	3	5	6
Таблица цепочек	А 0 Н 1 С 2	А 0 Н 1 С 2 АН 3	А 0 Н 1 С 2 АН 3 НА 4	А 0 Н 1 С 2 АН 3 НА 4 АНА 5	А 0 Н 1 С 2 АН 3 НА 4 АНА 5 АС 6
Код	0	1	3	0	2
Цепочка	А	Н	АН	А	С
Выходной поток	А	АН	АНАН	АНАНА	АНАНАС

Таблица 4

LZW-кодирование, порождающее исключительную ситуацию

Итерация	0	1	2	3	4	5	6		
Таблица цепочек	А 0 Н 1 С 2		А 0 Н 1 С 2 АН 3	А 0 Н 1 С 2 АН 3 НА 4		А 0 Н 1 С 2 АН 3 НА 4 АНА 5	А 0 Н 1 С 2 АН 3 НА 4 АНА 5 АНАС 6		
Префикс	«»	«»	«А»	«Н»	«А»	«АН»	«А»	«АН»	«АНА»
Символ	–	А	Н	А	Н	А	Н	А	С
Выходной поток	–		0	01		013	0135		

Теперь посмотрим, как будет обрабатывать результирующий поток 0135… декодер (табл. 5).

Таблица 5

Исключительная ситуация при LZW-декодировании

Итерация	0	1	3	5
Таблица цепочек	A 0 H 1 C 2	A 0 H 1 C 2 AH 3	A 0 H 1 C 2 AH 3 HA 4	A 0 H 1 C 2 AH 3 HA 4 ? ?
Код	0	1	3	5
Цепочка	A	H	AH	?
Выходной поток	A	AH	AHAH	?

Из табл. 5 видно, что на последнем шаге декодер прочитал код 5, которого еще нет в таблице цепочек. Однако внимательный анализ ситуации выявляет следующие факты. Во-первых, неизвестный декодеру код — это очередной код, который должен быть внесен в таблицу цепочек. Во-вторых, этот код (см. табл. 4) образован путем присоединения к текущей цепочке (AH) первого символа этой же цепочки. Таким образом, нам доступна вся информация для обработки исключительной ситуации: в таблицу необходимо добавить цепочку AHA с кодом 5 и успешно довести процесс декодирования до конца. Для исчерпывающего решения этой проблемы достаточно внести соответствующие поправки в алгоритм декодера:

```
Читать(Входной_Поток, Текущий_Код);
ЕСЛИ Текущий_Код есть в Таблице_Цепочек
 {
    Цепочка = Декодировать(Текущий_Код);
    Писать (Выходной_Поток, Цепочка);
    Цепочка = Декодировать(Код) + ПервыйСимвол(Цепочка);
    Добавить(Цепочка, Таблица_Цепочек);
```

```
  }
ИНАЧЕ
  {
     Цепочка = Цепочка + ПервыйСимвол(Цепочка);
     Писать (Выходной_Поток, Цепочка);
     Добавить(Цепочка, Таблица_Цепочек);
  }
```

К счастью, других концептуальных проблем у алгоритма LZW-декодирования не возникает. Рассмотрим программную реализацию декодера. Как это принято в семействе динамических алгоритмов, декодер использует те же ключевые механизмы инициализации и обновления таблицы цепочек, что и кодер. Сначала инициализируется корневая часть таблицы функцией `Init_table`, затем из входного потока функцией `Read_code` считывается первый код в переменную `code`. Как следует из алгоритма кодирования, это код однобайтовой цепочки. Это означает, что он присутствует в таблице, по коду функция `Code_found` находит соответствующую цепочку `str`, которая и отправляется в выходной поток. Записанная цепочка запоминается как префикс в переменной `chain`.

Далее процесс декодирования реализуется в цикле. Код цепочки, только что записанной в выходной поток, сохраняется в переменной `outcode`, это необходимо для обработки обсуждавшейся выше исключительной ситуации. Затем из входного потока считывается очередной код, производится поиск цепочки, соответствующей коду. Если цепочка не найдена, выполняются операции по разрешению исключительной ситуации. Цепочка `str` выводится в выходной поток, количество байтов в выходном потоке `bytes` увеличивается на длину записанной цепочки `len`. Если количество записанных байтов становится равным длине исходного файла `filesize` (это значение записывается в сжатый файл при кодировании), процесс прекращается. Если декодирование продолжается, то анализируется состояние таблицы, поскольку ее размер ограничен. Если таблица заполнена до конца, то, как и при кодировании, ее динамическая часть «очищается». Далее, чтобы обеспечить синхронность кодирования и декодирования, начальный байт цепочки `str[0]` добавля-

ется к префиксу chain, префикс длиной более одного байта помещается в таблицу функцией Add_chain, после чего префиксом становится последняя цепочка.

```
Init_table();
if (Read_code(code, word, bitsinword, bits, freebits))
{
  outcode = code;
  Code_found(str, code, &length);
  fwrite(str, 1, length, output);
  bytes += length;
  memcpy_s(chain, length, str, length);
  while (true)
  {
    outcode = code;
    Read_code(code, word, bitsinword, bits, freebits);
    if (!Code_found(str, code, &len))
    { // исключительная ситуация
      Code_found(str, outcode, &len);
      str[len] = str[0];
      len++;
    }
    fwrite(str, 1, len, output);
    bytes += len;  // количество байтов в выходном потоке
    if (bytes == filesize) break;
    if (count == maxcount)
    { // очищение динамической части таблицы
      count = 256;
      length = 0;
    }
    chain[length] = str[0];
    length++;
    if (length > 1)
      Add_chain(chain, length);
    memcpy_s(chain, len, str, len);
    length = len;
  }
}
```

Функция Read_code извлекает из входного потока код длины len_code и возвращает его параметром code. Поскольку длина кода может оказаться не кратной размеру байта, то при чтении кода, как и при записи, приходится прибегать к битовым операциям. Данные из входного потока считываются в двухбайтовую переменную word, количе-

ство неиспользованных битов в которой хранит переменная bitsinword. Для извлечения кода используется дополнительная переменная bits, которая упаковывается всегда 16-ю битами, количество свободных битов в ней хранит переменная freebits. Если битов в word недостаточно, то из потока считываются следующие два байта. В качестве кода извлекаются старшие len_code битов из переменной bits. Все переменные передаются в функцию параметрами. В случае успешного выполнения функция возвращает значение true.

```
bool Read_code(unsigned short &code, unsigned short &word,
               unsigned short &bitsinword, unsigned short &bits,
               unsigned char &freebits)
{
  int flag_read;
  if (freebits > 0)        // bits не упакована
  {
    if (bitsinword > 0) bits += word >> (16 - freebits);
    if (bitsinword >= freebits)
    {                // битов в word достаточно для упаковки bits
      word = word << freebits;
      bitsinword -= freebits;
      freebits = 0;
    }
    else
    {                // чтение word и доупаковка bits
      freebits = freebits - bitsinword;
      flag_read =
        fread_s(&word,sizeof(word),sizeof(word),1,input);
      if (!flag_read) return false;
      bits += word >> (16 - freebits);
      word <<= freebits;
      bitsinword = (16 - freebits);
      freebits = 0;
    }
  }
  // извлечение кода
  code = bits >> (16 - len_code);
  bits = bits << len_code;
  freebits = len_code;
  return true;
}
```

Функция Code_found выполняет поиск в таблице цепочки по ее коду. Сама операция поиска не требует детальных пояснений, поскольку в данной реализации таблица устроена так, что индекс элемента таблицы соответствует коду. Код передается параметром code, а цепочка и ее длина возвращаются параметрами chain и len соответственно. В случае успешного поиска функция возвращает значение true.

```
bool Code_found(unsigned char *chain, unsigned short code,
                    unsigned char *len)
{
  *len = 0;
  if (code < count)
  {
    memcpy_s(chain, table[code].len, table[code].chain,
             table[code].len);
    *len = table[code].len;
    return true;
  }
  return false;
}
```

2.2.3. Оптимизация хранения и поиска цепочек

К числу технических проблем реализации LZW, помимо обсуждавшегося ранее переполнения таблицы цепочек и работы с кодами, длина которых не кратна байту, относится организация эффективного поиска в таблице цепочек, а также хранение цепочек, длину которых алгоритм LZW принципиально не ограничивает. Рассмотрим возможные подходы к их решению.

Проблема поиска цепочки в таблице касается только кодера, так как декодер для поиска использует код цепочки, который, как уже говорилось, по существу, представляет собой смещение цепочки относительно начала таблицы. Процедура поиска цепочки по ее коду имеет эффективность индексного доступа, $N^0 = 1$, где N — размер таблицы. Другое дело — поиск в таблице по значению цепочки. Для этого необходимо просканировать всю таблицу, что дает оценку эффективности поиска $N^1 = N$. Так как кодер выполняет поиск цепочки каждый раз после добавления текущего символа к префиксу, этот линейный множитель появляется в оценке вы-

числительной эффективности кодера. Следовательно, линейный поиск замедляет процесс кодирования, и необходимо искать пути повышения эффективности этой процедуры. Первое, что приходит в голову, — уменьшить N — размер таблицы цепочек. Но борьба за повышение эффективности на этом направлении может привести к ухудшению основного показателя качества — степени сжатия. Поэтому следует сосредоточиться на решениях, которые при произвольном размере таблицы позволят получить лучшие, по сравнению с линейной, оценки вычислительной эффективности поиска. В качестве возможных кандидатов на решение проблемы можно рассматривать алгоритмы бинарного поиска и хеш-поиска.

Бинарный поиск на упорядоченной последовательности цепочек дает логарифмическую оценку вычислительной эффективности ($\log_2 N$), однако, поскольку таблица цепочек непрерывно пополняется, упорядочение необходимо восстанавливать после добавления каждой новой цепочки. Линейные вычислительные затраты на восстановление упорядочения перекроют возможный выигрыш. Следовательно, бинарный поиск не является перспективным кандидатом.

Хеш-поиск размещает ключи по псевдослучайным адресам. Его вычислительная эффективность может достигать $N^0 = 1$. Так как хеш-поиск не предполагает упорядоченности строковых ключей, необходимость восстановления упорядоченности после добавления нового ключа отпадает. Поэтому данный алгоритм (см. раздел П.2.4) можно рекомендовать для решения проблемы повышения поиска строковых ключей кодером.

Завершая обсуждение проблемы эффективного поиска строковых ключей, необходимо отметить три важных момента. Во-первых, размещение пар «цепочка — код» по случайным адресам, вычисленным с применением хеш-функций, приводит к различию между физическим (случайным) и логическим (по возрастанию кода) размещением пар «цепочка — код» и, как следствие, к необходимости хранения кода цепочки в явном виде. При совпадении физического и логического порядка это излишне: значение кода определяется как индекс цепочки в таблице. Во-вторых, случайное перемешивание строк в таблице цепочек требует контроля ситуации переполнения таблицы с помощью специального счетчика. В-третьих, для декодера, который ищет цепочку по ее коду, организация таблицы цепочек, а также способ обработки ситуации переполнения остаются прежними.

Цепочка	Код
А	0
Н	1
С	2
АН	3
НА	4
АНА	5
АНАС	6

а) явное представление цепочек

Префикс	Символ	Код
-	А	0
-	Н	1
-	С	2
0	Н	3
1	А	4
3	А	5
5	С	6

б) представление цепочек в виде рекуррентной структуры

Рис. 4. Структурное представление длинных цепочек

Проблема хранения длинных цепочек может быть решена двумя способами: силовым и интеллектуальным. В первом случае достаточно программно реализовать соглашение, по которому длина цепочки не может превышать наперед заданного значения. Но, ограничивая длину цепочки, мы лишаем кодер возможности полноценного анализа входного потока и тем самым уменьшаем степень сжатия. Интеллектуальный подход к решению проблемы основан на том, что любая новая цепочка состоит из префикса, который уже размещен в таблице, и последнего символа цепочки, который и является единственным отличием новой цепочки от уже существующей. Поэтому модель цепочки может быть представлена рекуррентной структурой «указатель на префикс — последний символ». Таким образом, в рамках этой структуры может быть неявно представлена цепочка произвольной длины (рис. 4). Недостаток этого подхода состоит в том, что для восстановления цепочки в явном виде потребуются дополнительные вычислительные затраты, чтобы «пробежаться» по цепочке ссылок на префиксы, и эти затраты могут быть довольно ощутимыми — порядка количества входов в таблице цепочек.

2.3. Обсуждение результатов

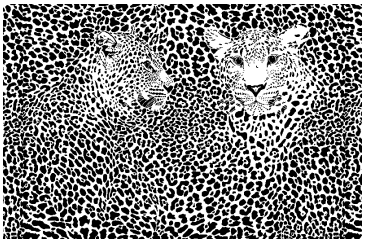
Приведенные в табл. 6 результаты испытаний демонстрируют преимущество алгоритма LZW, трактующего цепочку как произвольную последовательность символов, над алгоритмом RLE, который работает с цепочками повторяющихся символов (см. табл. 1). Преимущество алгоритма LZW вполне объяснимо, так как, во-первых, RLE-цепочка представ-

ляет собой частный случай цепочки LZW, а во-вторых, код цепочки LZW, в отличие от кода RLE, более компактен. Наилучший результат в обоих случаях получен для изобилующего цепочками контрастного растра. Следует также отметить абсолютно лучший среди всех испытанных алгоритмов результат сжатия, полученный для файла исполняемого формата.

Результаты испытаний алгоритма LZW были получены для фиксированного размера таблицы цепочек, который, как уже говорилось, определяет длину кода цепочки и напрямую влияет на величину степени сжатия. Можно предположить, что возможность настройки размера таблицы с учетом длины сжимаемого файла будет способствовать улучшению этого показателя.

Таблица 6

Результаты применения алгоритма LZW

Формат файла	Степень сжатия, %
 Растр	81,92
 Растр	23,16
Текст	37,25
Исполняемый формат	64,95

2.4. Вопросы и задания для самоконтроля

1. За счет чего достигается эффект сжатия данных при использовании словарных алгоритмов?
2. Дайте определение цепочки и кода цепочки в алгоритме RLE.
3. Что представляет собой модель кодирования, используемая в алгоритме RLE?
4. Примените алгоритм RLE для сжатия потока данных «АААААААААААББББББВВВГ». Какой будет степень сжатия, если для кодирования длины цепочки использовать три бита, а каждый символ входного потока представлен в формате байта?
5. Почему область эффективного применения алгоритма RLE ограничена сжатием черно-белых изображений?
6. Как преобразование Бэрроуза — Уилера позволяет преодолеть этот недостаток?
7. Что является объектом и результатом применения преобразования Бэрроуза — Уилера?
8. Какие характеристики прямого преобразования Бэрроуза — Уилера используются для выполнения обратного преобразования?
9. Примените алгоритм RLE для сжатия потока данных «АБРАКАДАБРА» с использованием предварительного преобразования Бэрроуза — Уилера и без него. Сравните значения степени сжатия для этих двух случаев при условии, что для кодирования длины цепочки используются три бита, а каждый символ входного потока представлен в формате байта.
10. Какие современные форматы сжатия используют алгоритм RLE?
11. Дайте определение цепочки и кода цепочки в алгоритме LZW.
12. Что представляет собой модель кодирования, используемая в алгоритме LZW?
13. Что содержат корневая и динамическая части таблицы цепочек? Когда они формируются по отношению к процессам кодирования/декодирования LZW?
14. Почему алгоритм построения цепочки кодером LZW называют «жадным»? Как может повлиять на сжатие ограничение длины цепочки?

15. В чем состоит исключительная ситуация, которая может возникнуть при обработке потока кодов декодером? Как эта ситуация обрабатывается?
16. Примените алгоритм кодирования/декодирования LZW к входному потоку «ccbaaa». Убедитесь в возникновении исключительной ситуации при декодировании и обработайте ее.
17. Какой компонент алгоритма LZW (кодер, декодер) и с какой целью может использовать алгоритм хеширования строк?
18. Как в алгоритме LZW может быть решена проблема компактного хранения длинных строк в таблице цепочек? На каком свойстве строк основан способ решения?
19. Что такое переполнение таблицы цепочек? Как эту ситуацию обрабатывают кодер и декодер LZW? Как это влияет на степень сжатия?
20. Какие программные продукты используют алгоритм LZW?
21. Почему словарные алгоритмы сжатия RLE и LZW позиционируются как динамические?

3. Частотные алгоритмы сжатия

As you are aware, E is the most common letter in the English alphabet, and it predominates to so marked an extent that even in a short sentence one would expect to find it most often.

Sir Arthur Conan Doyle.
The Adventure of the Dancing Men[10]

Подход к минимизации информационной избыточности сообщений, который реализуют частотные алгоритмы сжатия, исходит из того, что между частотой (в пределе — вероятностью) появления символа во входном потоке и длиной его кода в выходном потоке существует обратная зависимость. Чем чаще появляется символ во входном потоке кодера, тем меньше несет информации каждое его появление и тем короче должен быть его код в выходном потоке, и наоборот[11]. Отсюда вытекает очень важное свойство кодов выходного потока для тех частотных алгоритмов, которые представляют их в явном виде: эти двоичные коды имеют разную длину и называются кодами минимальной избыточности (minimum redundancy codes). По сравнению с алгоритмом LZW, который имеет дело с кодами одинаковой длины, возможно, не кратной длине байта, длина двоичных кодов частотных алгоритмов переменна, что создает определенные проблемы при их декодировании.

Под частотной моделью кодирования будем понимать таблицу, которая устанавливает отношение между символами входного потока и их частотами (вероятностями). В отличие от определения модели кодирования как структуры, связывающей символы входного потока и коды выходного потока (см. раздел «Введение»), частотную модель кодирования было бы правильнее называть метамоделью, т. е. правилом, с помощью которого можно построить модель кодирования. Необходимость в применении этого правила возникает у тех частотных кодеров, которые

---

[10] «Как Вам известно, E — самая распространенная буква английского алфавита, и она преобладает настолько, что даже в коротком предложении ее можно ожидать наиболее часто». Артур Конан Дойль. «Пляшущие человечки».

[11] Следовательно, частотные алгоритмы сжатия бессильны перед ситуацией, когда все 256 символов входного потока имеют одинаковую частоту.

генерируют двоичный код для каждого символа входного потока. Однако есть и такие кодеры, которые генерируют код для входного потока в целом, используя для этого саму частотную модель кодирования. Поэтому, чтобы придать общность последующим рассуждениям, в дальнейшем будем использовать наше определение, трактуя его при необходимости как порождающее правило.

3.1. Статические алгоритмы Шеннона — Фано и Хаффмана

Дерево, чего-чего только оно не может…
Рэй Брэдбери. «Марсианские хроники»

Алгоритм Шеннона — Фано был разработан в 1949 году на волне созданной чуть раньше одним из его создателей, Клодом Шенноном, теории информации, трактовавшей такие понятия, как энтропия, количество информации и информационная избыточность. Примечательно, что это произошло задолго до того, как компьютеры стали общепризнанным инструментом решения различных задач [1]. Напомним, что первое программируемое устройство, которое можно было назвать компьютером, появилось в США в 1941 году — это был «МАРК 1», и использовался он исключительно для военных приложений.

Алгоритм Хаффмана [5] был разработан в 1952 году аспирантом Массачусетского технологического института Дэвидом Хаффманом при выполнении им курсовой работы, целью которой было построение кодированного представления входного потока данных с длиной, близкой к его энтропии. По этой причине частотные алгоритмы иногда также называют алгоритмами энтропийного кодирования. Руководителем работы был профессор университета Роберт Фано. Единственное отличие алгоритма Хаффмана от алгоритма Шеннона — Фано заключается в способе использования таблицы частот для построения модели кодирования, устанавливающей соответствие между символами входного потока и их кодами в выходном потоке. Реализация остальных элементов шаблона статического кодирования (рис. 1*а, б*) у них одинакова. Поэтому имеет смысл начать знакомство с этими алгоритмами с обсуждения тех методов,

которые они используют для трансформации метамодели в виде частотной таблицы в настоящую модель кодирования.

Итак, примем, что частотная таблица тем или иным способом построена. Обычно для этого требуется выполнить частотный анализ входного потока путем его сканирования. Однако в тех случаях, когда частотная информация известна априори (например, информация о частоте различных букв в английском тексте), можно обойтись и без сканирования. Так или иначе, результатом является таблица, которая в общем случае содержит 256 входов, по числу возможных значений двоичного кода в формате байта. Каждый вход содержит значение частоты (количества появлений во входном потоке) для кода, значение которого определяет смещение этого входа относительно начала таблицы. Присутствие самого кода в таблице при таком подходе является излишним и даже вредным, так как модель кодирования передается декодеру в составе сжатого файла. Однако для примеров, которые будут обсуждаться ниже, по соображениям наглядности частотная таблица будет строиться только для символов, реально присутствующих во входном потоке. Поэтому каждый вход таблицы будет содержать пару «символ — частота».

Как алгоритм Шеннона — Фано, так и алгоритм Хаффмана строят модель кодирования в формате так называемого *дерева префиксных кодов* — ДПК. Это бинарное дерево, листьям которого сопоставлены символы входного потока кодера, а дуги размечены разрядами двоичного кода таким образом, что для любой родительской вершины одна исходящая дуга помечена нулем, а вторая — единицей. В рамках этого правила разметка может быть любой, важно только, чтобы кодер и декодер следовали единому соглашению. Например, если кодер помечает дугу от родителя к левой дочерней вершине нулем, то же самое должен делать и декодер. Так как модель кодирования — дерево, каждый из его листьев связан с корнем единственным путем, следовательно, конкатенацию двоичных разрядов разметки дуг на пути от корня к листу можно считать уникальным кодом символа. А поскольку символы входного потока сопоставлены листьям, можно утверждать, что *ни один из кодов не*

является префиксом другого. Именно это важное свойство отражено в названии дерева. Благодаря ему обеспечивается обратимость процесса кодирования: декодер сопоставляет биты входного потока битам разметки, начиная от корня и заканчивая листом, и отправляет символ листа в выходной поток. В результате сопоставления символов внутренним вершинам возникла бы неразрешимая дилемма: что считать результатом декодирования — символ внутренней вершины или символ листа?

3.1.1. Построение дерева префиксных кодов Шеннона — Фано

Построение дерева префиксных кодов Шеннона — Фано — это процедура рекурсивного разбиения предварительно упорядоченной[12] частотной таблицы на две части так, чтобы суммы частот в левой и правой частях отличались минимально. Результат каждого разбиения представляется в виде одноуровневого бинарного дерева, дочерние вершины которого имеют вес, равный сумме частот соответствующих частей разбиения, а вес корня равен сумме весов потомков[13]. Процедура разбиения заканчивается, когда длина участка таблицы станет равной 1. Этому единственному элементу сопоставляется символ, соответствующий данному входу таблицы.

Можно заметить, что в результате разбиения упорядоченной таблицы на две части первая часть окажется короче второй. Следовательно, для разбиения этой второй части потребуется больше итераций, и глубина листьев ее символов окажется больше, чем для символов первой части таблицы. А это означает, что, в соответствии с базовым принципом частотных алгоритмов, символам с меньшей частотой соответствует большая длина кода.

---

[12] Для этого алгоритма направление упорядочения не имеет значения. Но при обсуждении примеров будем считать, что речь идет об упорядочении по убыванию частот.

[13] Здесь и далее, когда речь заходит о суммировании весов, на уровне программной реализации необходимо учитывать возможность переполнения разрядной сетки представления целых чисел (16 или 32 бита).

Рассмотрим процесс построения дерева Шеннона — Фано на примере последовательности символов АБРАКАДАБРА. Упорядоченная таблица частот имеет следующий вид:

А	Б	Р	Д	К
5	2	2	1	1

Всего символов 5, сумма всех частот равна 11. Согласно алгоритму построения дерева Шеннона — Фано таблицу необходимо разбить на две части с наиболее близкими суммами частот, что приведет к следующему результату:

А
5

Б	Р	Д	К
2	2	1	1

В левой части один символ с частотой 5, это означает, что левое поддерево содержит только один узел — лист, которому сопоставлен символ «А». В правой части 4 символа с суммарной частотой 6, т. е. она снова подлежит разбиению. Принцип разбиения остается прежним, и здесь возможны два варианта:

1)

Б
2

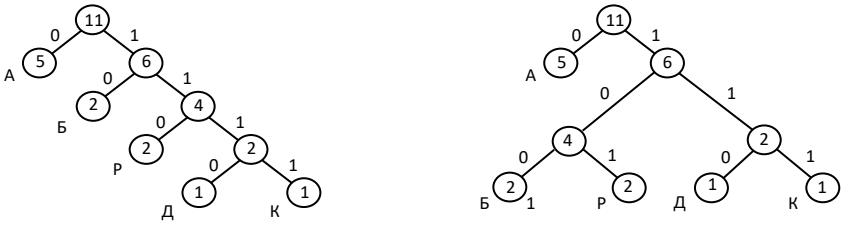
Р	К	Д
2	1	1

2)

Б	Р
2	2

Д	К
1	1

Следует заметить, что оба варианта равнозначны. Чтобы убедиться в этом, достаточно, продолжив процесс разбиения, построить дерево Шеннона — Фано для обоих вариантов (рис. 5).



а) б)

Рис. 5. Два варианта построения дерева префиксных кодов Шеннона — Фано

Коды для первого и второго случая представлены ниже:

Первый вариант			Второй вариант		
Символ	Код	Длина кода	Символ	Код	Длина кода
А	0	1	А	0	1
Б	10	2	Б	100	3
Р	110	3	Р	101	3
Д	1110	4	Д	110	3
К	1111	4	К	111	3

Кодирование слова АБРАКАДАБРА в первом варианте достигается длиной кода 5×1 + 2×2 + 2×3 + 4 + 4 = 23 бита, во втором варианте — 5×1 + 2×3 + + 2×3 + 3 + 3 = 23, т. е. общая длина кода в обоих вариантах совпадает.

Программная реализация этого алгоритма опирается на определение узла бинарного дерева, каковым и является дерево префиксных кодов Шеннона — Фано. Узел — структура Node — является шаблоном, который определяет связи вершины бинарного дерева с левым (left) и правым (right) потомками. Структура PrefixNode с полями: *w* — частота символа, *sym* — символ вершины (это поле имеет смысл только для листьев) определяет содержательное наполнение узла и передается в шаблон Node в качестве параметра.

```
template <class T>
struct Node {
 T data;
 Node *left, *right;
};

struct PrefixNode
{
 long long w;
 unsigned char sym;
};
```

Частотная таблица представлена массивом на 256 элементов по возможному набору значений одного байта, каждый элемент массива является структурой PrefixNode.

Sort() — функция сортировки таблицы частот по убыванию значения поля w. Переменная count — длина вектора частот, после сортировки корректируется и содержит количество байт-кодов с ненулевой частотой повторения.

Построение дерева Шеннона — Фано реализуют две функции. Функция CreateTree инициирует процесс: сортирует вектор частот и определяет начальные границы интервала разбиения. Собственно построение как процедуру рекурсивного разбиения этого начального интервала с созданием потомков текущего корня на каждом уровне рекурсии реализует функция AddNode. Она делает это путем поиска точки разбиения частотного вектора, для которой разница «суммы слева» и «суммы справа» минимальна. Этой точке соответствует значение кумулятивной частоты, накопленное в переменной currsum, и корень дерева на текущем уровне рекурсии, относительно которого создаются корни левого и правого поддеревьев. Представленный код строит дерево по второму варианту. Цепочка рекурсивных вызовов заканчивается в узлах-листьях, когда ширина интервала разбиения становится равной 1. Текущий корень становится листом и дополняется значением атрибута «символ листа» — sym.

```
void AddNode(long long sum, unsigned char left,
             unsigned char right, Node <PrefixNode> **root)
{
  if (left < right)
  {
    long long halfsum = sum / 2, currsum = 0;
    unsigned short i, middle;
    *root = CreateNode({ sum, 0 });
    for (i = left; currsum < halfsum; i++)
      currsum += list[i].w;
    if (halfsum - (currsum - list[i - 1].w) < currsum - halfsum)
    {
      middle = i - 2;
      currsum -= list[i - 1].w;
    }
    else middle = i - 1;
    if (middle == right) middle--;
    AddNode(currsum, left, middle, &((*root)->left));
    AddNode(sum - currsum, middle+1, right, &((*root)->right));
  }
  else
```

```
    *root = CreateNode({ sum, list[left].sym });
}

virtual void CreateTree()
{
  Sort();
  long long sum = 0;
  for (int i = 0; i < count; i++)
    sum += list[i].w;
  if (count > 0)
  {
    unsigned char left = 0, right = count - 1;
    AddNode(sum, left, right, &root);
  }
}
```

3.1.2. Построение дерева префиксных кодов в алгоритме Хаффмана

Если алгоритм построения дерева префиксных кодов Шеннона — Фано строит это дерево «сверху», то построение дерева в алгоритме Хаффмана начинается «снизу», от листьев. В соответствии с этой стратегией для каждого символа таблицы частот создается лист дерева, атрибутом которого является частота этого символа. Затем, пока длина таблицы частот больше 1, алгоритм упорядочивает таблицу по убыванию частот[14], замещает два последних элемента одним, частота которого равна сумме частот этих двух элементов, и создает для него внутренний узел дерева, потомками которого будут узлы, соответствующие двум последним элементам. Созданный узел перемещается в таблице на новое место, исходя из условия упорядоченности частот. Можно заметить, что в результате применения этого алгоритма листья символов, имеющих меньшую частоту, окажутся в дереве на большей глубине, чем те, которые расположены ближе к началу таблицы, упорядоченной по убыванию частот. Это полностью отражает базовый принцип: чем больше частота символа, тем меньше должна быть длина его кода.

---

[14] Выбор направления упорядочения определяет порядок просмотра таблицы. Для таблицы, упорядоченной по возрастанию частот, необходимо начинать ее просмотр сначала.

Для иллюстрации алгоритма построения дерева префиксных кодов Хаффмана рассмотрим все тот же пример со словом АБРАКАДАБРА. Таблица частот для этого входного потока приведена выше. В процессе построения дерева по алгоритму Хаффмана производится поэтапная трансформация исходной таблицы (обратите внимание: упорядочение таблицы восстанавливается на каждой итерации):

Итерация						
0	Индекс	0	1	2	3	4
	Символ	А	Б	Р	К	Д
	Частота	5	2	2	1	1
1	Индекс	0	1	2	3	
	Символ	А	Б	Р	К+Д	
	Частота	5	2	2	2	
2	Индекс	0	1		2	
	Символ	А	Р+К+Д		Б	
	Частота	5	4		2	
3	Индекс	0			1	
	Символ	Р+К+Д+Б			А	
	Частота	6			5	
4	Индекс	0				
	Символ	Р+К+Д+Б+А				
	Частота	11				

Дерево префиксного кодирования, построенное по алгоритму Хаффмана, представлено на рис. 6.

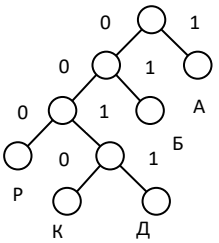


Рис. 6. Дерево префиксного кодирования Хаффмана

В программной реализации этого алгоритма используются те же структуры, что и в алгоритме Шеннона — Фано. Однако в алгоритме Хаффмана для построения дерева применяется список узлов, который представлен массивом структур NodeList:

```
struct NodeList
{
  PrefixNode data;
  Node <PrefixNode> *ptr;// указатель на узел дерева
};
```

Здесь вложенная структура data типа PrefixNode содержит байт-код (sym) и частоту его повторения (w), ptr — адрес узла в дереве Хаффмана.

Алгоритм создания дерева (функция CreateTree) сортирует частотную таблицу, заполняет список узлов, а затем, пока в этом списке не останется только один элемент, выполняет следующую процедуру:

- создает узлы для двух последних элементов списка, если эти элементы соответствуют листьям дерева;
- создает для этих двух узлов родительский узел;
- находит линейным просмотром место вставки родительского узла, сохраняющее упорядоченность списка по значению частоты w;
- уменьшает длину списка.

Корень дерева root адресует указатель, соответствующий первому элементу списка.

```
virtual void CreateTree()
{
  Sort();
  for (int i = 0; i < count; i++)
    nodelist[i].data = list[i];
  int nodecount = count - 1;
  while (nodecount)
  {
    if (!nodelist[nodecount - 1].ptr)
    // узел (nodecount - 1) в дерево не добавлен
      nodelist[nodecount - 1].ptr =
           CreateNode({ nodelist[nodecount - 1].data.w,
           nodelist[nodecount - 1].data.sym });
    if (!nodelist[nodecount].ptr)
    // узел (nodecount) в дерево не добавлен
      nodelist[nodecount].ptr =
           CreateNode({ nodelist[nodecount].data.w,
```

```
            nodelist[nodecount].data.sym });
      // создание родительского узла
      Node <PrefixNode> *tmp =
            CreateNode({ nodelist[nodecount - 1].data.w +
            nodelist[nodecount].data.w, 0 });
      tmp->left = nodelist[nodecount - 1].ptr;
      tmp->right = nodelist[nodecount].ptr;
      int i;
      // вставка родительского узла в список узлов
      // с сохранением упорядоченности
      for (i = nodecount - 2; i >= 0 &&
           nodelist[i].data.w < tmp->data.w; i--)
        nodelist[i + 1] = nodelist[i];
      nodelist[i + 1].data.w = tmp->data.w;
      nodelist[i + 1].data.sym = 0;
      nodelist[i + 1].ptr = tmp;
      nodecount--;
    }
    root = nodelist[0].ptr;
  }
```

Обсуждая приведенные выше примеры, прежде всего необходимо отметить, что, как и ожидалось, построенные деревья отражают обратную зависимость длины кода от частоты. Так, символ «А» с частотой 5 имеет длину кода 1 бит, а символ «Д» с частотой 1 — 4 бита. Степень сжатия для данного примера, вычисленная по формуле (1), составляет

$$R = \frac{88-(5+2*2+2*3+4*2)}{88} * 100 = 74\%.$$

Это означает, что объем данных на выходе кодера на 74% меньше объема входного потока. Правда, этот показатель при реальном использовании алгоритма будет хуже, а для небольших файлов и вовсе может стать отрицательным из-за необходимости передавать декодеру модель кодирования в составе сжатого файла (рис. 1*а, б*). Можно также заметить, что степень сжатия для обеих моделей, и Шеннона — Фано и Хаффмана, совпадает, так как они топологически (с точностью до поворота поддеревьев) идентичны.

В целом, алгоритмы кодирования Шеннона — Фано и Хаффмана достаточно близки по показателю «степень сжатия». Но алгоритм Хаффмана обеспечивает, по крайней мере, равное, а часто — лучшее значе-

ние степени сжатия, поэтому на сегодняшний день большинство коммерческих приложений для генерации кодов минимальной избыточности используют именно его [11]. К тому же Хаффману удалось доказать, что его метод обеспечивает лучший из возможных результатов среди алгоритмов, представляющих результаты кодирования символа целым числом битов [1]. Поэтому дерево префиксного кодирования, построенное алгоритмом Хаффмана, называют деревом *оптимального* префиксного кодирования.

3.1.2.1. Кодер статического алгоритма

За исключением отличий в построении модели кодирования, само кодирование выполняется алгоритмами Шеннона — Фано и Хаффмана одинаково в соответствии со схемой, показанной на рис. 1*а* и включает этапы частотного анализа входного потока, построения модели кодирования, записи этой модели в выходной поток и собственно кодирования. Применение объектно-ориентированного подхода к реализации этих алгоритмов для обеспечения возможности повторного использования кода обсуждается в разделе П.4.

Кодирование реализуется циклом считывания символов входного потока, поиском соответствующего этому символу кода в модели кодирования и записью найденного кода в выходной поток. Программная реализация алгоритма кодирования наталкивается на технические сложности, связанные с нюансами работы с двоичными кодами переменной длины, а также с выбором формата хранения модели кодирования в выходном потоке кодера.

Первая проблема должна быть решена как для представления кодов переменной длины в оперативной памяти, так и для обмена этими кодами с памятью на внешних устройствах. В оперативной памяти она решается при помощи операций поразрядного сдвига и двоичной логики для формирования и выделения значений разрядов бинарного кода. При обмене с внешними устройствами проблема решается при помощи библиотеки классов, реализующих операции побитового чтения и записи (см. раздел П.2.1).

Рассмотрим в качестве примера работы с кодами переменной длины функцию CreateCode, которая при помощи бинарного дерева Шеннона — Фано или Хаффмана строит таблицу префиксных кодов для его листьев. Каждый элемент этой таблицы — структура PrefixCode с полями: code — код символа для выходного потока кодера, и len — длина этого кода. Значение байт-кода во входном потоке определяет индекс элемента в этой таблице. Обратите внимание, что префиксный код code представлен в формате 64-битного целого без знака. Это значит, что длина префиксного кода в данной реализации не может быть больше 64, при том что теоретически она может достигать 255. Ограничение на длину префиксного кода можно снять, организовав его хранение в массиве (см. раздел П.4).

```
struct PrefixCode
{
  unsigned char len;
  unsigned long long code;
};
```

Таблица префиксных кодов заполняется в процессе обхода дерева кодирования методом исчерпывающего поиска в глубину. Соответствующая рекурсивная функция стартует от корня дерева с нулевыми значениями кода code и его длины len. Значение len инкрементируется при каждом рекурсивном вызове. Значение code изменяется только тогда, когда спуск в дереве происходит по дуге, помеченной единицей. По принятому при реализации функции соглашению, которому должен следовать и декодер, единицей помечается дуга, соединяющая родительскую вершину с правым потомком. Благодаря двоичному сдвигу на len разрядов влево, эта единица добавляется в текущий старший разряд двоичного кода. Чем больше значение len, тем левее в коде оказывается соответствующий бит: код как бы записывается справа налево, так его и следует читать при выводе префиксного кода символа в выходной поток. Значение префиксного кода символа и его длина заносятся в таблицу для каждого листа дерева.

```
void CreateCode(Node<PrefixNode> *t, unsigned long long code,
                unsigned short depth)
{
   if (t != nullptr)
   {
      if (!t->left && !t->right)
      {
         codes[t->data.sym].len = len;
         codes[t->data.sym].code = code;
      }
      CreateCode(t->left, code, len + 1);
      code += 1 << len;
      CreateCode(t->right, code, len + 1);
   }
}
```

Запись двоичного разряда префиксного кода в выходной поток выполняет функция writebit(), которая описана в разделе П.2.1. Функция Encode вызывает writebit() в цикле по длине этого кода. Вызову предшествует выделение текущего разряда с применением операций двоичного сдвига: влево, чтобы сделать выделяемый разряд самым старшим в 64-битной сетке, затем вправо на ширину этой сетки, чтобы сделать его младшим. Сдвиг влево «гасит» все биты старше выделяемого, сдвиг вправо делает то же самое с битами, которые младше выделяемого. Обратите внимание на заголовок цикла по выделению разрядов кода: разряды выделяются в том же порядке, в каком были записаны, т. е. справа налево!

```
 void Encode()
 {
   reader = new Reader(fnamein);    // "читатель" входного потока
   if (reader->isopen())            // поток открыт
   {
     list = new PrefixNode[256];    // таблица частот
     codes = new PrefixCode[256];   // таблица кодов
     memset(codes, 0, sizeof(PrefixCode) * 256);
     memset(list, 0, sizeof(PrefixNode) * 256);
     if (Statistic())     // сбор статистики
     {
       CreateTree();      // построение дерева
       if (root)
       {
         CreateCode(root, 0, 0);   // построение таблицы кодов
```

```
        unsigned long long flen = reader->getfilelength();
        unsigned char c;
        // кодирование
        for (long long i = 0; i < flen; i++)
        {
          reader->readbyte(&c);
          for (int k = 0; k < codes[c].len; k++)
          {
            // запись кода в файл, начиная с младшего бита
            unsigned char bit = (codes[c].code << (63 - k)) >> 63;
            writer->writebit(bit);
          }
        }
        writer->flush();
        writer->close();
        return 0;    // успешное завершение
      }
      return 3;    // не создано дерево
    }
    return 2;    // не создан выходной поток
  }
  return 1;    // не открыт входной поток
}
```

Проблема использования кодов переменной длины при работе с внешними устройствами проявляется также в том, что длина сжатого файла в общем случае не кратна байту. Это очень существенно для декодера, распознающего коды переменной длины во входном потоке. Ему необходимо остановиться, прочитав последний код, чтобы не интерпретировать остаток последнего байта во входном потоке как код символа, которого в исходном файле не существовало. Чтобы этого не произошло, кодер может передать декодеру длину (количество байтов) исходного файла flen:

```
unsigned long long flen = reader->getfilelength();
writer->writebyte((long long*)&flen);
```

Возможен и иной подход к решению этой проблемы. А именно, дополнение набора байт-кодов кодом служебного символа eof (конец файла) с частотой, равной 1. Префиксный код для этого символа будет сформирован кодером с помощью обсуждавшегося выше алгоритма построения дерева, передан декодеру в составе модели кодирования и

записан последним в выходной поток кодера. Для декодера распознавание этого кода во входном потоке будет условием выхода из цикла декодирования. Подробности программной реализации этого подхода будут рассмотрены далее при обсуждении адаптивной версии алгоритма Хаффмана.

Вторая проблема — увеличение длины сжатого файла за счет включения в него заголовка, содержащего модель кодирования. Необходимость передачи заголовка является неизбежным злом, вытекающим из природы статических алгоритмов. Если предположить, что частотная статистика хранится в векторе частот list (поле *w* *i*-го элемента списка содержит значение частоты символа с кодом *i*), то сформировать заголовок можно следующим образом:

```
for (int i = 0; i < 256; i++)
 writer->writebyte(&list[i].w,4);
```

Так как частота символа имеет тот же порядок, что и длина файла, а длина файла, например, для файловой системы FAT 32 составляет 4 гигабайта, для хранения значения частоты потребуется 4 байта. Соответственно, для хранения заголовка потребуется 1 килобайт. Предположим, что длина сжимаемого файла не превышает килобайта. Тогда в результате добавления к сжатому файлу заголовка размером 1 килобайт его общий размер по отношению к исходному файлу не уменьшится, а наоборот увеличится. Поэтому возникает проблема выбора компактного представления модели кодирования для передачи декодеру. Можно сохранять только те элементы таблицы, байт-коды которых присутствуют в сжимаемом файле, это позволит сократить объем информации для сильно разреженной таблицы. Можно уменьшить размер таблицы, применив к ее элементам масштабирование к формату байта:

$$\frac{\text{Частота}}{\text{Максимальная частота}} * 255.$$

Для хранения масштабированной таблицы потребуется 256 байт, но следует учитывать, что операция масштабирования выполняется в целых числах, что приводит к потере точности исходной статистики. Напри-

мер, масштабированное значение частоты для некоторых символов входного потока может оказаться нулевым. Поэтому есть смысл обсудить еще один формат передачи модели кодирования — вектор длин кодов. Для рассмотренного выше примера (рис. 5*а*) вектор длин кодов выглядит так:

Символ	А	Б	Р	Д	К
Длина кода, бит	1	2	3	4	4

Вектор длин может быть использован для построения дерева оптимального префиксного кодирования, если, во-первых, элементы этого вектора упорядочены, и, во-вторых, для построения дерева кодирования используется процедура левостороннего или правостороннего[15] исчерпывающего поиска в глубину. Эта процедура, используя заданный приоритет («сначала влево» или «сначала вправо»), ищет способ построения префиксного кода, начиная с кода максимальной длины. Например, для приведенного выше вектора (1, 2, 3, 4, 4) с приоритетом поиска «сначала вправо» будет построен код «1111» для символа «К». Затем, при попытке построения префиксного кода длины 4 для символа «Д», процедура обнаружит, что последний разряд кода уже занят, и в соответствии со стратегией поиска в глубину построит код «1110». Действуя согласно этой схеме, процедура достроит коды «110», «10» и «0». Программный код построения дерева кодирования по вектору длин кодов приведен в разделе П.4 Приложения.

Чтобы оценить выгоды использования этого подхода к построению модели кодирования, необходимо ответить на вопрос, какова предельная длина префиксного кода. Если дерево кодов имеет вырожденную структуру, то максимальная длина кода на единицу меньше длины таблицы. Так, при длине таблицы, равной 5, длина кода равна 4 (рис. 5*а*). Максимальная длина таблицы — 256 входов, следовательно, максимально возможная длина префиксного кода — 255[16]. Это значение вписывается в формат байта, то есть для хранения модели кодирования в формате вектора длин потребуется 256 байт, как и для хранения масштабированного

---

[15] Приоритет поиска определяется выбранным направлением упорядочения.

[16] Это теоретически возможно, когда в модели кодирования присутствуют все символы, и их частоты образуют ряд чисел Фибоначчи.

вектора частот. Однако использование вектора длин не приводит к потере точности исходной статистики.

Применение этого алгоритма для обработки кодером входного потока «АБРА…» с помощью модели кодирования, показанной на рис. 5*а*, порождает выходной поток «0101100…».

3.1.2.2. Декодер статического алгоритма

Что касается декодера, он должен построить модель кодирования, действуя так же, как и кодер, а затем с ее помощью преобразовать поток двоичных кодов переменной длины на входе в поток символов на выходе. Обсудим алгоритм декодирования в предположении, что входной поток предоставляет возможность побитного считывания кодов (функция readbit описана в разделе П. 2.1). Алгоритм декодирования символа реализует рекурсивный процесс спуска из корневой вершины в одну из дочерних, в соответствии со значением считанного бита. В соответствии с соглашением между кодером и декодером, если считан ноль, поиск продолжается в левом поддереве, иначе — в правом. Процесс заканчивается распознаванием закодированного символа в листе.

```
void FindSym(Node<PrefixNode> *t, unsigned char *sym)
{
 if (!t->left && !t->right) // достигнут лист дерева
  *sym = t->data.sym;
 else
 {
  unsigned char c = reader->readbit();
  if (!c)   // прочитан нулевой бит
   FindSym(t->left, sym);
  else FindSym(t->right, sym);
 }
}
```

Эта функция выполняется в цикле считывания битов, пока не будет достигнут конец входного потока (в приведенной реализации предполагается, что кодер передал декодеру длину исходного файла flen):

```
virtual int Decode()
{
  reader = new Reader(fnamein);  // "читатель" входного потока
  if (reader->isopen())
```

```
  {
    if (GetFileExtension())      // создание выходного потока
    {
      list = new PrefixNode[256];    // таблица частот
      codes = new PrefixCode[256];   // таблица кодов
      memset(codes, 0, sizeof(PrefixCode) * 256);
      memset(list, 0, sizeof(PrefixNode) * 256);
      unsigned long long flen, count = 0;
      reader->readbyte((long long*)&flen);
      GetStatistic();        // считывание модели кодирования
      CreateTree();          // построение дерева
      if (root)
      {
        unsigned char c;
        while (count < flen) // цикл считывания битов
        {
          FindSym(root, &c);
          writer->writebyte(&c);
          count++;
        }
        writer->flush();
        writer->close();
        return 0;          // успешное завершение
      }
      else return 3;    // не создано дерево
    }
    else return 2;    // не создан выходной поток
  }
  else return 1;    // не открыт входной поток
}
```

Работу декодера можно проиллюстрировать примером распознавания символа «Б», который, в соответствии с моделью на рис. 5*а*, закодирован цепочкой битов «10». Фазы декодирования показаны на рис. 7. Текущий узел дерева кодирования выделен заливкой.

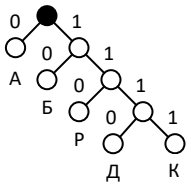
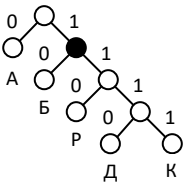
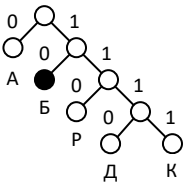
Входной поток	«10»	«0»	«»
Позиция текущего узла			

Рис. 7. Декодирование кода минимальной избыточности

3.2. Адаптивный алгоритм Хаффмана

Перемены — это единственное, что постоянно в жизни. Умение адаптироваться к этим переменам будет определять ваш успех в жизни.

Бенджамин Франклин

Варианты улучшения алгоритма Хаффмана были предложены в семидесятых годах двадцатого века Фоллером, Галлахером и Кнутом (FGK-алгоритм, [7]) и в восьмидесятых годах — Виттером (V-алгоритм, [8]). И в том, и в другом варианте речь идет о динамической (адаптивной) версии алгоритма Хаффмана. Она отличается от исходной версии тем, что модель кодирования в виде дерева префиксных кодов строится непосредственно в процессе кодирования/декодирования и адаптируется к частотной статистике входного потока. С точки зрения результата динамическое построение модели кодирования неизбежно должно привести к уменьшению степени сжатия из-за неполноты текущей статистики, хотя этот недостаток в какой-то степени компенсируется отсутствием необходимости передачи модели кодирования декодеру. С точки зрения процессов кодирования и декодирования это означает, что для обеспечения обратимости сжатия кодер и декодер должны использовать одни и те же операции для инициализации дерева и его обновления после кодирования символа или его декодирования. Идентичный результат инициализации позволяет декодеру распознать первый закодированный символ, а идентичные процедуры обновления модели после кодирования/декодирования очередного символа позволяют декодеру полностью восстановить исходный поток символов.

Учитывая ключевую роль процедур инициализации и обновления модели кодирования, начнем обсуждение динамического алгоритма Хаффмана именно с них.

3.2.1. Инициализация модели кодирования

Инициализация выполняется в тот момент, когда кодер и декодер еще ничего не знают о входном потоке. Поэтому в ее основе должно лежать некое предположение о том, «что будет дальше». Один из двух возможных

вариантов этого предположения состоит в том, что в потоке встретятся все 256 символов байт-кода. Инициализированная в соответствии с этим предположением модель кодирования должна содержать 256 префиксных кодов. Так как априорная информация о частоте символов отсутствует, логично предположить равную вероятность их появления, поэтому все коды будут иметь одинаковую длину — 8 бит и частоту — 0. Иначе говоря, префиксные коды после инициализации будут просто альтернативой байт-кода, и на начальном участке кодирования эффект сжатия будет отсутствовать или проявится в минимальной степени. По мере накопления статистики функция обновления модели кодирования трансформирует дерево префиксных кодов в соответствии с базовым принципом частотного сжатия: «больше частота — меньше длина кода». Таким образом, эффект сжатия при использовании этого подхода к инициализации модели кодирования будет проявляться только на достаточно длинных потоках.

Второй вариант гипотезы о распределении частот символов во входном потоке кодера состоит в отказе предполагать наличие каких-либо символов во входном потоке на этапе инициализации модели кодирования. В этом случае модель инициализируется «пустым» деревом, вернее, деревом, которое содержит только два служебных символа, заведомо отсутствующих во входном потоке кодера (рис. 8). Один из них — символ «конец файла» — eof, а другой — символ «вне контекста» — esc. Служебный символ eof нужен для решения проблемы корректного завершения работы декодера. Целесообразность его использования уже обсуждалась (см. раздел 3.1.2.1). Служебный символ esc необходим как следствие инициализации модели кодирования пустым деревом: появление во входном потоке кодера каждого отсутствующего в модели символа (символа «вне контекста» модели) требует особой обработки при кодировании и декодировании. Частоты служебных символов eof и esc (их значения показаны внутри вершин дерева) равны нулю, так как они фактически отсутствуют во входном потоке. Поскольку этот вариант инициализации модели кодирования при наличии соответствующей реализации функции обновления обеспечивает хорошие показатели степени сжатия даже для коротких потоков, в дальнейшем будет обсуждаться именно он.

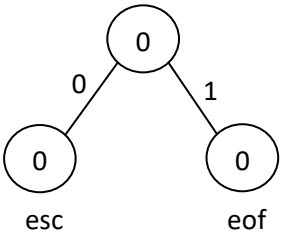


Рис. 8. Динамическая модель кодирования после инициализации

3.2.2. Обновление модели кодирования

Самый простой способ обновления модели кодирования — ее полное построение в соответствии с обновленной статистикой после кодирования/декодирования каждого очередного символа. Однако построение дерева префиксных кодов связано с необходимостью многократной сортировки вектора частот, в худшем случае — после каждого суммирования пары наименьших элементов таблицы частот (см. раздел 3.1.2).

Поэтому данный способ предельно неэффективен. Результатом его применения был бы «самый медленный в мире алгоритм сжатия данных» [1]. Таким образом, динамический алгоритм полностью оправдывает определение прогресса, который, согласно Г. Х. Эллису, состоит в замене одних неприятностей другими[17]. Избавившись от необходимости передавать декодеру модель кодирования, динамический алгоритм породил проблему выбора структуры данных для представления дерева Хаффмана, которая обеспечила бы эффективную реализацию операций добавления символа и обновления дерева символом. Чтобы вычислительные затраты на обновления модели кодирования стали приемлемыми, необходимо сделать обновление инкрементным, а для этого дополнительно потребовать, чтобы дерево префиксных кодов на каждом шаге обновления обладало *свойством упорядоченности* (sibling property — [1]). Дерево обладает свойством упорядоченности, если:

1) его узлы могут быть перечислены в порядке возрастания веса (последовательность вершин с одинаковым значением веса называется блоком);

[17] https://ru.citaty.net/avtory/genri-khevlok-ellis/.

2) в этом перечислении каждый узел находится рядом со своим «братом», т. е. узлом, который имеет с ним общего родителя.

Такое дерево называется *деревом Хаффмана* (пример см. на рис. 9), и благодаря свойству упорядоченности может быть представлено линейной структурой (списком узлов), реализация процедуры обновления на которой более эффективна по сравнению с нелинейным (древовидным) представлением. Базовым элементом этого списка является структура «Узел дерева Хаффмана» — HuffmanNode. Структура объединяет содержательные и навигационные характеристики узла. К содержательным относятся вес узла w и байт-код узла sym. Навигационные включают индекс родителя parent и индекс правого потомка child в линейном представлении дерева.

```
struct HuffmanNode
{
 long long w;
 short sym;     // устанавливается в -1 для внутренних узлов
 short parent;  // устанавливается в -1 для корня
 short child;   // устанавливается в -1 для листьев
};
```

Сам список декларируется как массив узлов:

```
HuffmanNode nodes[515];
```

Кроме того, для исключения непроизводительных вычислительных затрат на сканирование списка nodes с целью поиска листа, соответствующего символу, определен массив, содержащий индексы листьев в списочном представлении дерева для каждого из 256 значений байт-кода и двух служебных символов eof и esc:

```
short symbol_nodes[258];
```

Так как линейный список, представляющий дерево, будет расти вправо, определена переменная, указывающая на первый свободный элемент:

```
unsigned short free_node;   // индекс свободного узла
```

Заметим, что дерево, построенное функцией инициализации, обладает свойством упорядоченности, следовательно, может быть пред-

ставлено с помощью описанных выше структур данных. Вот как выглядит выполняемый и кодером, и декодером код инициализации дерева, показанного на рис. 8:

```
void init_tree() // инициализация дерева
{
  for (int i = 0; i < 256; i++)
    symbol_nodes[i] = -1;
  nodes[0] = { 0, -1, -1, 1 };
  nodes[1] = { 0, eof, 0, -1 };
  nodes[2] = { 0, esc, 0, -1 };
  symbol_nodes[eof] = 1;
  symbol_nodes[esc] = 2;
  free_node = 3;
}
```

Можно заметить, что в результате инициализации корень дерева размещен в начале списка, его правым потомком является лист eof, дочерние узлы eof и esc ссылаются на корень как на родительскую вершину, а веса всех вершин равны нулю. Кроме того, при инициализации элементы массива индексов листьев, соответствующие узлам eof и esc, получают значения, соответствующие позиции этих листьев в списке nodes. До инициализации дерева весь массив индексов листьев заполняется значениями «–1», так что наличие «–1» в ячейке массива означает, что соответствующий символ имеет статус «вне контекста», т. е. отсутствует в модели. Наконец, в результате инициализации формируется указатель на первый свободный элемент дерева.

Рассмотрим процедуру обновления дерева символом на примере входного потока «РДАКРАААААББ» (результат применения преобразования Бэрроуза — Уилера к блоку «АБРАКАДАБРА», см. раздел П.2.3), такую, чтобы после каждого ее применения свойство упорядоченности сохранялось (рис. 9). Учтем, что при поступлении символа на вход кодера возможны две ситуации:

1) символ уже присутствует в модели;

2) символ еще не передавался — not yet transmitted (nyt), эта аббревиатура в некоторых реализациях алгоритма используется вместо esc.

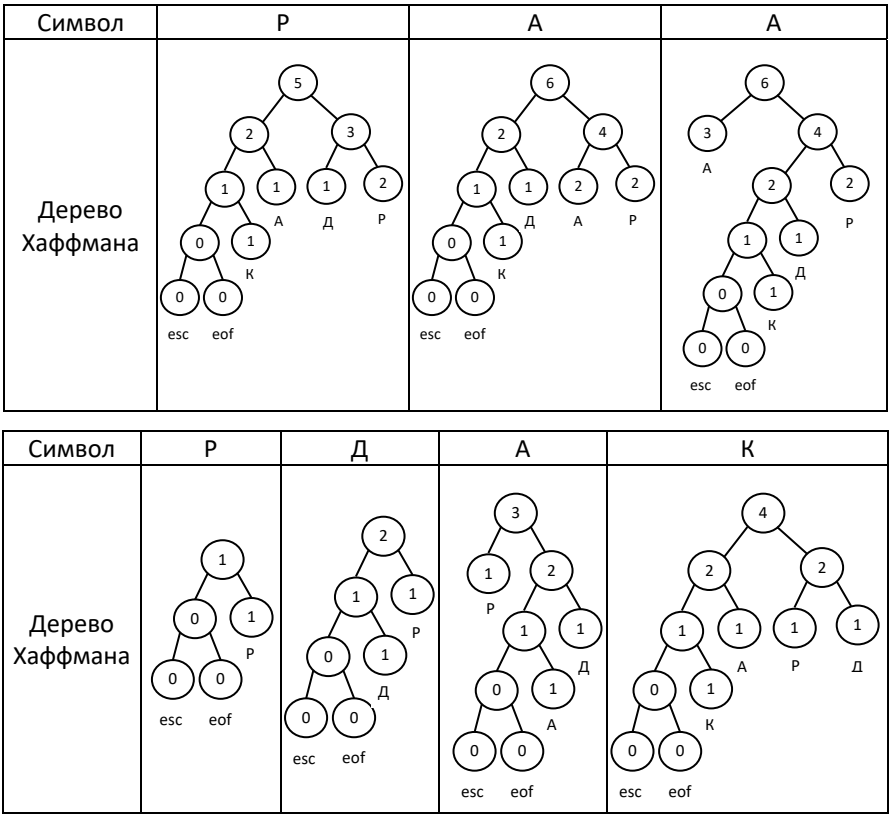


Рис. 9. Процесс формирования дерева Хаффмана адаптивным алгоритмом

Первые четыре символа соответствуют ситуации, когда символ в модель еще не передавался, т. е. в дерево Хаффмана каждый раз следует добавить новый лист. Поскольку частота нового символа равна 1, то и лист добавляется справа от узла с нулевой частотой — родительского узла для символов eof и esc, а в качестве родительского для них создается еще один узел с единичным весом. Так происходит при добавлении первых двух символов; стоит заметить, что второй символ увеличивает вес корня дерева до 2. Добавление в модель третьего символа «А» приводит к необходимости восстановления упорядоченности дерева. Сам процесс восстановления весьма прост: узел, вес которого увеличился, следует поменять местами с самым верхним узлом с весом, меньшим, чем у данного узла. Так, добавление символа «А» привело к тому, что

дочерние узлы корня дерева поменялись местами, а добавление четвертого символа «К» изменило структуру дерева. Следующие далее символы «РАА…» обрабатываются в контексте модели.

Можно заметить, что с увеличением частоты символа «А» длина его кода уменьшается, в то время как длина кодов esc и eof, имеющих нулевую частоту, напротив, увеличивается. Из этого наблюдения следует сделать два вывода. Во-первых, на данном примере подтверждается, что динамическая версия алгоритма Хаффмана реализует базовый принцип частотных алгоритмов: чем больше частота символа, тем меньше длина его кода. Во-вторых, в отличие от статического алгоритма Хаффмана, код конкретного символа подвержен изменениям по мере чтения входного потока. Например, код символа «А» после первого его появления — 101, а после третьего — 0.

Остановимся на программной реализации процедуры восстановления упорядоченности дерева Хаффмана. Напомним, что в линейном представлении дерево Хаффмана — это массив узлов, упорядоченный по убыванию весов, причем за каждым родительским узлом сначала следует его правый потомок, а непосредственно за ним — левый. Добавление нового символа в дерево приводит к необходимости либо увеличить вес существующего листа, либо создать новый лист. При этом должны корректироваться веса всех родительских узлов, что может нарушить свойство упорядоченности дерева. Для восстановления этого свойства в упорядоченном списке узлов нужно найти самый верхний (левый) узел с весом, меньшим веса, увеличенного при добавлении символа, и эти узлы поменять местами. При этом не следует забывать о необходимости корректировки связей между родительскими и дочерними узлами. Специфика списочного представления выражается в необходимости изменения указателей на родителя у дочерних вершин, если меняются местами внутренние вершины, перестановки указателей на родителя между перемещенными узлами и, наконец, обновления указателей на переставляемые узлы в массиве индексов, если эти узлы являются листьями. Затем следует перейти к родителю перемещенного узла и повторять процесс до тех пор, пока не будет достигнут корень дерева. Вот как вы-

глядит функция обновления дерева кодирования символом, присутствующим в модели, в привязке к списочному представлению этого дерева.

```
void UpdateTree(short curr_node)
{
  // пока текущая вершина не корень
  while (curr_node >= 0)
  {
    // увеличиваем вес текущего узла
    nodes[curr_node].w++;
    int curr = curr_node - 1;
    // ищем место для перемещения узла
    while (curr>0 && nodes[curr_node].w > nodes[curr].w) curr--;
    curr++;
    // если текущий узел нужно перемещать
    // (узел не нужно обменивать со своим родителем,
    // т.к. его вес будет увеличен на следующей итерации)
    if (curr != curr_node && curr != nodes[curr_node].parent)
    {
      //если текущий узел меняем с листом
      if (nodes[curr].child == -1)
        // обновляем указатель на этот лист
        symbol_nodes[nodes[curr].sym] = curr_node;
      else
      {
        // если меняем с внутренним узлом, устанавливаем
        // текущий узел как родительский для потомков этого узла
        nodes[nodes[curr].child].parent = curr_node;
        nodes[nodes[curr].child + 1].parent = curr_node;
      }
      // если текущий узел - лист
      if (nodes[curr_node].child == -1
        // обновляем указатель на него
        symbol_nodes[nodes[curr_node].sym] = curr;
      else
      {
        // иначе перенастраиваем ссылки его потомков на родителя
        nodes[nodes[curr_node].child].parent = curr;
        nodes[nodes[curr_node].child + 1].parent = curr;
      }
      // меняем узлы местами
      HuffmanNode buf = nodes[curr];
      nodes[curr] = nodes[curr_node];
      nodes[curr_node] = buf;
      // переопределяем родителей переставляемых узлов
      short bufparent = nodes[curr].parent;
```

```
      nodes[curr].parent = nodes[curr_node].parent;
      nodes[curr_node].parent = bufparent;
    }
    // переходим к родителю перемещенного узла
    if (curr != nodes[curr_node].parent)
      curr_node = nodes[curr].parent;
    else
      curr_node = nodes[curr_node].parent;
  }
}
```

Побочный эффект обновления заключается в том, что по мере накопления частотной статистики значение суммы частот в корне дерева Хаффмана может выйти за пределы диапазона представления беззнаковых целых чисел. Чтобы предупредить возникновение этой проблемы, автор [1] предлагает при приближении веса корневой вершины к предельному значению масштабировать веса вершин дерева путем их деления на два. Однако это решение порождает новую проблему: так как деление выполняется в целых числах, ошибка округления может привести к нарушению свойства упорядоченности и необходимости ее последующего восстановления при помощи уже рассмотренных процедур. Решение этой задачи здесь не рассматривается, однако желающие могут обратиться к первоисточнику.

В случае обновления дерева символом, который еще не встречался во входном потоке кодера, необходимо обеспечить добавление в дерево нового листа. Функция, реализующая выполнение этих действий, подготавливает место для вставки двух новых узлов и создает узел с весом 0, соответствующий новому символу, и его родительский узел тоже с нулевым весом, правым потомком которого становится вновь добавленный лист, а левым — дерево инициализации. Увеличение веса вновь созданных узлов обеспечивается вызовом обсуждавшейся ранее функции обновления дерева — UpdateTree, там же устраняются и возможные нарушения упорядоченности. Функция добавления символа в модель выглядит так.

```
void AddList(unsigned char c, bool encode)
{
```

```
  // сдвигаем на две позиции вправо три последних узла дерева
  nodes[free_node+1]=nodes[free_node-1];
  nodes[free_node]=nodes[free_node-2];
  nodes[free_node-1]=nodes[free_node-3];
  // создаем новый родительский узел для перемещенного поддерева
  nodes[free_node-3]={0,-1,nodes[free_node-1].parent,free_node-2};
  // создаем новый лист для добавляемого символа c
  nodes[free_node-2]={0,c,free_node-3,-1};
  // настраиваем связи перемещенного поддерева
  nodes[free_node-1].parent=free_node-3;
  nodes[free_node-1].child=free_node;
  nodes[free_node].parent=free_node-1;
  nodes[free_node+1].parent=free_node-1;
  // при кодировании обновляем указатели на перемещенные листья
  if (encode)
  {
    symbol_nodes[nodes[free_node].sym] = free_node;
    symbol_nodes[nodes[free_node + 1].sym] = free_node + 1;
    symbol_nodes[c] = free_node - 2;
  }
  // обновляем индекс свободного узла
  free_node+=2;
}
```

3.2.3. Кодер динамического алгоритма Хаффмана

С учетом описанных ранее функций инициализации модели, обновления модели и добавления символа алгоритм кодера можно представить следующим ниже кодом.

```
  void Encode()    // сжатие файла
  {
    short current_node;
    // массив для записи кода Хаффмана для символа
    unsigned char code[256];
    // реализация класса reader описана в  разделе П.2.1
    reader = new Reader(fnamein);
    if (reader->isopen())
    {
     // здесь должен быть код формирования заголовка сжатого файла
      unsigned char c;
      bool end_of_file = false;
      do
      {
        if (!reader->readbyte(&c))
```

```
{
  // в выходной поток будет отправлен код eof
  current_node = symbol_nodes[eof];
  end_of_file = true;
}
else
// если прочитан символ вне контекста
if (symbol_nodes[c] == -1)
  // в выходной поток будет отправлен код esc
  current_node = symbol_nodes[esc];
else
  // в выходной поток будет отправлен код символа
  current_node = symbol_nodes[c];
// строится код Хаффмана, начиная с вершины current_node
short depth = 0;
do
{
  // для правых вершин (с нечетным номером) добавляется 1
  if (current_node & 1)
    code[depth++] = 1;
  // для левых вершин добавляется 0
  else
    code[depth++] = 0;
  // переход к родителю в дереве Хаффмана
  current_node = nodes[current_node].parent;
  // пока текущая вершина – не корень
} while (current_node);
depth--;
// код Хаффмана записывается в выходной поток
while (depth >= 0)
{
  writer->writebit(code[depth]);
  depth--;
}
// если записан код esc
if (symbol_nodes[c] == -1)
{
  // пишется байт-код символа из входного потока
  for (int i = 0; i < 8; i++)
    writer->writebit((c >> (7 - i)) & 1 );
  // добавление символа в дерево
  AddList(c, true);
}
// обновление модели символом
```

```
        UpdateTree(symbol_nodes[c]);
      } while (! end_of_file);
      writer->flush();
      writer->close();
    }
  }
```

Логика кодера заключается в последовательном сканировании входного потока до обнаружения признака конца файла. Если прочитанный символ ранее во входном потоке не встречался (формально это устанавливается путем анализа значения указателя на узел символа в дереве: symbol_nodes[c] == –1), кодер строит префиксный код для зарезервированного символа esc. Если прочитанный символ входного потока присутствует в дереве, кодер строит префиксный код для этого символа.

В отличие от статической версии кодера Хаффмана, коды символов входного потока изменяются по мере накопления статистики. Чтобы построить текущий код символа, кодеру необходимо организовать цикл просмотра узлов дерева от листа, соответствующего символу, до корня. Значения двоичных разрядов кода на каждом шаге перехода от узла-потомка к узлу-родителю накапливаются в массиве code. Позиция разряда в коде и соответствующий ей индекс в массиве code определяется значением переменной depth, которая инкрементируется на каждом переходе «потомок — родитель». Что касается самих двоичных значений, кодер определяет их, руководствуясь простым правилом: разряд кода, формируемый при подъеме от правого потомка к родителю, имеет значение 1; разряд кода, формируемый при подъеме от левого потомка, имеет значение 0. Формально это устанавливается путем проверки четности индекса в списке узлов дерева: нечетные индексы имеют правые потомки.

После того, как код построен, он выводится в выходной поток кодера. При этом кодер учитывает, что разряды префиксного кода заносились в массив code в обратном порядке — от листа к корню, тогда как в выходной поток они должны выводиться в порядке «от корня к листу».

Поэтому биты префиксного кода выбираются циклом вывода, начиная со старшего разряда массива code.

Затем кодер вновь проверяет, код какого символа был им построен, и если это код служебного символа esc, выводит в выходной поток восьмибитовый код нового символа. Следует понимать, что, в соответствии с принятым способом хранения, биты этого кода хранятся в обратном порядке. Вслед за этим кодер добавляет новый символ в модель и вызывает функцию UpdateTree(), чтобы устранить возможные нарушения свойства упорядоченности.

3.2.4. Декодер динамического алгоритма Хаффмана

Декодер, используя те же правила инициализации и обновления дерева кодирования, что и кодер, применяет для распознавания символов по их префиксным кодам ту же логику спуска по дереву кодирования, что и декодер статического алгоритма. Однако у динамической версии декодера есть несколько особенностей. Главная из них заключается в необходимости обновлять дерево кодирования после распознавания каждого символа. Обновлению должен предшествовать анализ частных случаев. В ходе анализа декодер выясняет, присутствует распознаваемый символ в дереве кодирования или нет, и в зависимости от результата либо выводит в выходной поток символ листа, либо интерпретирует следующие восемь битов входного потока как оригинальный символ. Еще одна особенность декодирования потока префиксных кодов заключается в присутствии в нем завершающего кода для служебного символа eof. Распознав его, декодер заканчивает свою работу. Поскольку eof присутствует в дереве префиксного кодирования по определению, у декодера нет необходимости использовать длину исходного файла для проверки условия завершения своей работы. Наконец, существует еще одна, на этот раз техническая особенность. Она связана со списочным представлением дерева кодов, которое, благодаря возможности его «точечного» обновления, делает разумными вычислительные затраты на реа-

лизацию процесса динамического кодирования и декодирования. Все эти особенности учитывает приведенный ниже код.

```
void Decode()  // декодирование файла
{
 unsigned char bit, code;
 reader = new Reader(fnamein);
 if (reader->isopen())
 {
  // здесь должен быть код для считывания заголовка сжатого файла
  short node = 0;    // 0 – индекс корня в дереве
  // «бесконечный» цикл, выход из которого происходит по eof
  while (true)
  {
   bit = reader->readbit();   // читаем бит входного потока
   // спускаемся по дереву влево (прочитан 0) или вправо (1)
   if (bit == 1)
    node = nodes[node].child;
   else node = nodes[node].child + 1;
   if (nodes[node].child == -1)  // если текущий узел – лист
   {
    if (nodes[node].sym == eof)  // если лист "eof"
     break;                      // завершаем цикл
    else
    if (nodes[node].sym == esc)  // если лист "esc"
    {
     // следующие 8 битов читаем как байт-код символа
     code = 0;
     for (int i = 0; i < 8; i++)
     {
      bit = reader->readbit();
      code = (code << 1) + bit;
     }
     writer->writebyte(&code);
     AddList(code, false);  // добавление нового узла в дерево
     UpdateTree(free_node - 4);
    }
    else
    {
     // вывод символа, соответствующего листу
     writer->writebyte((unsigned char *)&nodes[node].sym);
     UpdateTree(node);   // восстановление упорядоченности дерева
    }
```

```
      node = 0;
     }
    }
    writer->flush();
    writer->close();
   }
  }
```

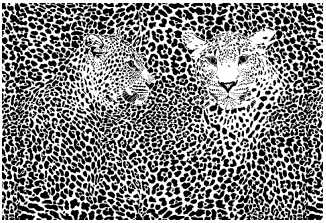
3.2.5. Обсуждение результатов

Результаты проведенных испытаний (табл. 7) подтверждают правоту Хаффмана, доказавшего, что его алгоритм обеспечивает лучший результат среди алгоритмов, представляющих результаты кодирования символа целым числом битов. Действительно, статический алгоритм Хаффмана имеет символическое преимущество над алгоритмом Шеннона — Фано. Что касается адаптивного алгоритма Хаффмана, он также во всех тестах показал небольшое преимущество над своими статическими конкурентами. Интересно, что если подсчитать степень сжатия, достигнутую статическим алгоритмом сжатия (например, алгоритмом Хаффмана), при условии, что в сжатый файл не включена таблица частот, то преимущество окажется на стороне статической версии. Например, на тестовом файле этот показатель для статической версии станет равным 40,28%, тогда как для адаптивной он равен 40,07%. Понятно, что с ростом размера сжимаемого файла доля таблицы частот в объеме сжатого файла будет неуклонно уменьшаться, в силу чего преимущество адаптивного алгоритма в достижимой степени сжатия будет стремиться к 0.

Если к тому же учесть сложность программной реализации и то, что вычислительная эффективность адаптивного алгоритма, обновляющего дерево кодирования на каждом шаге, ниже, чем у его статического аналога, перспективы практического применения адаптивной версии становятся и вовсе не радужными. Интересны результаты испытания статического алгоритма Хаффмана, передающего декодеру модель кодирования в формате вектора длин. Здесь выигрыш в степени сжатия, по сравнению с версией, передающей декодеру таблицу частот, достигается за счет уменьшения длины заголовка сжатого файла. Тело сжатого файла в обоих случаях имеет одинаковую длину.

Таблица 7

Результаты применения алгоритмов Шеннона — Фано и Хаффмана

Формат/размер файла (байт)	Степень сжатия, %			
	Шеннон-Фано	Хаффман	Хаффман (вектор длин)	Хаффман адапт.
 Растр/96782	76,11	76,49	77,23	77,33
 Растр/ 54062	36,39	36,47	37,62	37,85
Текст/ 65536	38,42	38,72	40,01	40,07
Исполняемый формат	44,84	46,03	46,85	47,04

3.3. Алгоритм арифметического кодирования

Мир построен на силе чисел.
Пифагор

Хаффману удалось доказать, что его алгоритм обеспечивает максимально возможную степень сжатия для случая, когда префиксный код имеет длину, измеряемую целым числом битов. Однако предложенная

Шенноном формула для определения количества информации, передаваемой символом *C*:

$$I(C) = -p(C)\log_2 p\,(C),$$

где *p(C)* — вероятность появления символа *C* в сообщении (входном потоке), предполагает, что количество информации в общем случае выражено дробным числом битов. Согласно этой формуле символу, вероятность появления которого во входном потоке равна, например, 0,9, должен быть сопоставлен код длиной 0,14 бит (рис. 10), но алгоритм Хаффмана в принципе не может сделать код короче 1 бита.

Поскольку выделение дробного числа битов под отдельно взятый символ при дискретном кодировании невозможно, следует сосредоточиться на поисках такого подхода, при котором объектом кодирования является не отдельный символ, а все входное сообщение. Именно такой подход реализует алгоритм арифметического кодирования, базовая концепция которого была впервые сформулирована в шестидесятых годах XX века [9].

Вместо того чтобы заменять каждый символ входного потока его кодом в выходном потоке, арифметическое кодирование заменяет весь входной поток единственным числом с плавающей точкой из диапазона 0..1. Для построения этого числа алгоритм арифметического кодирования, как и алгоритм Хаффмана, использует таблицу частот символов входного потока. Однако способ представления и использования этой информации совершенно иной. Частотная статистика представляется в виде интервала вещественных чисел от 0 до 1, разбитого на подынтервалы, ширина которых пропорциональна вероятности появления соответствующего символа во входном потоке. Если *N* — количество символов в алфавите входного сообщения, а p_i — вероятность появления *i*-го символа во входном потоке, то этому символу будет сопоставлен интервал, который включает свою нижнюю границу и не включает верхнюю [11]:

$$\left[\sum_{k=0}^{i-1} p_k, \sum_{k=0}^{i} p_k\right), i = 1, N;\ p_0 = 0. \qquad (2)$$

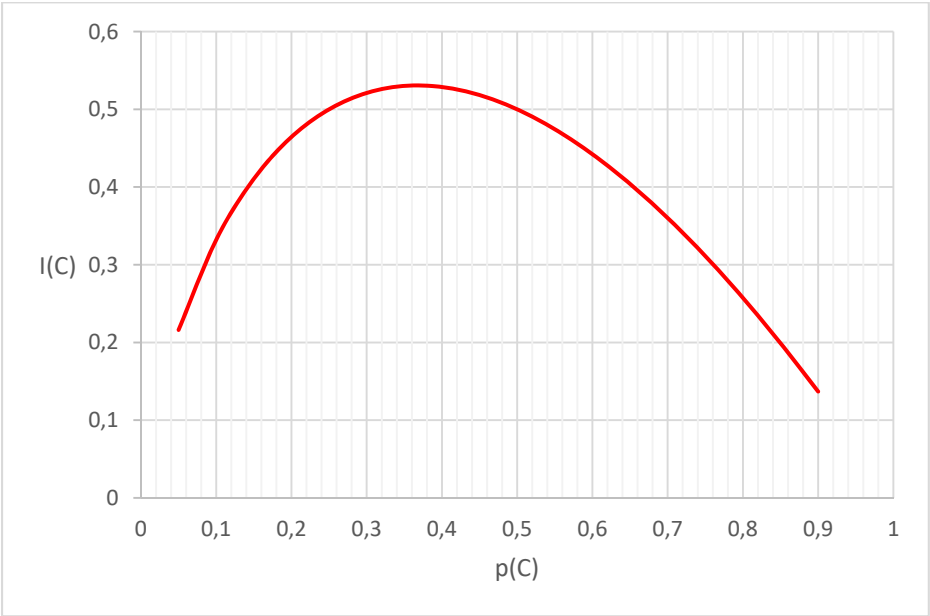


Рис. 10. Количество информации I(C), которое несет символ «C», передаваемый с вероятностью p(C)

Понятно, что появление одного из символов алфавита во входном потоке — достоверное событие, поэтому суммарная ширина N интервалов равна 1. Порядок следования участков не является существенным, важно лишь, чтобы и кодер и декодер в определении этого порядка следовали единому соглашению. По сути дела, такое представление — это предельный переход от частоты символа к вероятности его появления во входном потоке. В зависимости от того, используют ли кодер и декодер заранее сформированную частотную статистику или же формируют ее в процессе анализа входного потока, алгоритмы арифметического кодирования подразделяются на статические и динамические.

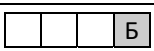
3.3.1. Статическое арифметическое кодирование

Статический алгоритм предполагает, что частотная статистика уже собрана и представлена в виде шкалы интервалов вероятности до начала кодирования/декодирования. Пока не обнаружен конец входно-

го потока, кодер читает символ, находит интервал, соответствующий этому символу, устанавливает границы шкалы равными границам найденного интервала и разбивает суженную шкалу на интервалы, ширина которых пропорциональна частотам соответствующих символов. В качестве результата кодирования всего входного потока кодер может использовать любое вещественное число, которое лежит внутри полученного интервала, в том числе — включенную нижнюю границу интервала. Рассмотрим работу этого алгоритма на простом примере [11]. Пусть на вход кодера поступает поток символов «АААБ». Процесс кодирования иллюстрирует табл. 8. Результат кодирования, как уже было сказано, можно представить любым числом, которое лежит внутри интервала [81/256, 27/64).

Таблица 8

Масштабирование шкалы интервалов при кодировании

Входной поток	Нижняя граница	Верхняя граница	Шкала интервалов
АААБ	0	1	
ААБ	0	3/4	
АБ	0	9/16	
Б	0	27/64	
eof	81/256	27/64	

Для того чтобы оценить эффективность сжатия, которую обеспечивает алгоритм арифметического кодирования, выберем принадлежащее интервалу число 3/8. Это двоичная дробь, знаменатель которой представлен наименьшей для данного интервала степенью двойки. Учитывая, что целая часть результата кодирования всегда равна нулю, этот результат может быть представлен тремя битами выходного потока. Таким образом, получается, что в среднем на один символ входного потока приходится ¾ бита в выходном потоке. При использовании алгоритма Хаффмана мы получили бы 4 бита, так как в построенном дереве каждый символ из алфавита входного потока должен быть представлен как минимум одним битом кода. Данный пример может служить иллюстрацией того факта, что арифметическое кодирование обеспечивает более вы-

сокую, по сравнению с алгоритмом Хаффмана, степень сжатия. Действия кодера формализует следующий алгоритм [1]:

```
low = 0.0
high = 1.0;
пока не конец входного потока
{
    читать символ с
    /* пересчитать границы интервала*/
    range = high - low;
    high = low + range * high_range(c);
    low = low + range * low_range(c);
}
/*вывести результат кодирования*/
Вывести значение low;
```

Здесь переменные low и high обозначают текущие нижнюю и верхнюю границы шкалы распределения вероятностей соответственно, range — ширину интервала, high_range(c), low_range(c) — функции, возвращающие значение верхней и нижней границы для символа *c* на шкале единичной ширины. В качестве результата кодирования в данном случае используется значение нижней границы текущего интервала, хотя, как уже говорилось, результат может быть представлен любым значением внутри него.

Статический декодер, получив в качестве результата кодирования вещественное число, преобразует его в исходный поток символов, используя для этого переданную ему шкалу интервалов. Пока декодирование не закончено, он находит интервал, которому принадлежит кодовое число, выводит в выходной поток соответствующий символ, рассчитывает границы суженного интервала и масштабирует его. Проблема заключается в интерпретации условия «пока декодирование не закончено». Декодер может ее решить, если ему известна длина входного потока кодера или очередным декодированным символом окажется признак конца потока (eof). Описанные действия соответствуют следующему алгоритму:

```
прочитать вещественное число number из входного потока
пока декодирование не закончено
```

```
{
    найти символ symbol, соответствующий number
    вывести symbol в выходной поток
    /*обновить ширину интервала*/
    range = high_range( symbol ) - low_range( symbol );
    /*привести число к новому диапазону*/
    number = number - low_range( symbol )/range;
}
```

В рассмотренном выше примере на вход декодера поступает число 81/256[18] (0.316406 в десятичном выражении), которое принадлежит интервалу, отведенному символу «А». Затем кодовое число трансформируется в 27/64 (0.421875) и снова попадает в интервал «А». Следующее значение кодового числа 9/16 (0.5625) вновь оказывается внутри интервала «А». Наконец, значение кодового числа в результате последней трансформации оказывается равным 3/4 (0.75) и попадает в интервал «Б». Декодирование, в предположении, что декодер знает, где остановиться, заканчивается. Исходный поток «АААБ» успешно декодирован.

Чтобы оценить практические последствия применения такого подхода, рассмотрим более сложный пример [1]. Частотную статистику входного потока «арифметика» алгоритм арифметического кодирования представляет в виде табл. 9. Здесь «[» в обозначении диапазона означает включение нижней границы, а «)» — исключение верхней границы.

Таблица 9

Распределение вероятностей для входного потока «арифметика»

Символ	Вероятность	Диапазон
а	1/5	[0, 1/5)
е	1/10	[1/5, 3/10)
и	1/5	[3/10, 1/2)
к	1/10	[1/2, 3/5)
м	1/10	[3/5, 7/10)
р	1/10	[7/10, 4/5)
т	1/10	[4/5, 9/10)
ф	1/10	[9/10, 1)

[18] Использование 3/8 в качестве кодирующего числа привело бы к точно таким же результатам.

Результаты пошагового применения этого алгоритма к тестовому входному потоку представлены на рис. 11. Итак, результат кодирования входного потока «арифметика» с помощью описанного алгоритма представлен значением нижней границы (0.14985136).

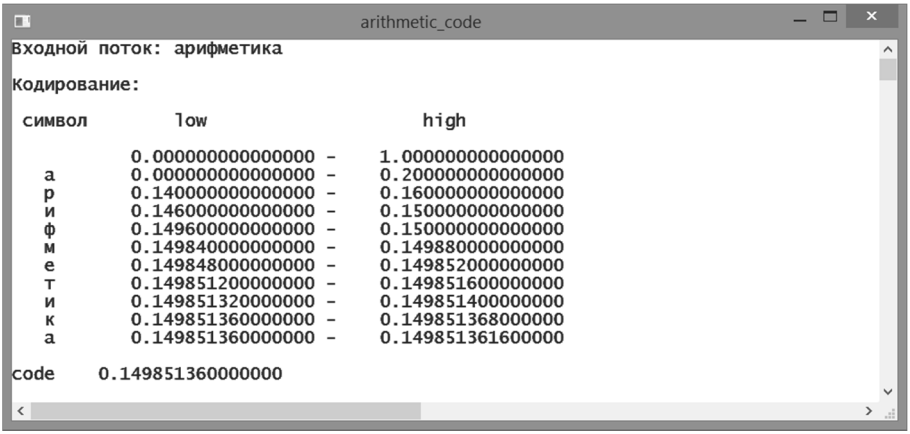


Рис. 11. Процесс кодирования входного потока «арифметика»

Декодер, опираясь на то же самое представление частотной статистики, использует полученное кодером значение для восстановления входного потока. Результаты декодирования потока «арифметика» приведены на рис. 12.

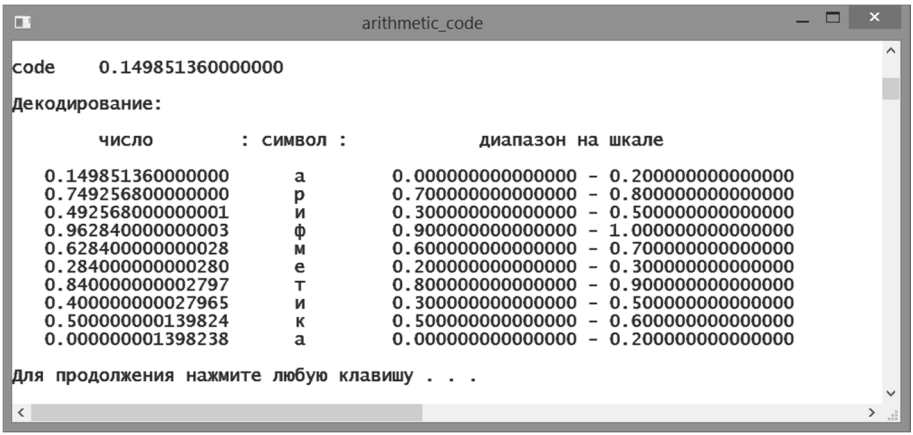


Рис. 12. Процесс декодирования входного потока «арифметика»

Таким образом, кодер сужает первоначальный интервал пропорционально предопределенной вероятности очередного символа входного потока, тогда как декодер расширяет этот интервал пропорционально предопределенной вероятности декодированного символа.

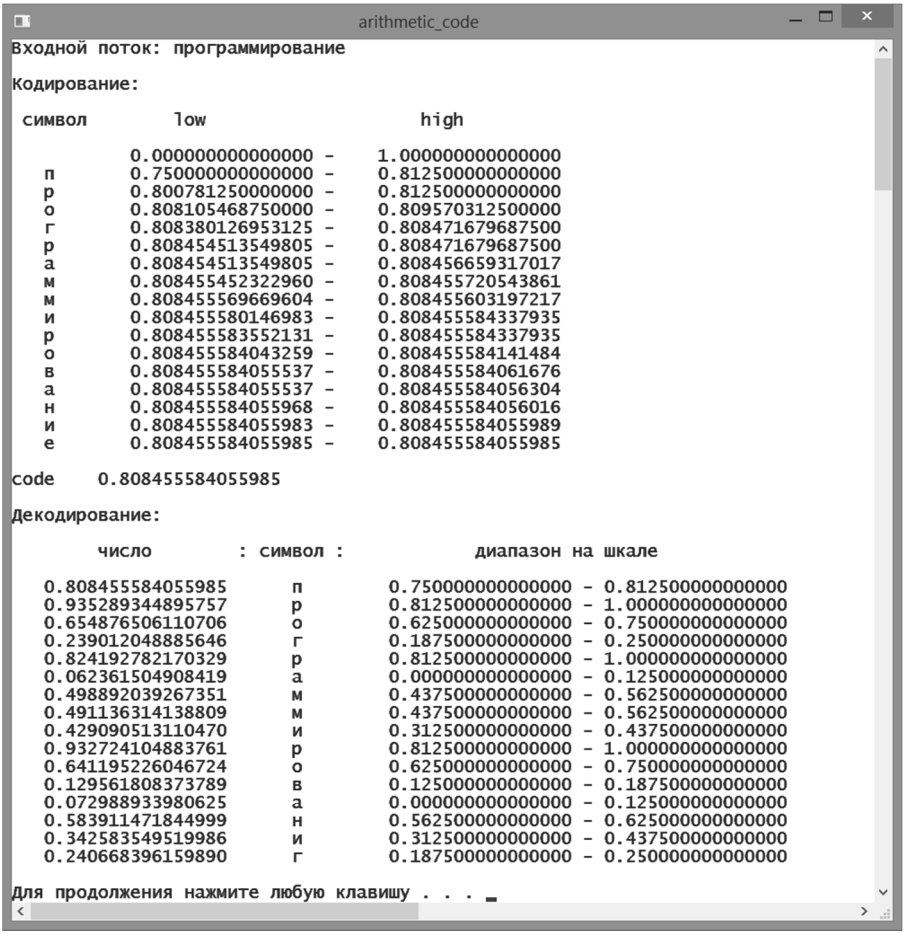


Рис. 13. Потеря точности при вещественном арифметическом кодировании

Рассмотренный пример позволяет сделать следующий вывод: чем длиннее входной поток, тем длиннее должна быть мантисса вещественного числа, которым представлен результат кодирования. В этом состоит главная проблема практического применения арифметического

кодирования, так как в аппаратных реализациях ЭВМ разрядность мантиссы ограничена[19]. Например, для типа float (4 байта) она содержит 6–7, а для типа double (8 байт) — 15–16 десятичных разрядов[20]. Следствием этого ограничения является потеря точности из-за исчезновения порядка при кодировании и вытекающая из этого невозможность последующего декодирования (рис. 13).

Чтобы избежать потери точности в рамках вещественной реализации арифметического кодирования, необходимо периодически, и довольно часто, выводить в выходной поток кодера вещественное число, находящееся «на грани» потери точности, а затем восстанавливать исходное состояние интервала кодирования. Понятно, что такой подход не способствует достижению высоких показателей степени сжатия, так как в этом случае итоговый результат работы кодера вместо одного вещественного числа будет представлен десятками, сотнями и тысячами вещественных чисел. Выход из этого затруднительного положения возможен, если удастся найти способ представления и работы с вещественными числами бесконечной разрядности.

3.3.1.1. Целочисленное арифметическое кодирование

Просто сосредоточься и следи, как цифры бегут перед глазами. Они тебе все расскажут.

Харуки Мураками. «1Q84»

Итак, серьезным препятствием для практического применения описанных выше алгоритмов арифметического кодирования/декодирования является потеря точности представления результата, вызванная ограниченным форматом представления вещественных чисел. Эксперименты показывают, что эта проблема начинает проявляться, когда длина входного потока кодера превышает 10–12 символов. К тому же, форматы

---

[19] Единственное исключение из этого правила — ЭВМ «Мир», выпускавшаяся в СССР в шестидесятых-семидесятых годах прошлого века. Эта ЭВМ могла выполнять операции с числами произвольной длины, ограниченной только объемом оперативной памяти.

[20] https://docs.microsoft.com/ru-ru/cpp/c-language/type-float?view=vs-2019.

представления чисел с плавающей точкой могут быть разными для разных линеек компьютеров. Поэтому на практике используются версии этих алгоритмов, работающие с целыми числами бесконечной длины. Алгоритм кодирования в этом случае реализует рекуррентную схему вытеснения в выходной поток старшего разряда дробной части вещественного числа (целая часть полагается равной нулю) за счет добавления нового младшего разряда, в результате чего формируется поток разрядов, представляющий сколь угодно длинное целое число.

3.3.1.1.1. Алгоритм целочисленного кодирования

Рассмотренный ранее алгоритм кодирования отслеживает значение верхней и нижней границ диапазона, содержащего возможное выходное число. Этот диапазон инициализируется значениями 0 (нижняя граница) и 1 (верхняя граница). Целочисленная версия этого алгоритма базируется на следующих допущениях.

1. Значение верхней границы, которая, в соответствии с (2), не включена в диапазон, принимается равным бесконечной десятичной дроби 0.999…, или 0.111… в двоичном представлении, или 0.FFF… в 16-ричном представлении.
2. В целочисленном представлении хранится только дробная часть нижней и верхней границ интервала, так как целая часть всегда равна 0 (предполагаемая точка, отделяющая целую часть от дробной, в таком формате находится слева).
3. Начальные значения нижней и верхней границ формируются как минимальное и максимальное представление целого числа без знака в одном из форматов фиксированной длины. Например, в формате двухбайтового целого это 0x0000 и 0xFFFF соответственно.
4. По мере вытеснения старших разрядов в выходной поток значения нижней границы дополняются справа нулями, а значения верхней границы двоичными единицами, десятичными девятками или 16-ричными F в зависимости от основания используемой системы счисления.

Рассмотрим применение этих допущений на тестовом примере «арифметика», воспользовавшись для наглядности пятиразрядным представлением десятичного целого числа. С учетом допущения 2, верхняя граница диапазона (0.99999) трансформируется в 99999, а нижняя (0.00000) — в 00000, соответственно.

Определим ширину интервала с помощью приведенного выше алгоритма арифметического кодирования. Учитывая допущение о бесконечнозначном представлении границ, ширина будет равна 100000 (подразумевается присутствие десятичной точки после старшего разряда), а не 99999 (точка подразумевается слева). Недостающую единицу младшего разряда следует добавить при вычислении.

Определим верхнюю границу интервала после кодирования символа «а» по формуле:

high = low + high_range(symbol).

Полученному значению .20 соответствует пятиразрядное десятичное 20000, которое перед сохранением следует уменьшить на 1, чтобы отразить факт «невключения» верхней границы. Поэтому значение верхней границы интервала будет 19999. Вычисление нижней границы по формуле low = low + low_range(symbol) дает значение 0.

Можно заметить, что старшие разряды в целочисленном представлении верхней и нижней границы не совпадают, поэтому на первом шаге значение старшего разряда в выходной поток не вытесняется. Это происходит только после кодирования символа «р». Так как в дальнейшем, в соответствии с алгоритмом кодирования, верхняя и нижняя границы будут продолжать сближение, совпадающие старшие разряды останутся неизменными. Поэтому можно вывести значение старшего разряда в выходной поток в качестве первого разряда кодирующего числа, а целочисленные значения верхней и нижней границ интервала сдвинуть на один разряд влево, дополнив значение high девяткой в младшем разряде. Аналогичные операции можно проделывать в любой позиционной системе счисления. По мере продолжения кодирования, верхняя и нижняя границы интервала непрерывно сближаются, в результате чего всё новые и новые совпадающие старшие разряды вытесняются в выходной поток. Пошаговые результаты кодирования тестового входного потока «арифметика» показаны на рис. 14. Обратите внимание,

что вытеснение очередного разряда в выходной поток происходит только при совпадении значений старшего разряда на нижней и верхней границах, а ширина интервалов, с учетом сделанных выше допущений о целочисленном представлении мантиссы вещественного числа, совпадает с соответствующими значениями, полученными в процессе вещественного кодирования (рис. 11).

После обработки всех символов входного потока к закодированному целочисленному представлению следует добавить два старших разряда из представления нижней границы, чтобы декодер смог правильно восстановить последний символ. Полученный результат (1498513600) внешне идентичен результату, приведенному на рис. 11. Однако если последний хранится в формате вещественного числа с плавающей точкой и поэтому подвержен ошибкам округления, целочисленное представление дробной части с неограниченной разрядностью свободно от этого недостатка.

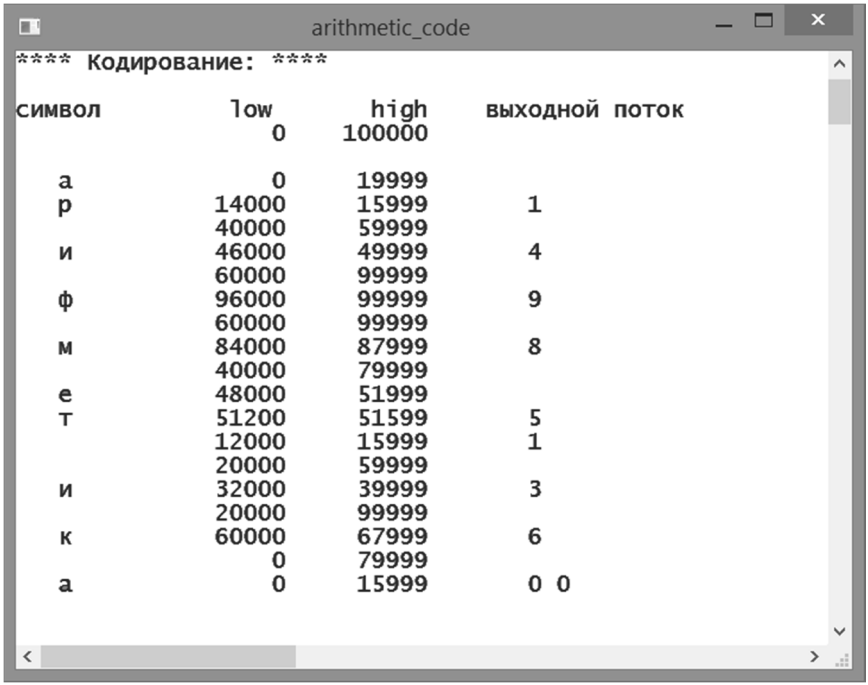


Рис. 14. Трассировка процесса целочисленного арифметического кодирования

Описанная схема инкрементного арифметического кодирования обладает достаточной точностью при работе с длинными целыми, однако существует исключительная ситуация, в которой потеря точности все же возможна. Это ситуация, при которой систематическое сужение интервала кодирования не приводит к совпадению значений старших разрядов верхней и нижней границ. Например, десятичная нижняя граница равна 69999, а верхняя — 70000. Ситуация, когда старшие разряды отличаются на единицу, следующий разряд верхней границы содержит 0, а следующий разряд нижней границы содержит 9, требует особой обработки. Эта обработка сводится к удалению второго по старшинству разряда из текущего значения верхней и нижней границ со сдвигом младших разрядов влево и установкой в 1 счетчика исчезновения порядка. При этом старший разряд в записи значений верхней и нижней границ остается на своем месте. Процедуру обработки иллюстрирует табл. 10 [1].

Таблица 10

Обработка исключительной ситуации исчезновения порядка

	До обработки	После обработки
Нижняя граница	39810	38100
Верхняя граница	40344	43449
Счетчик исчезновения порядка	0	1

Если после пересчета границ исключительная ситуация сохраняется, вновь выполняется сдвиг младших разрядов влево, и счетчик исчезновения порядка инкрементируется. Когда старшие разряды, в конце концов, сойдутся к одному значению, это значение отправляется в выходной поток, а вслед за ним туда же отправляются отброшенные ранее разряды, породившие ситуацию исчезновения порядка. Количество этих разрядов равно значению счетчика исчезновения порядка. Если совпадение произошло по значению старшего разряда на верхней границе, в выходной поток отправляются, в зависимости от основания используемой системы счисления, десятичные девятки, шестнадцатеричные F или двоичные единицы. Если совпадение произошло по значению старшего разряда на нижней границе, в выходной поток отправляются нули.

3.3.1.1.2. Алгоритм целочисленного декодирования

Идеальный процесс декодирования должен работать со всем закодированным числом, выполняя операции типа «разделить закодированное число на вероятность символа»:

value = value / range;

Практически операции с числами бесконечно большой разрядности невозможны. Однако декодер, как и кодер, может работать с целыми числами в формате 16, 32 или 64 бит. Для этого кроме значений нижней и верхней границ интервала ему требуется целочисленная переменная, содержащая 16 (32, 64) текущих бита кодового числа из входного потока декодера. Кодовое число всегда находится между нижней и верхней границами. По мере приближения границ к кодовому числу будут выполняться новые операции сдвига, отодвигающие обе границы от значения кода.

Декодер обновляет значения верхней и нижней границ интервала после декодирования каждого очередного символа точно так же, как это делает кодер. Обновленные значения, полученные на соответствующих шагах кодирования/декодирования, должны совпадать. Сравнивая старшие разряды верхней и нижней границы, декодер выясняет, когда следует выполнять очередной поразрядный сдвиг во входном коде. Кроме того, он выполняет те же самые тесты на исчезновение порядка, что и кодер. Процесс целочисленного декодирования для входного потока дробных десятичных разрядов 1498513600 иллюстрирует рис. 15.

В идеальном алгоритме декодирования текущий символ распознается в результате нахождения интервала, заключающего текущее значение кода. С использованием целочисленной математики проблема распознавания решается несколько сложнее. В этом случае границы шкалы распределения вероятностей определяются не диапазоном 0..1, а диапазоном между двумя целочисленными значениями длиной 16 (32, 64) битов. Положение текущего кода в этом диапазоне определяет значение вероятности, которое используется для идентификации символа, в соответствии с выражением:

P = (value – low) / (high – low + 1).

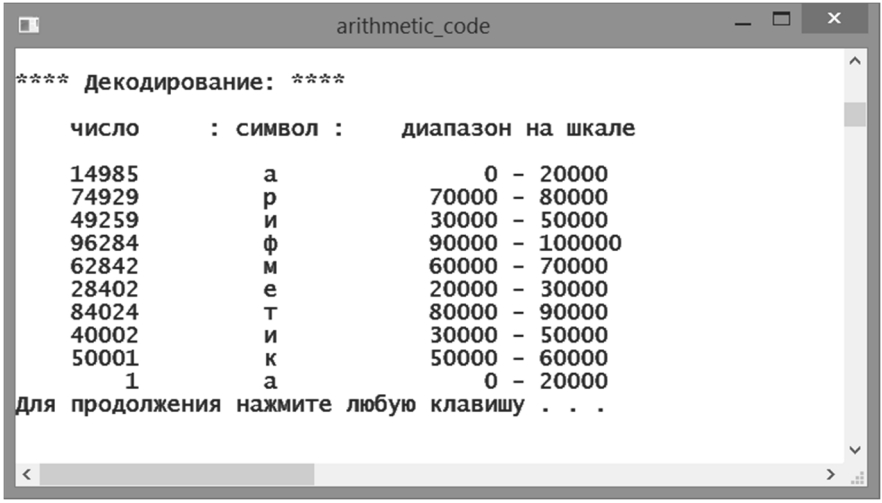


Рис. 15. Трассировка процесса целочисленного арифметического декодирования

3.3.1.1.3. Проблемы реализации

Основной проблемой целочисленной реализации арифметического кодирования является выбор такой разрядности представления величин, вовлеченных в вычисления, которая обеспечит необходимую точность эмуляции операций с плавающей точкой, и тем самым — возможность последующего декодирования, и при этом не вызовет переполнение разрядной сетки. С учетом последнего обстоятельства в обсуждаемой далее бинарной реализации использовано следующее правило выбора разрядности: количество битов в представлении частот плюс количество битов для представления значений границ не должно превышать количества битов для выполнения арифметических операций в процессе кодирования.

В соответствии с этим правилом для вычислений используется 64-битный тип unsigned long long. Значение ширины шкалы распределения вероятностей (ШРВ) определено максимальным 32-битным значением, следовательно, и границы не могут выходить за этот диапазон. Это ограничение требует, чтобы представление длины входного потока было хотя бы на 1 меньше ширины шкалы. Удовлетворение этого требования

может быть достигнуто путем масштабирования таблицы частот в процессе сбора частотной статистики.

Кодер

Кодер начинает работу с построения модели кодирования, реализованного в функции statistic(). Она сканирует входной поток для построения частотной таблицы stat и выводит эту таблицу в заголовок выходного файла. Таблица содержит 257 элементов: первые 256 соответствуют байт-кодам, которые могут присутствовать во входном потоке, а последний элемент соответствует служебному коду eof и имеет частоту 1. Построенная частотная таблица масштабируется в соответствии с максимальным значением ширины шкалы max_scale.

```
void statistic()
{
  unsigned char c;
  long long count = reader->getfilelength();
  for (long long i = 0; i < count; i++)
  {
    reader->readbyte(&c);
    stat[c]++;
  }
  stat[256] = 1;
  count++;
  // запись статистики в выходной поток
  // пишем 4 байта из 8
  for (int i = 0; i < 256; i++)
    writer->writebyte((long long*)&stat[i], 4);
  for (int i = 0; i < 257; i++)
    stat[i] = stat[i]* max_scale / count;
  reader->reset();
}
```

Далее для удобства последующей идентификации интервала, соответствующего символу, функция setverify() перестраивает частотную таблицу stat с учетом кумулятивного эффекта накопления частот, определяя для каждого элемента таблицы нижнюю границу соответствующего интервала как сумму частот предшествующих элементов. Верхняя граница интервала i совпадает с нижней границей интервала i+1, которому значение границы и принадлежит.

```
void setverify()
{
  long long low = 0;
  for (int i = 0; i < 258; i++)
  {
    unsigned long long tmp = stat[i];
    stat[i] = low;
    low = stat[i] + tmp;
  }
}
```

Собственно арифметическое кодирование выполняет функция encode(). Используя данные из таблицы stat, она инициализирует значения нижней (low) и верхней (high) границ ШРВ, а также значение ширины шкалы scale.

```
  unsigned long long low = 0;
  unsigned long long scale = stat[257];
  unsigned long long high = scale;
```

Кроме того, она инициализирует нулем счетчик битов исчезновения порядка underflow_bits, открывает входной и создает выходной потоки и выполняет описанные выше операции сбора статистики и перестройки построенной таблицы stat для удобства идентификации интервала, соответствующего прочитанному символу. После этих подготовительных операций запускается сам цикл кодирования. Количество повторений цикла определяется как filelen+1, где filelen — длина входного потока. Последний проход цикла эмулирует считывание служебного символа eof.

Вслед за считыванием каждого символа кодер определяет текущее значение ширины шкалы scale, верхнюю (high) и нижнюю (low) границы соответствующего символу интервала.

```
unsigned long long range = high - low + 1;
high = low + stat[c + 1] * range / scale - 1;
low = low + stat[c] * range / scale;
```

После пересчета границ интервала может возникнуть ситуация, требующая вытеснения бита кодового слова в выходной поток. Так как количество вытесняемых битов заранее неизвестно, кодер выполняет вытеснение в бесконечном цикле while, условием выхода из которого

является отсутствие разрядов для вытеснения. Необходимость в отправке бита в выходной поток возникает в двух случаях.

1. Если значения старших битов low и high совпадают, бит вытесняется в выходной поток:

```
writer->writebit(rang_low);
```

2. Вслед за этим выполняется цикл обработки битов исчезновения порядка. Количество повторений этого цикла определяется значением счетчика underflow_bits. При каждом выполнении цикла в выходной поток отправляется инвертированный старший бит high, т. е. если high и low сошлись с нулем в старшем разряде, выводятся единицы, в противном случае нули, при этом значение underflow_bits декрементируется.

```
while (underflow_bits)
{
  // инвертированный старший бит
  bit = (~high & 0x80000000) >> 31;
  writer->writebit(bit);
  underflow_bits--;
}
```

Если значения старших битов не совпадают (старший бит low равен 0, а старший бит high — 1), а значения следующих по старшинству битов противоположны, причем второй по старшинству бит low равен 1, а второй по старшинству бит high равен 0, имеет место ситуация исчезновения порядка. В этой ситуации алгоритм целочисленного арифметического кодирования предписывает сдвиг 30 младших двоичных разрядов на одну позицию влево, при этом старший бит должен остаться на своем месте. Программирование этого сдвига можно упростить, если установить второй бит low и high в значение старшего бита на соответствующей границе. В результате кодовое слово для low примет вид 00XXX…, а для high — 11XXX, где XXX… — несущественные для выполнения операции значения оставшихся разрядов. Результат сдвига на один двоичный разряд влево модифицированных таким образом low и high равносилен исключению второго по старшинству разряда. После выполнения каждой такой операции счетчик underflow_bits инкрементируется.

```
low &= 0x3fffffff;   // обнуляем второй бит
```

```
        high |= 0x40000000;  // второй бит устанавливаем в 1
        underflow_bits++;
```

Независимо от того, возникла ли в результате чтения очередного символа ситуация, требующая записи в выходной поток совпадающего старшего бита low и high, или же имела место ситуация исчезновения порядка, значения low и high сдвигаются на 1 двоичный разряд влево. При этом значение младшего бита high устанавливается в 1.

```
        low <<= 1;
        high <<= 1;
        high |= 1;
```

После выхода из цикла чтения входного потока кодер вызывает процедуру flush_encoder(), которая выполняет обработку оставшихся битов исчезновения порядка.

```
  void flush_encoder(unsigned long long low, unsigned int under-
flow_bits)
  {
    unsigned char bit = low & 0x40000000 >> 31;
    writer->writebit(bit);
    underflow_bits++;
    while (underflow_bits-- > 0)
    {
      bit = ~low & 0x40000000 >> 31;
      writer->writebit(~low & 0x40000000);
    }
  }
```

Ниже приведен полный текст функции encode().

```
  int encode()
  {
    unsigned long long low = 0;
    unsigned int underflow_bits = 0;
    unsigned char bit;
    int i = 0;

    reader = new Reader(fnamein, 1);
    if (!reader->isopen()) return 1;
    if (!PutFileExtension()) return 2;

    statistic();
    setverify();

    unsigned long long scale = stat[257];
```

```
unsigned long long high = scale;
unsigned char cd = 0;
short c = 0;
unsigned long long rang_low, rang_high;

long long filelen = reader->getfilelength();
// для уверенного декодирования выполняем
// одну дополнительную итерацию
for (long long i = 0; i <= filelen + 1; i++)
{
  if (i < filelen)
    reader->readbyte((unsigned char*)&c, 1);
  else
  {
    c = 256;
  }
  unsigned long long range = high - low + 1;

  high = low + stat[c + 1] * range / scale - 1;
  low = low + stat[c] * range / scale;
  while (true)
  {
    rang_low = low >> 31;  // выделяем старшие биты
    rang_high = high >> 31;
    if (rang_low == rang_high)
    {
      writer->writebit(rang_low);
      while (underflow_bits)
      {
        bit = (~high & 0x80000000) >> 31;
        writer->writebit(bit);
        underflow_bits--;
      }
    }
    else
    {
      if ((low & 0x40000000) == 0x40000000 &&
           (high & 0x40000000) == 0)
      {
        low &= 0x3fffffff;
        high |= 0x40000000;
        underflow_bits++;
      }
      else break;
    }
    int gg = 0;
```

```
      low <<= 1;
      high <<= 1;
      high |= 1;
      // отсекаем биты, превышающие 32 разряда
      low &= 0xffffffff;
      high &= 0xffffffff;
    }
  }
  flush_encoder(low, underflow_bits);
  writer->flush();
  writer->close();
  reader->close();
  return 0;
}
```

Декодер

Декодер начинает свою работу с построения модели кодирования. Для этого он вызывает функцию readstat(), которая читает из входного потока таблицу статистики stat, масштабирует ее под максимальный размер шкалы и возвращает длину несжатого файла для последующей проверки корректности декодирования.

```
unsigned long long readstat()
{
  unsigned long long count = 0;
  // читаем по 4 байта
  for (int i = 0; i < 256; i++)
  {
    reader->readbyte((long long*)&stat[i], 4);
    count += stat[i];
  }
  stat[256] = 1;
  count++;
  for (int i = 0; i < 257; i++)
    stat[i] = stat[i] * max_scale / count;
  return count - 1;   // длина исходного файла
}
```

Затем декодер при помощи описанной выше функции setverify() перестраивает частотную таблицу stat с учетом кумулятивного эффекта накопления частот, инициализирует нижнюю (low) и верхнюю (high) границы, а также её ширину (scale) и считывает начальную часть (первые 32 разряда) кодового слова (code).

```
  for (int i = 0; i < bitlen; ++i)
```

```
        {
          unsigned char bit = reader->readbit();
          value = (value << 1) | bit;
        }
        code = value;
```

Цикл восстановления сжатого файла управляется значением флага end_of_file, который перед входом в цикл сбрасывается и устанавливается после декодирования служебного символа eof. Внутри цикла декодер приводит кодовое слово value в соответствие с текущими значениями интервала, находит интервал ШРВ, соответствующий значению value, и отправляет декодированный символ в выходной поток, пересчитывает границы интервала и обрабатывает ситуации, требующие сдвига кодового слова. Так же как и кодер, декодер в качестве таких ситуаций рассматривает в бесконечном цикле совпадение старших разрядов верхней и нижней границ интервала и исчезновение порядка. Но, в отличие от кодера, который пишет вытесняемые биты в выходной поток и считает количество битов исчезновения порядка, декодер считывает очередной бит из входного потока и добавляет его к кодовому слову, предварительно смещенному на 1 бит влево.

```
while (true)
{
  rang_low = low >> 31;
  rang_high = high >> 31;
  if (rang_low == rang_high)
  {
  }
  else
  if ((low & 0x40000000) == 0x40000000 &&
      (high & 0x40000000) == 0)
  {
    low &= 0x3fffffff;
    high |= 0x40000000;
    code ^= 0x40000000;
  }
  else break;
  low <<= 1;
  high <<= 1;
  high |= 1;
  // отсекаем биты, превышающие 32 разряда
  low &= 0xffffffff;
  high &= 0xffffffff;
  // если при считывании достигнут конец файл
```

```
  if ((c = reader->readbit()) == 2)
    end_of_file = true;
  code = (code << 1) + c;
  code &= 0xffffffff;
}
```

Цикл восстановления сжатого файла заканчивает работу, если декодер распознал служебный символ eof. Если количество фактически декодированных символов count совпадает с длиной несжатого файла countbyte, вычисленной функцией readstat(), функция decode() возвращает нормальный код завершения, иначе возвращает код, который интерпретируется как ошибка восстановления сжатого файла.

Ниже приведен полный текст функции decode():

```
int decode()
{
  bool eqw = true;
  long long countbyte; // количество байтов в несжатом файле
  reader = new Reader(fnamein);
  if (!reader->isopen()) return 1;
  if (!GetFileExtension()) return 2;
  countbyte = readstat();
  setverify();
  long long count = reader->getfilelength();
  unsigned char c;
  unsigned long long scale = stat[257];
  unsigned long long code, range, rang_low, rang_high;
  unsigned long long i, low = 0, high = scale;
  unsigned long long value = 0;
  bool flag = false;
  for (int i = 0; i < bitlen; ++i)
  {
    unsigned char bit = reader->readbit();
    value = (value << 1) | bit;
  }
  code = value;
  c = 0;
  count = 0;
  bool end_of_file = false;
  while (!end_of_file)
  {
    int i = 0;
    range = high - low + 1;
    value = ((code - low + 1)*scale - 1) / range;
    for (i = 0; i < 257; i++)
    if (value >= stat[i] && value < stat[i+1])
```

```
    {
      if (i == 256)
        end_of_file = true;
      else
      {
        writer->writebyte((unsigned char*)&i);
        count++;
      }
      break;
    }
    range = high - low + 1;
    high = low + stat[i + 1]* range / scale - 1;
    low = low + stat[i] * range / scale;
    while (true)
    {
      rang_low = low >> 31;
      rang_high = high >> 31;
      if (rang_low == rang_high)
      {
      }
      else
      if ((low & 0x40000000) == 0x40000000 &&
          (high & 0x40000000) == 0)
      {
        {
          low &= 0x3fffffff;
          high |= 0x40000000;
          code ^= 0x40000000;
        }
      }
      else break;
      low <<= 1;
      high <<= 1;
      high |= 1;
      // отсекаем биты, превышающие 32 разряда
      low &= 0xffffffff;
      high &= 0xffffffff;
      if ((c = reader->readbit()) == 2)
              // при считывании достигнут конец файла
        end_of_file = true;
      code = (code << 1) + c;
      code &= 0xffffffff;
    }
  }
  if (count == countbyte) return 0;
  else return 3;
}
```

3.3.1.2. Адаптивное целочисленное арифметическое кодирование

Динамический подход исходит из того, что априорно неизвестная модель кодирования идентичным способом инициализируется на стороне кодера и декодера до начала соответствующего процесса, а затем систематически обновляется, адаптируясь к собранной статистике внутри этого процесса. Таким образом, ключевые моменты реализации адаптивного арифметического кодирования — это процедуры инициализации и обновления модели кодирования.

Так как результатом арифметического кодирования является единственное число из интервала 0..1, вопрос о выборе способа инициализации модели кодирования проще всего решить в пользу гипотезы о равной вероятности появления любого символа во входном потоке. Напомним, что при выборе способа инициализации модели кодирования для динамического алгоритма Хаффмана этот вариант инициализации был отвергнут из-за того, что для небольших файлов степень сжатия может оказаться небольшой или вовсе отсутствовать, так как длина кодов Хаффмана на начальном участке кодирования/декодирования будет приблизительно равна длине байта. В динамической реализации арифметического кодирования такой выбор не повлияет на степень сжатия, и поэтому ввиду исключительной простоты именно он будет обсуждаться далее. Все, что следует сделать кодеру и декодеру для инициализации модели — это разбить шкалу единичной ширины на равные интервалы по количеству символов в алфавите входного потока и принять, что частота каждого из символов во входном потоке равна 1. Приведенный ниже код функции setverify() инициализирует равномерное разбиение с учетом кумулятивного эффекта накопления частот, определяя для каждого элемента таблицы нижнюю границу соответствующего интервала как сумму частот предшествующих элементов.

```
void setverify()
{
  long long low = 0;
  for (int i = 0; i < 258; i++)
    stat[i] = i;
}
```

Ширина интервала, отведенного символу c_i, изменяется после каждого его кодирования и декодирования, и в каждый момент времени определяется соотношением

$$w_i = \frac{f_i}{\sum_{k=1}^{N} f_k},$$

где f_i — текущее значение частоты символа c_i, N — количество символов в алфавите. Вследствие этого возникает необходимость обновления модели кодирования на каждом шаге. Это отрицательно сказывается на вычислительной эффективности адаптивного кодера и декодера, так как связано с линейным сканированием массивов, размер которых равен N. Ниже приведен код функции updateverify (unsigned char c), обновляющей статистику с учетом кумулятивного эффекта после считывания очередного символа *c*.

```
void updateverify(unsigned char c)
{
  for (int i = c+1; i < 258; i++)
    stat[i]++;
}
```

За исключением отмеченных различий, адаптивный арифметический кодер реализует ту же самую логику формирования кодового слова. Ниже приведен фрагмент функции encode(), реализующий процесс кодирования входного потока.

```
  for (long long i = 0; i <= filelen + 1; i++)
  {
    if (i < filelen)
      reader->readbyte((unsigned char*)&c, 1);
    else
      c = 256;
    unsigned long long range = high - low + 1;
    high = low + stat[c + 1] * range / stat[257] - 1;
    low = low + stat[c] * range / stat[257];
    updateverifity(c);
    while (true)
    {
      rang_low = low >> 31;
      rang_high = high >> 31;
      if (rang_low == rang_high)
```

```
    {
      writer->writebit(rang_low);
      while (underflow_bits)
      {
        bit = (~high & 0x80000000) >> 31; // старший бит
        writer->writebit(bit);
        underflow_bits--;
      }
    }
    else
      if ((low & 0x40000000) == 0x40000000 &&
          (high & 0x40000000) == 0)
      {
        low &= 0x3fffffff;
        high |= 0x40000000;
        underflow_bits++;
      }
      else break;
    low <<= 1;
    high <<= 1;
    high |= 1;
    // отсекаем биты, превышающие 32 разряда
    low &= 0xffffffff;
    high &= 0xffffffff;
  }
}
```

Декодер, опираясь на те же данные инициализации, будет адаптировать их по мере соотнесения входного числа с текущим разбиением шкалы. Ниже приведен код основного цикла функции декодирования.

```
while (!end_of_file)
{
  int i = 0;
  range = high - low + 1;
  value = ((code - low + 1) * stat[257] - 1) / range;
  for (i = 0; i < 257; i++)
    if (value >= stat[i] && value < stat[i + 1])
    {
      if (i == 256)
        end_of_file = true;
      else
      {
        writer->writebyte((unsigned char*)&i);
        count++;
      }
```

```
      break;
    }
  range = high - low + 1;
  high = low + stat[i + 1] * range / stat[257] - 1;
  low = low + stat[i] * range / stat[257];
  updateverifity(i);
  while (true)
  {
    rang_low = low >> 31;
    rang_high = high >> 31;
    if (rang_low == rang_high)
    {
    }
    else
      if ((low & 0x40000000) == 0x40000000 &&
          (high & 0x40000000) == 0)
      {
        low &= 0x3fffffff;
        high |= 0x40000000;
        code ^= 0x40000000;
      }
      else break;
    low <<= 1;
    high <<= 1;
    high |= 1;
    // отсекаем биты, превышающие 32 разряда
    low &= 0xffffffff;
    high &= 0xffffffff;
    if ((c = reader->readbit()) == 2)
      end_of_file = true;
    code = (code << 1) + c;
    code &= 0xffffffff;
  }
}
```

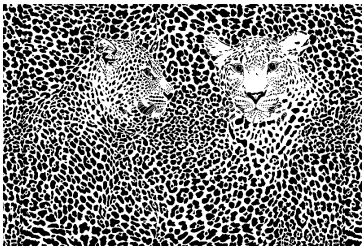
3.4. Обсуждение результатов

Анализ представленных в табл. 11 результатов позволяет сделать вывод о незначительном преимуществе адаптивного алгоритма в достигаемой степени сжатия. Однако если учесть, что в сжатый статическим алгоритмом файл входит частотная таблица (1024 байта), то окажется, что «чистый результат» статического алгоритма, например, для текстового файла (40,67%) лучше, чем у его адаптивного конкурента (40,27%). С ростом размера сжимаемого файла «удельный вес» заголовка файла,

сжатого статическим алгоритмом, будет уменьшаться. Это позволяет предположить, что и так не слишком большое преимущество адаптивного алгоритма в степени сжатия для больших файлов будет сведено на нет. Если же сравнивать степень сжатия файлов при помощи арифметического кодирования с результатами применения других алгоритмов частотной группы, то арифметический алгоритм выглядит чуть-чуть лучше своих конкурентов. По-видимому, сказывается возможность кодирования символа дробным числом битов.

Таблица 11

Результаты применения алгоритма арифметического кодирования

Формат файла	Степень сжатия, %	
	Статический алгоритм	Адаптивный алгоритм
 Растр	79,95	80,73
 Растр	36,92	38,41
Текст	39,11	40,27
Исполняемый формат	46,96	48,31

3.5. Вопросы и задания для самоконтроля

1. За счет чего достигается эффект сжатия данных при использовании частотных алгоритмов?
2. Что такое частотная таблица (вектор частот)? Можно ли ее считать моделью кодирования?
3. Дайте определение префиксного кода и дерева префиксного кодирования.
4. Какова максимальная длина префиксного кода? При каких условиях она может быть достигнута?
5. Какие проблемы, связанные с использованием кодов переменной длины, возникают у кодера и декодера? Как они решаются?
6. Какой прием, систематически используемый в процессе построения дерева префиксных кодов в рамках статического алгоритма Шеннона — Фано, обеспечивает обратную зависимость между частотой символа в потоке данных и длиной его кода?
7. Постройте частотную таблицу и используйте ее для построения дерева префиксных кодов в рамках статического алгоритма Шеннона — Фано для потока данных «МАТЕМАТИКА». Код какого символа оказался самым коротким? Почему?
8. Какой прием, систематически используемый в процессе построения дерева префиксных кодов в рамках статического алгоритма Хаффмана, обеспечивает обратную зависимость между частотой символа в потоке данных и длиной его кода?
9. Постройте частотную таблицу и используйте ее для построения дерева префиксных кодов в рамках статического алгоритма Хаффмана для потока данных «МАТЕМАТИКА». Код какого символа оказался самым коротким? Почему?
10. Какой из двух статических частотных методов — метод Шеннона — Фано или метод Хаффмана — получил более широкое распространение и почему?
11. Почему в статических алгоритмах необходима передача декодеру модели кодирования? Как это влияет на величину степени сжатия?

12. Какие форматы модели кодирования используют для передачи декодеру статические частотные алгоритмы? Какой из них наиболее предпочтителен? Почему?
13. Примените дерево префиксных кодов, построенное алгоритмом Шеннона — Фано (Хаффмана), чтобы закодировать поток данных «МАТЕМАТИКА». Оцените полученную степень сжатия без учета передачи модели кодирования. С помощью той же модели выполните декодирование.
14. Какое соглашение позволяет адаптивному алгоритму Хаффмана обойтись без передачи модели кодирования декодеру?
15. Какие способы инициализации модели кодирования для адаптивного алгоритма Хаффмана возможны? Какой из них более предпочтителен? Почему?
16. В чем заключается свойство упорядоченности дерева префиксного кодирования?
17. В чем разница между процедурами обновления модели символом «в контексте» модели кодирования и символом «вне контекста»?
18. С чем связано изменение кода символа по мере адаптивного кодирования? Как при этом изменяется длина кода?
19. Закодируйте поток данных «МАТЕМАТИКА» с помощью адаптивного алгоритма Хаффмана. Для динамического построения дерева префиксных кодов используйте процедуры инициализации и обновления модели символом. Оцените полученную степень сжатия, сравнив ее с ранее полученными для этого примера результатами.
20. Декодируйте поток данных, полученный в результате выполнения предыдущего задания, с помощью адаптивного алгоритма Хаффмана.
21. В чем заключаются проблемы работы с кодами переменной длины? Как они решаются кодером и декодером частотных методов?
22. Какие операции входят в интерфейс абстрактного типа данных «Дерево Хаффмана»?
23. Какая структура данных лежит в основе реализации абстрактного типа данных «Дерево Хаффмана»? Опишите ее компоненты и связи между ними.

24. В чем заключается принципиальное отличие алгоритма арифметического кодирования от других алгоритмов статистической группы? Как это отличие влияет на степень сжатия, обеспечиваемую арифметическим алгоритмом?
25. Что представляет собой модель кодирования для арифметического метода?
26. Что представляет собой результат кодирования для арифметического метода?
27. В чем причина возникновения ситуации исчезновения порядка при арифметическом кодировании? Какие допущения позволяют исключить причину ее возникновения?
28. Что такое инкрементное арифметическое кодирование? Какая исключительная ситуация может возникнуть в процессе инкрементного кодирования? Как она обрабатывается?
29. Закодируйте входной поток «МАТЕМАТИКА» вещественным числом при помощи статического арифметического алгоритма. Результат кодирования представьте в виде двоичной дроби. Оцените среднее количество битов, затраченное на кодирование каждого символа. Выполните декодирование.
30. Как арифметический декодер решает проблему обнаружения конца декодируемого файла?

4. Рекомендации по выполнению курсового проекта

Целью выполнения курсового проекта является освоение навыков программной реализации алгоритмов сжатия и исследования зависимости степени сжатия данных от параметров настройки алгоритма. Рекомендуемые варианты заданий на курсовое проектирование приведены в разделе П.1. Формулировки заданий предусматривают разработку кодера и декодера конкретного алгоритма как различные варианты задания, хотя при наличии достаточного количества часов, отведенных на выполнение проекта, это ограничение может быть снято. Однако если оно остается в силе, для выполнения работ по проекту необходимо организовать команды из двух исполнителей, один из которых разрабатывает кодер, а второй — декодер. Для успешного выполнения проекта члены команды должны согласовать форматы передачи данных, при реализации динамических алгоритмов — соглашения о реализации процедур, используемых как кодером, так и декодером, а также соглашения о формате кода, который бы допускал возможность последующего объединения результатов программной реализации.

Характерной особенностью всех рассмотренных алгоритмов сжатия является относительная простота исходной концепции и достаточно высокая сложность программной реализации этой концепции, которая обеспечивала бы высокую эффективность алгоритма как в плане достижения целевого показателя (степени сжатия), так и в вычислительном аспекте. Поэтому выполнение курсового проекта целесообразно организовать в соответствии со спиральной моделью разработки программного обеспечения. Использование этой модели позволит учесть такие связанные с программной реализацией алгоритмов сжатия риски, как недостаточная эффективность алгоритма, с одной стороны, и неоправданное стремление к совершенству — с другой. Кроме того, использование спиральной модели позволяет построить процесс разработки в виде серии итераций, на каждой из которых последовательно определяется цель (эталонный результат разработки по завершении итерации), специфици-

руются связанные с достижением этой цели проблемы, выполняется разработка и тестирование очередной версии и планируется следующая итерация. Применение итеративной схемы разработки имеет два основных преимущества. Во-первых, результативное завершение каждой итерации позволяет оценить требования к продукту в терминах разности его текущих и финальных характеристик, а это гораздо проще, чем сформулировать детальные требования к продукту до начала его разработки. Во-вторых, такая организация разработки наилучшим образом соответствует используемым многими вузами технологии промежуточного контроля знаний внутри семестра: гораздо проще оценивать работоспособный промежуточный результат, чем незавершенный процесс разработки. И, кроме того, не стоит сбрасывать со счетов такой фактор, как эмоциональное состояние разработчика, достигшего определенного результата, в противоположность состоянию разработчика, который находится «в процессе». С учетом сказанного, можно рекомендовать схему разработки, включающую три цикла итерации, каждый из которых должен завершаться созданием работоспособного приложения.

На *первой итерации* отрабатывается функциональная составляющая приложения при максимуме допущений, ограничивающих его эффективность. То есть создается версия, которая реализует «принцип действия» алгоритма сжатия/декомпрессии при упрощениях, способных свести на нет количественный эффект сжатия и/или вычислительную эффективность алгоритма. Учитывая относительную простоту концепций алгоритмов сжатия, можно считать результатом этой версии появление «быстрого прототипа», который, с одной стороны, может служить действующим полигоном для испытания алгоритма, а с другой — может быть использован для спецификации точек его усовершенствования на следующей итерации.

На *второй итерации* отрабатываются вопросы эффективности программной реализации алгоритма в целевом и вычислительном аспектах. С этой целью определяется план последовательного снятия допущений, сделанных на первой итерации. Учитывая потенциально достаточно большое количество первоначальных допущений, можно говорить о двух

возможных циклах «итерации эффективности», если это вписывается в принятую систему рубежного контроля знаний внутри семестра. Продуктом этих итераций должна быть рабочая версия программы, которую можно использовать для проведения испытаний с целью определения достигнутых показателей целевой и вычислительной эффективности.

На *третьей итерации* основным направлением усовершенствования продукта становится достижение необходимых эргономических показателей продукта. Если предыдущие версии могли иметь формат консольного приложения, то теперь необходимо сосредоточиться на разработке оконного интерфейса (см. раздел П.3), а также на выполнении тех функциональных требований, которые не имеют прямого отношения к сжатию данных, но создают дополнительные удобства пользователю. Например, можно озаботиться решением вопроса о передаче атрибутов сжимаемого файла от кодера декодеру через заголовочную часть файла, чтобы исключить дополнительные затраты времени пользователя на переименование декодированного файла и восстановление других его атрибутов.

С целью упрощения согласований внутри команды проекта рекомендуется использовать технологии управления версиями программного продукта, а также технологии объектно-ориентированного программирования.

Заключение

Скоро сказка сказывается,
да не скоро дело делается!
Русская пословица

Приступая к работе над этим разделом, авторы испытали те же ощущения, которые, возможно, испытывает победитель марафона, когда у него берут интервью на финише: надо что-то говорить, а все силы уже отданы борьбе за победу. Сравнительно небольшое по объему пособие насыщено подробно аннотированным программным кодом, часто весьма нетривиальным. Написание, отладка и комментирование кода действительно отняли много времени и сил, зато на финише авторы с гордостью могут сказать, что весь этот код работает (и это заслуга второго автора), и понятно, почему (на это надеется первый автор)!

Утверждение о работоспособности кода основывается на результатах проведенных испытаний, хотя, как известно, тестирование показывает наличие ошибок, но не доказывает их отсутствия. Поэтому авторы будут признательны читателям за обнаруженные ими ошибки и конструктивную критику предложенной реализации.

Сложно написать программу и заставить её работать. Не менее сложно объяснить алгоритм, лежащий в основе программной реализации, и те приёмы программирования, которые пришлось использовать, чтобы сделать алгоритм не только работоспособным, но и эффективным. Для достижения этой цели авторы использовали два проверенных жизнью принципа: «от простого — к сложному» и «практика — критерий истины». Согласно первому принципу, обсуждение каждого алгоритма авторы начинали на уровне его базовой концепции, и лишь затем переходили к обсуждению деталей её программной реализации. Согласно второму принципу, каждый этап обсуждения завершался проверкой состоятельности полученного решения путем практического применения алгоритма.

Мир алгоритмов — это искусственная вселенная, в которой движение каждого отдельно взятого объекта обусловлено движением других объектов. Поэтому, работая над материалом пособия, авторы посто-

янно боролись с искушением рассказать всё об алгоритмах и технологиях, оказывающих влияние на алгоритмы сжатия. Так как «рассказать всё» — задача заведомо невыполнимая, при отборе смежных тем авторы руководствовались собственными предпочтениями. Результаты этой субъективной фильтрации собраны в разделах Приложения, которые по объему сопоставимы с основной частью пособия.

Имели ли успех эти усилия? Стали ли авторы победителями марафона? Об этом судить читателям, для которых они старались сделать материал понятным и увлекательным. Авторы сочтут свою задачу выполненной, если их книга послужит первым шагом на пути в сложный и увлекательный мир алгоритмов сжатия данных.

Приложения

П.1. Варианты заданий для курсового проектирования

Реализовать кодер (декодер) для определенного вариантом задания алгоритма сжатия. Процесс разработки разбить на этапы. Поэтапные результаты разработки:

1. Быстрый прототип: работоспособная версия алгоритма, разработанная с учетом упрощающих допущений, определенных вариантом задания (табл. П.1).
2. Эффективная реализация алгоритма в формате консольного приложения, полностью или частично снимающего допущения, принятые на предыдущем этапе. Оценка вычислительной эффективности алгоритма.
3. Реализация алгоритма сжатия в формате оконного приложения, объединяющего кодер и декодер и предлагающего пользователю необходимые интерфейсы для выбора и сохранения файлов, мониторинга процесса и результатов сжатия и настройки параметров алгоритма, определенных вариантом задания.
4. Исследование степени сжатия, обеспечиваемой разработанным приложением, на тестовом наборе файлов. Оформление пояснительной записки.

Таблица П.1

Варианты задания на курсовое проектирование

№ вар.	Наименование алгоритма сжатия	Допущения при разработке быстрого прототипа	Параметры настройки алгоритма
1.	Кодер RLE+BWT	Длина цепочки не превышает 255 байт, длина блока равна 255 байт	Максимальная длина цепочки, размер блока BWT
2.	Декодер RLE+BWT	Длина цепочки не превышает 255 байт, длина блока равна 255 байт	Максимальная длина цепочки, размер блока BWT
3.	Кодер LZW	Поиск цепочек выполняется линейным просмотром таблицы, размер таблицы равен 64К, длина сжимаемых файлов менее 64К	Размер таблицы цепочек, максимальная длина цепочки

Продолжение табл. П.1

№ вар.	Наименование алгоритма сжатия	Допущения при разработке быстрого прототипа	Параметры настройки алгоритма
4.	Декодер LZW	Размер таблицы равен 64К	Размер таблицы цепочек, максимальная длина цепочки
5.	Кодер Шеннона — Фано	Модель кодирования передается декодеру в отдельном файле, допускается замена битов байтами в выходном потоке	
6.	Декодер Шеннона — Фано	Модель кодирования доступна в отдельном файле, допускается интерпретация байтов входного потока битами	
7.	Кодер Хаффмана статический		Формат передаваемой модели кодирования (вектор длин, вектор частот)
8.	Декодер Хаффмана статический		Формат передаваемой модели кодирования (вектор длин, вектор частот)
9.	Кодер Хаффмана адаптивный		
10.	Декодер Хаффмана адаптивный		
11.	Статический арифметический кодер		Основание системы счисления в представлении кодового числа
12.	Статический арифметический декодер		Основание системы счисления в представлении кодового числа
13.	Адаптивный арифметический кодер		Основание системы счисления в представлении кодового числа
14.	Адаптивный арифметический декодер		Основание системы счисления в представлении кодового числа

П.2. Вспомогательные алгоритмы

Дьявол кроется в деталях.
Идиома

Описанные выше алгоритмы сжатия достаточно просты на концептуальном уровне, однако их программная реализация, если она преследует цели достижения реального эффекта сжатия за приемлемое время, сталкивается со сложностями, на преодоление которых приходится тратить незапланированное время и усилия. Круг обсуждаемых в этом разделе проблем продиктован следующими соображениями.

Самая большая сложность связана с необходимостью побитовой записи и побитового чтения потоков данных, присутствующей практически в каждом из рассмотренных алгоритмов. Крайне желательно «спрятать» детали реализации этой сложности таким образом, чтобы «на поверхности» остался простой и неизбыточный интерфейс, не затеняющий логику алгоритма сжатия. Адекватной методологией, обеспечивающей такое абстрагирование, является объектно-ориентированный подход.

Еще одно затруднение, встречающееся в ходе программной реализации алгоритмов сжатия, состоит в необходимости формирования заголовочной части сжатого файла, через которую кодер передает декодеру информацию, необходимую для восстановления исходного файла. Эта информация включает в себя атрибуты файла, такие как, например, его расширение, а также, для статических алгоритмов, модель кодирования. Наличие заголовка неизбежно увеличивает длину сжатого файла, поэтому наряду с чисто техническими проблемами, связанными с организацией чтения и записи заголовка, существует проблема минимизации его длины за счет выбора формата передачи модели кодирования. К сожалению, только этот фрагмент заголовка можно кратно уменьшить надлежащим выбором формата, поэтому данная тема заслуживает отдельного обсуждения.

Несмотря на то что алгоритмы сжатия без потерь инвариантны к типу сжимаемого файла, результирующая степень сжатия может сильно отличаться для разных файлов. Поскольку сжатие данных и есть главная цель применения этих алгоритмов, для ее достижения хороши все сред-

ства, в том числе и те, которые не имеют прямого отношения к обсуждаемой теме. В связи с этим обсуждается тема преобразования Бэрроуза — Уилера, гипотетически способствующего улучшению данного целевого показателя.

Алгоритмы сжатия должны выполнять свою миссию при минимально возможных вычислительных затратах. Учитывая важную роль, которую в реализации этих алгоритмов имеют методы поиска, целесообразно знакомство с алгоритмами хэш-поиска, способствующими повышению вычислительной эффективности этой вспомогательной процедуры.

Наконец, алгоритмы сжатия широко унифицированы в части выполнения ряда вспомогательных функций, таких, например, как построение частотной модели кодирования, ее запись в выходной поток, чтение модели из входного потока, а иногда и основных функций кодирования/декодирования. Поэтому имеет смысл обсуждение способов построения объектной модели не только для работы с потоками бинарных данных, но и для конкретного семейства алгоритмов сжатия в целом.

П.2.1. Работа с потоковыми данными в двоичном формате

Сложность обмена двоичными кодами с потоками данных заключается в том, что длина этих кодов не кратна байту — стандартной единице обмена. В результате возникает необходимость «упаковки» коротких кодов в один байт или «нарезки» длинного кода для размещения в последовательных байтах. В любом случае необходимо контролировать выраженное в битах смещение как внутри байта, предназначенного для обмена с файлом, так и внутри кода переменной длины.

Операция записи битовых данных в выходной поток реализована в классе Writer, а операция чтения битовых данных из входного потока — в классе Reader. С целью минимизации количества операций обмена с файлом при чтении используется буфер, размер которого по умолчанию определен равным 4 Кб, что соответствует размеру одной страницы памяти в Windows, а для управления операциями обмена используются указатель текущего байта и счетчик доступных байтов. Кроме того,

для удобства в оба класса инкапсулированы функции, реализующие обмен байтовыми данными с потоком. Описание членов классов приведено в табл. П.2.

Таблица П.2

Описание членов классов, обеспечивающих обмен битовыми данными с потоком

Наименование	Назначение
Класс Writer	
Данные	
`int bitpos`	позиция бита в байте при записи битовой информации
`unsigned char data`	байт, аккумулирующий биты при записи битовой информации
`FILE *filestream`	поток, связанный с файлом
`string filename`	имя файла
Методы	
`Writer(string filename);`	конструктор инициализирует объект, создает бинарный файл, filename — имя файла
`~Writer();`	деструктор закрывает поток
`bool isopen();`	функция возвращает true, если поток открыт, false — в противном случае

`int writebyte(unsigned char *buf, int count = 1);`
`int writebyte(long long *buf, int count = 8);`
`int writebyte(int *buf, int count = 4);`
	функция записывает в поток count байтов из buf, возвращает: при успешном завершении — количество записанных байтов; 0 при некорректных значениях параметров; –1, если поток не открыт (+2 перегрузки)

`int writebit(unsigned char bit);`	функция записывает 1 бит в поток, значение бита — младший бит, возвращает: 1 при успешном завершении; –1, если поток не открыт
`int flush();`	функция сбрасывает в поток незаписанные биты, возвращает: 0 при успешном завершении; –1, если поток не открыт
`int close();`	функция закрывает поток, если он открыт, возвращает: 0 при успешном завершении; –1, если поток не был открыт
`string getfilename();`	функция возвращает имя файла

Продолжение табл. П.2

Наименование	Назначение
Класс Reader	
Данные	
`vector <unsigned char> bufer`	массив, используемый в качестве буфера при чтении данных из файла
`unsigned int bufsize;`	размер буфера в байтах
`long bytecount`	количество доступных байтов в буфере
`long byteindex`	индекс доступного байта в буфере
`unsigned char data`	байт, используемый при чтении битовой информации
`int bitpos`	позиция бита в байте при чтении битовой информации
`FILE *filestream`	поток, связанный с файлом
`string filename`	имя файла
`long long filelength`	длина файла в байтах
Методы	
`Reader(string filename, unsigned int bufsize = 4096);`	конструктор инициализирует объект, открывает бинарный файл, filename — имя файла
`~Reader();`	деструктор освобождает память и закрывает поток
`bool isopen();`	функция возвращает true, если поток открыт, false — в противном случае
`void reset();`	устанавливает указатель текущей позиции в начало файла

`int readbyte(unsigned char *buf, int count = 1);`
`int readbyte(int *buf, int count = 4);`
`int readbyte(long long *buf, int count = 8);`
	функция читает из потока count байтов в buf, возвращает: при успешном завершении — количество прочитанных байтов; 0 при некорректных значениях параметров; –1, если поток не открыт (+2 перегрузки)

`unsigned char readbit();`	функция читает 1 бит из потока, возвращает: значение бита при успешном завершении; 128 в случае отсутствия данных (достигнут конец файла)
`int close();`	функция закрывает поток, если он открыт, возвращает: 0 при успешном завершении; –1, если поток не был открыт
`string getfilename();`	функция возвращает имя файла
`long long getfilelength();`	функция возвращает длину файла в байтах

Классы представлены библиотекой bitstream.lib. Заголовочный файл имеет вид:

```
#pragma once
#include <string.h>
#include <io.h>
#include <math.h>
#include <Windows.h>
#include <string>
#include <vector>

using namespace std;

class Writer
{
private:
  int bitpos;
  unsigned char data;
  FILE *filestream;
  string filename;
public:
  Writer(string filename);
  bool isopen();
  int writebyte(unsigned char *buf, int count = 1);
  int writebyte(long long *buf, int count = 8);
  int writebyte(int *buf, int count = 4);
  int writebit(unsigned char bit);
  int flush();
  void close();
  ~Writer();
  string getfilename();
};

class Reader
{
  unsigned int bufsize;
  unsigned char data;
  int bitpos;
  long byteindex, bytecount;
  vector <unsigned char> bufer;
  string filename;
  FILE* filestream;
  long long filelength;
public:
  Reader(string filename, unsigned int bufsize = 4096);
  ~Reader();
```

```
  void reset();
  int readbyte(unsigned char *buf, int count = 1);
  int readbyte(long long *buf, int count = 8);
  int readbyte(int *buf, int count = 4);
  unsigned char readbit();
  long long getfilelength();
  bool isopen();
  string getfilename();
  int close();
};
```

Ниже представлен файл исходного кода, реализующий компонентные функции классов Writer и Reader.

```
#include "pch.h"
#include "framework.h"
#include "bitstream.h"
#include <string.h>
#include <io.h>
#include <math.h>
#include <Windows.h>
#include <string>
#include <vector>

using namespace std;

Writer::Writer(string filename)
{
  bitpos = 0;
  data = 0;
  this->filename = filename;
  fopen_s(&filestream, filename.c_str(), "wb");
}

bool Writer::isopen()
{
  if filestream != nullptr;
}

Writer::~Writer()
{
   close();
}

int Writer::writebyte(unsigned char* buf, int count)
{
  if (filestream)
```

```
    if (!buf || count <= 0) // некорректные параметры
      return 0;
    else return fwrite(buf, 1, count, filestream);
  else return -1;
}

int Writer::writebyte(long long* buf, int count)
{
  if (filestream)
    if (!buf || count <= 0) // некорректные параметры
      return 0;
    else return fwrite(buf, 1, count, filestream);
  else return -1;
}

int Writer::writebyte(int* buf, int count)
{
  if (filestream)
    if (!buf || count <= 0) // некорректные параметры
      return 0;
    else return fwrite(buf, 1, count, filestream);
  else return -1;
}

int Writer::writebit(unsigned char bit)
{
  if (filestream)
  {
    data <<= 1;       // добавление бита в байт data
    data = data | (bit & 1);
    ++bitpos;
    if (bitpos == 8)  // байт укомплектован
    {
      fwrite(&data, 1, 1, filestream);
      bitpos = 0;
    }
    return 1;
  }
  else return -1;
}

int Writer::flush()
{
  if (filestream)
  {
    while (bitpos != 0) // доукомплектование байта нулевыми битами
      writebit(0);
```

```
    return 0;
  }
  else return -1;
}

string Writer::getfilename()
{
  return filename;
}

void Writer::close()
{
  if (filestream)
  {
    fclose(filestream);
    filestream = nullptr;
  }
}

Reader::Reader(string filename, unsigned int bufsize)
{
  fopen_s(&filestream, filename.c_str(), "rb");
  this->bufsize = bufsize;
  filelength = 0;
  this->filename = filename;
  if (filestream)
  {
    filelength = _filelength(_fileno(filestream));
    bufer.resize(bufsize);   // создание буфера
    bitpos = 8;
    data = 0;
    byteindex = 0;
    bytecount = 0;
  }
}

Reader::~Reader()
{
  close();
}

void Reader::reset()
{
  fseek(filestream, 0, 0);
}

int Reader::readbyte(unsigned char* buf, int count)
```

```
{
  if (filestream)
    if (!buf || count <= 0) // некорректные параметры
      return 0;
    else return fread(buf, 1, count, filestream);
  else return -1;
}

int Reader::readbyte(long long* buf, int count)
{
  if (filestream)
    if (!buf || count <= 0) // некорректные параметры
      return 0;
    else return fread(buf, 1, count, filestream);
  else return -1;
 }

int Reader::readbyte(int* buf, int count)
{
  if (filestream)
    if (!buf || count <= 0) // некорректные параметры
      return 0;
    else return fread(buf, 1, count, filestream);
  else return -1;
 }

unsigned char Reader::readbit()
{
  if (bytecount == 0 && bitpos == 8)  // в буфере нет доступных
  {                                   // байтов
    bytecount = fread(&bufer[0], 1, bufsize, filestream);
    if (!bytecount) return 2;        // достигнут конец файла
    byteindex = 0;
    data = bufer[byteindex];  // берем из буфера доступный байт
    bytecount--;
    bitpos = 0;
  }
  if (bitpos == 8)  // в байте нет доступных битов
  {
    byteindex++;
    bitpos = 0;
    data = bufer[byteindex]; // берем из буфера доступный байт
    bytecount--;
  }
  // выделяем старший бит
  unsigned char bit = (unsigned char)(data) >> 7;
  data <<= 1;  // сдвигаем оставшиеся биты влево
```

```
  ++bitpos;
  return bit;
}

long long Reader::getfilelength()
{
  return filelength;
}

bool Reader::isopen()
{
  return filestream != nullptr;
}

string Reader::getfilename()
{
  return filename;
}

void Reader::close()
{
  if (filestream)
  {
    fclose(filestream);
    filestream = nullptr;
  }
}
```

П.2.2. Сохранение/восстановление атрибутов сжимаемого файла

Кодер передает атрибуты исходного файла декодеру в заголовке сжатого файла, откуда они впоследствии и восстанавливаются декодером. Разумеется, кодер и декодер должны соблюдать при этом соглашения, касающиеся структуры заголовка и формата его фрагментов. Ниже приведены примеры функций записи (PutFileExtension) и чтения (GetFileExtension) расширения исходного файла, реализованные в составе алгоритма арифметического кодирования.

Функция PutFileExtension() создает выходной файл с именем исходного файла и расширением *.art, записывает туда длину и расширение исходного файла, возвращает true, если файл создан успешно, и false

в противном случае. Имя и расширение исходного файла функция извлекает из строки fnamein как части строки до и после точки разделителя, соответственно.

```
bool PutFileExtension()
{
  char* path = new char[255], *ext = new char[10];
  int ind = fnamein.find('.');
  fnamein.copy(path, ind + 1);
  path[ind + 1] = '\0';
  string pathout = string(path);
  fnamein.copy(ext, fnamein.size() - ind - 1, ind + 1);
  ext[fnamein.size() - ind - 1] = '\0';
  int len = fnamein.size() - ind - 1;
  fnameout = pathout + "art";
  writer = new Writer(fnameout);
  if (writer->isopen())
  {
    //запись длины расширения
    writer->writebyte(&len);
    //запись расширения
    writer->writebyte((unsigned char*)ext, strlen(ext));
    return true;
  }
  else return false;
}
```

Функция GetFileExtension() читает длину расширения и само расширение из заголовка сжатого файла, создает файл с именем сжатого файла и считанным расширением, возвращает true, если файл создан успешно, и false в противном случае.

```
bool GetFileExtension()
{
  int ex;
  // читаем длину расширения
  int k = reader->readbyte(&ex);
  char* path = new char[255], *ext = new char[ex + 1];
  int ind = fnamein.find('.');
  fnamein.copy(path, ind + 1);
  path[ind + 1] = '\0';
  // читаем расширение
```

```
  reader->readbyte((unsigned char*)ext, ex);
  ext[ex] = '\0';
  strcat(path, ext);
  // создаем выходной поток
  writer = new Writer(fnameout);
  if (writer->isopen()) return true;
  else return false;
}
```

П.2.3. Преобразование Бэрроуза — Уилера (BWT)

BWT применяется к фрагментам фиксированной длины (блокам) входного потока. Понятно, что чем больше размер блока, тем больше возможностей для увеличения длин серий открывает метод. С другой стороны, увеличение размера блока влечет за собой рост вычислительных затрат, необходимых для выполнения преобразования. Поэтому необходим компромисс. Рекомендованная длина блока — 1–2 килобайта.

Рассмотрим *алгоритм прямого преобразования*. К выделенному блоку длины n применяется процедура циклического сдвига, суть которой состоит в сдвиге блока на одну позицию вправо и переносе символа, находящегося на правой границе, в начало блока. К полученному результату вновь применяется процедура циклического сдвига. Этот процесс повторяется $n-1$ раз. Его результаты можно представить в виде матрицы циклического сдвига. Например, матрица циклического сдвига для блока «АБРАКАДАБРА» будет иметь следующий вид:

	0	1	2	3	4	5	6	7	8	9	10
0	А	Б	Р	А	К	А	Д	А	Б	Р	А
1	А	А	Б	Р	А	К	А	Д	А	Б	Р
2	Р	А	А	Б	Р	А	К	А	Д	А	Б
3	Б	Р	А	А	Б	Р	А	К	А	Д	А
4	А	Б	Р	А	А	Б	Р	А	К	А	Д
5	Д	А	Б	Р	А	А	Б	Р	А	К	А
6	А	Д	А	Б	Р	А	А	Б	Р	А	К
7	К	А	Д	А	Б	Р	А	А	Б	Р	А
8	А	К	А	Д	А	Б	Р	А	А	Б	Р
9	Р	А	К	А	Д	А	Б	Р	А	А	Б
10	Б	Р	А	К	А	Д	А	Б	Р	А	А

Следующий этап прямого преобразования — сортировка строк матрицы циклического сдвига в лексикографическом порядке. В результате применения этой процедуры для нашего примера получим:

	0	1	2	3	4	5	6	7	8	9	10
0	А	А	Б	Р	А	К	А	Д	А	Б	Р
1	А	Б	Р	А	А	Б	Р	А	К	А	Д
2	А	Б	Р	А	К	А	Д	А	Б	Р	А
3	А	Д	А	Б	Р	А	А	Б	Р	А	К
4	А	К	А	Д	А	Б	Р	А	А	Б	Р
5	Б	Р	А	А	Б	Р	А	К	А	Д	А
6	Б	Р	А	К	А	Д	А	Б	Р	А	А
7	Д	А	Б	Р	А	А	Б	Р	А	К	А
8	К	А	Д	А	Б	Р	А	А	Б	Р	А
9	Р	А	А	Б	Р	А	К	А	Д	А	Б
10	Р	А	К	А	Д	А	Б	Р	А	А	Б

Упорядоченная матрица содержит результат прямого преобразования в виде последнего столбца и индекса исходного блока (выделены заливкой). Обратите внимание: в последнем столбце появились серии, длина которых больше 1, в исходном блоке их не было! Может возникнуть вопрос — а почему бы тогда не использовать в качестве результата первый столбец упорядоченной матрицы, ведь там цепочки еще длиннее! Ответ такой: вследствие сортировки первый столбец утратил информацию о последовательности символов в блоке, а без нее обратное преобразование невозможно. Но почему же все-таки именно последний столбец используется для восстановления блока? Все дело в том, что именно из последнего столбца берутся символы для выполнения циклического сдвига!

Алгоритм обратного преобразования, наряду с результатами прямого преобразования, использует следующие факты.

1. Строки матрицы упорядочены в лексикографическом порядке.
2. Последний столбец содержит все символы блока.
3. Строки неупорядоченной матрицы образованы путем последовательного применения операции циклического сдвига.

Совместное использование первых двух фактов немедленно позволяет восстановить содержание первого столбца:

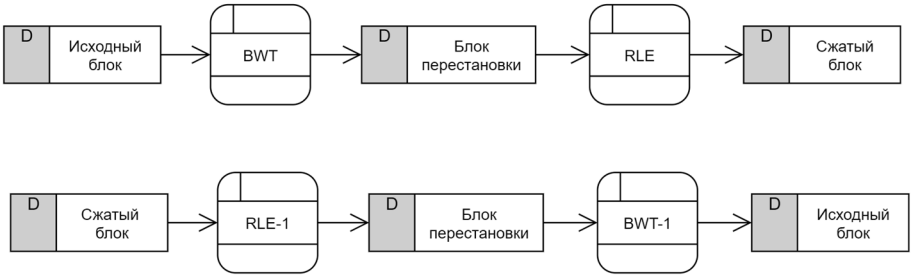
	0	1	2	3	4	5	6	7	8	9	10
0	А										Р
1	А										Д
2	А										А
3	А										К
4	А										Р
5	Б										А
6	Б										А
7	Д										А
8	К										А
9	Р										Б
10	Р										Б

Факт 3 позволяет установить, что в матрице присутствуют строки, начинающиеся на «РА», «ДА», «АА», «КА» и т. д., а факт 1 однозначно определяет, в каком порядке эти строки расположены. Результат — восстановление первых двух столбцов:

	0	1	2	3	4	5	6	7	8	9	10
0	А	А									Р
1	А	Б									Д
2	А	Б									А
3	А	Д									К
4	А	К									Р
5	Б	Р									А
6	Б	Р									А
7	Д	А									А
8	К	А									А
9	Р	А									Б
10	Р	А									Б

Действуя в том же духе, можно восстановить третий, четвертый и далее — все незаполненные столбцы, после чего останется только взять строку по индексу исходного блока.

Ниже показана схема совместного использования RLE и BWT (RLE^{-1} и BWT^{-1} использованы для обозначения декодера RLE и обратного BWT).



Программная реализация BWT

Применение преобразования Бэрроуза — Уилера требует дополнительных вычислительных затрат на кодирование и декодирование. Кроме того, явное представление матрицы циклического сдвига в оперативной памяти компьютера при рекомендованных размерах блока от 1 до 2 Кб может потребовать от 1 до 4 Мб. Хотя для современных компьютеров такие затраты памяти не являются критичными, идея сделать их пропорциональными длине блока, а не квадрату этой длины сохраняет свою привлекательность. Таким образом, речь идет о выборе варианта программной реализации BWT, который, с одной стороны, работает с компактным представлением матрицы циклического сдвига, а с другой — уменьшает или, по крайней мере, не увеличивает вычислительные затраты на выполнение прямого и обратного преобразования, по сравнению с алгоритмами, которые работают с полной матрицей. Попытки одновременной экономии сразу двух ресурсов — памяти и процессорного времени — редко бывают удачными. Чаще экономия одного ресурса покупается ценой перерасхода другого, например, алгоритмы хэш-поиска [4] «покупают» высокую вычислительную эффективность за счет увеличения адресного пространства поиска. Однако преобразование Бэрроуза — Уилера в этом смысле является счастливым исключением.

При **прямом преобразовании** проблему экономии памяти можно решить с помощью вспомогательного массива, элементы которого содержат смещения начала строк матрицы циклического сдвига относи-

тельно исходной строки *до* выполнения сортировки, а индексы показывают позиции этих строк в матрице *после* сортировки. До сортировки значения элементов совпадают со значениями индексов. Зная способ формирования строк матрицы циклического сдвига, по указателю начала строки легко восстановить и саму строку, по значению которой производится сортировка. В результате сортировки элемент *i* вспомогательного массива примет значение, определяющее позицию, которую должна занять в упорядоченной последовательности соответствующая строка. Вот как, например, будет выглядеть вспомогательный массив после сортировки блока «АБРАКАДАБРА».

Позиция после сортировки	0	1	2	3	4	5	6	7	8	9	10
Смещение	1	4	0	6	8	3	10	5	7	2	9

Отсюда, в частности, видно, что строка АБРАКАДАБРА со смещением 0 (исходная) в результате сортировки «переместилась» в позицию 2, а строка АБРААБРАКАД со смещением 4 «переместилась» в позицию 1, что соответствует результатам выполнения сортировки на полной матрице сдвига. Для дальнейшего обсуждения очень важно понимать два момента. Во-первых, матрица циклического сдвига в явном виде не существует. Каждая строка восстанавливается по значению ее смещения относительно исходной. Во-вторых, сортировка строк в этой виртуальной матрице сдвига выполняется без перемещения объектов сортировки. Перемещаются лишь указатели на начало этих строк. Сортировку, при которой перемещаются не сами данные, а указатели на них, называют логической, в отличие от физической, предполагающей перемещение данных [4].

Вычислительная эффективность сортировки существенно влияет на эффективность прямого преобразования и кодера RLE в целом. Это обстоятельство следует учитывать при выборе алгоритма сортировки. Можно рекомендовать алгоритмы быстрой сортировки, сортировки слиянием или пирамидальной сортировки [4], эффективность которых лучше квадратичной.

Приведенный ниже код прямого преобразования BWT написан в предположении, что соответствие между физическим адресом строки в матрице сдвига и логическим адресом этой же строки в матрице после сортировки зафиксировано в массиве index: номер ячейки массива соответствует логическому адресу, а значение соответствующего элемента — физическому. Кроме того, код использует следующие декларации.

```
// размер блока
int len;
// входной и выходной блок
unsigned char *str_in, *str_out;
// массив индексов, смещение исходного блока в BWT
unsigned short int *index, ptr_out;
```

Смещение исходной строки в матрице сдвига определяется как индекс элемента в массиве index, значение которого равно 0. Выходной блок формируется из последних элементов строк отсортированной матрицы сдвига. Вычислительные затраты на выполнение этой операции пропорциональны длине блока.

```
 for (int j = 0; j < len; j++)
 {
  //определение смещения исходной строки в отсортированной матрице
  if (index[j] == 0) ptr_out = j;
  //формирование преобразованного блока
  str_out[j] = *(str_in + len - 1 - index[j]);
 }
```

Описанный вариант программной реализации прямого преобразования укладывается в программу-минимум: экономит память и не ухудшает вычислительной эффективности по сравнению с базовым вариантом: в том и другом случае основная составляющая вычислительных затрат приходится на алгоритм сортировки.

Обратное преобразование начинается с получения первого столбца матрицы путем упорядочения по возрастанию последнего столбца — строки, полученной с помощью прямого преобразования (рис. П.1).

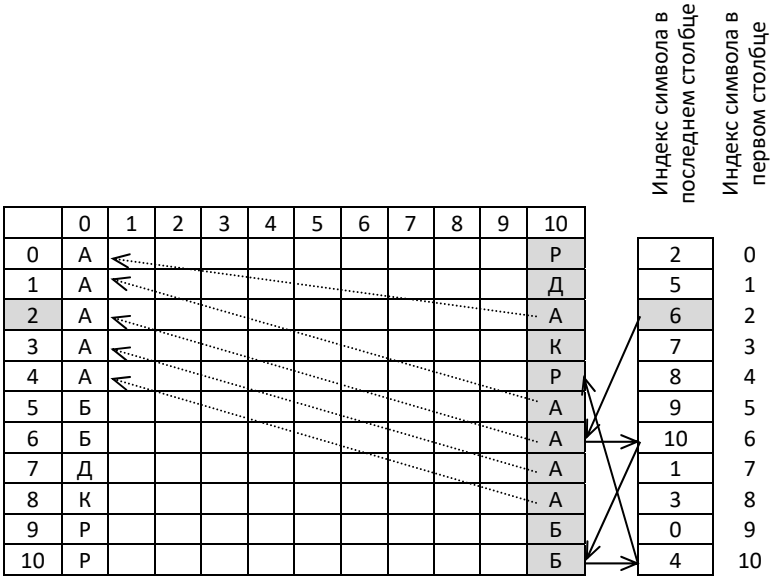


Рис. П.1. Эффективная реализация обратного преобразования

Поскольку прямое преобразование использует циклический сдвиг вправо, все двухсимвольные последовательности РА, ДА, КА, РА, АБ, АБ, АД, АК, БР и БР входят в восстанавливаемую строку, остается правильно соединить их между собой. Для этого достаточно знать, в какую позицию первого столбца в результате сортировки переместится каждый символ последнего столбца. Установить соответствие между физическим адресом символа в последнем столбце и его логическим адресом в первом столбце можно, применив индексную сортировку. Ее результаты показаны в двух правых колонках таблицы и для символов «А» продублированы пунктирными стрелками на рис. П.1. **Для последующего применения обратного BWT-преобразования критически важно сохранить порядок следования одинаковых символов до и после сортировки**. То есть, например, символ «А», который находится в последнем столбце на второй позиции, в первом столбце должен предшествовать символу «А» из шестой позиции последнего столбца. К сожалению, не все алгоритмы сортировки обеспечивают сохранение порядка. Так, например, сортировка слиянием сохраняет порядок одинаковых символов в отсортиро-

ванной последовательности, а пирамидальная сортировка «перемешивает» его. Для сортировки следует использовать только те алгоритмы, которые сохраняют порядок одинаковых символов в отсортированной последовательности! Восстановление начинается с элемента, соответствующего адресу смещения исходного блока в результате BW-преобразования (в нашем случае это элемент с индексом 2). Символ «А», с которого начинается вторая строка, соответствует шестой позиции последнего столбца. Первую позицию шестой строки занимает символ «Б», что позволяет восстановить начальную часть блока — «АБ». В свою очередь, символ «Б» соответствует десятой позиции последнего столбца. В первой позиции десятой строки расположен символ «Р», который является третьим символом блока, и т. д. Начальная часть процесса обратного преобразования показана на рис. П.1 сплошными стрелками.

Программная реализация алгоритма основывается на следующих декларациях.

```
// размер блока
int len;
// входной и выходной блок
unsigned char *str_in, *str_out;
// массив индексов, смещение исходного блока в BWT
unsigned short int *index, start;
```

Приведенный ниже код обратного преобразования BWT написан в предположении, что соответствие между физическим адресом символа в последнем столбце и его логическим адресом в первом столбце зафиксировано в массиве index. Код опровергает известную пословицу «Скоро сказка сказывается, да не скоро дело делается». Во-первых, описание алгоритма на языке C++ гораздо компактнее словесного (см. выше): движение по восстанавливаемому блоку str_out реализуется единственным оператором `j = index[j]` внутри цикла по длине этого блока len!

```
for (int j = start, i = 0; i < len; i++)
{
 str_out[i] = str_in[index[j]];
 j = index[j];
}
```

Во-вторых, рассмотренный алгоритм обратного преобразования экономит не только память, но и вычислительные затраты. Количество итераций восстановления пропорционально длине блока len, а не квадрату этой длины, как это было бы в случае использования матрицы.

П.2.4. Хеширование по строковому ключу

Хеш-поиск — это специфическая процедура, в основе которой лежит предположение о существовании функциональной связи между значением ключа и его адресом [4]. Функция, принимающая ключ и возвращающая адрес, называется *хеш-функцией*. Вычисление хеш-функции напоминает генерацию псевдослучайных чисел в том смысле, что вычисленный адрес в таблице цепочек является случайным. Существуют различные методы вычисления хеш-функций. Например, для целочисленных ключей известен метод деления, согласно которому адрес ключа *A* получается как модуль от деления значения ключа *K* на размер адресного пространства *N* (в нашем случае — размер таблицы цепочек):

$$A = K \bmod N.$$

Пример использования хеш-функции, реализующей метод деления для вычисления адреса в пространстве размером $N = 4$, приведен на рис. П.2.

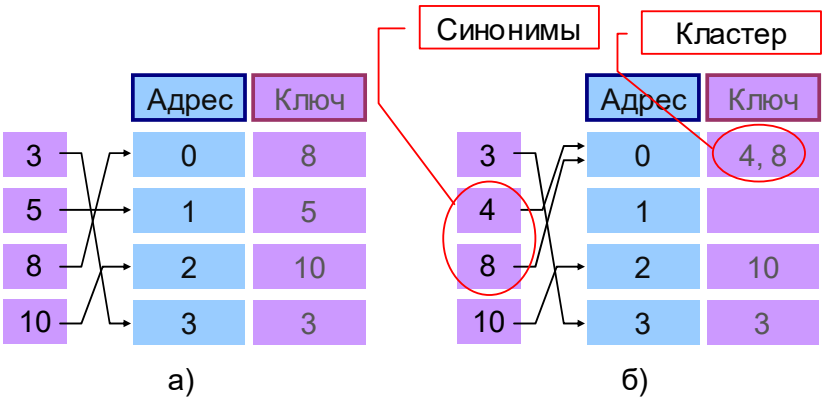


Рис. П.2. Хеш-функция, реализующая метод деления

Без учета *коллизий*, то есть ситуаций, когда на один и тот же адрес претендуют несколько ключей (рис. П.2*б*), эффективность хеш-поиска можно считать равной эффективности индексного доступа (рис. П.2*а*). Возникновение коллизии требует дополнительных вычислительных затрат на ее разрешение[21]. Поэтому вероятность коллизий необходимо уменьшать, в том числе — за счет увеличения размера таблицы цепочек. Следовательно, применяя методы хеш-поиска, мы сознательно идем на перерасход одного ресурса (памяти) с целью экономии другого (вычислительной мощности процессора). Однако эти издержки можно свести к минимуму за счет правильного выбора хеш-функции.

Кодер LZW работает со строковым ключом, поэтому прежде всего следует преобразовать этот ключ к формату целого числа, к которому впоследствии будет применен алгоритм хеширования. Один из популярных способов преобразования[22] заключается в разбиении строкового ключа на блоки размером 2–4 байта и последующем преобразовании содержимого этих блоков в целое число соответствующей длины. Для преобразования к содержимому каждого блока последовательно применяется функция «исключающее или» — XOR. Чтобы в дальнейшем исключить коллизии для строковых ключей, отличающихся только порядком символов (например, «АН» и «НА», табл. 2), к каждому промежуточному результату рекомендуется применять операцию циклического сдвига на один разряд влево. Ниже приведен код, который реализует предложенный алгоритм.

```
printf("Введите строку\n");
fflush(stdin);
gets_s(str);
unsigned short code = 0;
for (int i = 0; i < strlen(str); i += 2)
{
  unsigned short tmp = *(short*)(str + i);
  code = code ^ tmp;
  tmp = code >> 15;
```

[21] Разрешение коллизии предполагает последовательный просмотр ключей-конкурентов в рамках кластера или изолированного списка.

[22] https://studopedia.ru/7_150154_heshirovanie-strokovih-klyuchey.html.

```
    code = code << 1;
    code = code + tmp;
  }
  printf("%d\n", code);
```

Популярность хеширования постоянно растет, что требует унификации применяемых алгоритмов. Поэтому язык Python, например, поставляет программисту так называемые безопасные алгоритмы хеширования (Secure Hash Algorithms) в составе библиотеки hashlib. Функции этой библиотеки генерируют уникальный адрес для заданной входной строки, исключающий возможность коллизии. Этот адрес можно интерпретировать как шифрованную версию строки и использовать для хранения паролей на веб-сайтах или для организации связи транзакций в блокчейн-технологии [12].

П.3. Оконный интерфейс программы сжатия

Оконный интерфейс программы сжатия должен обеспечивать поддержку базового функционала программы и возможностей выполнения простейших исследований на его основе.

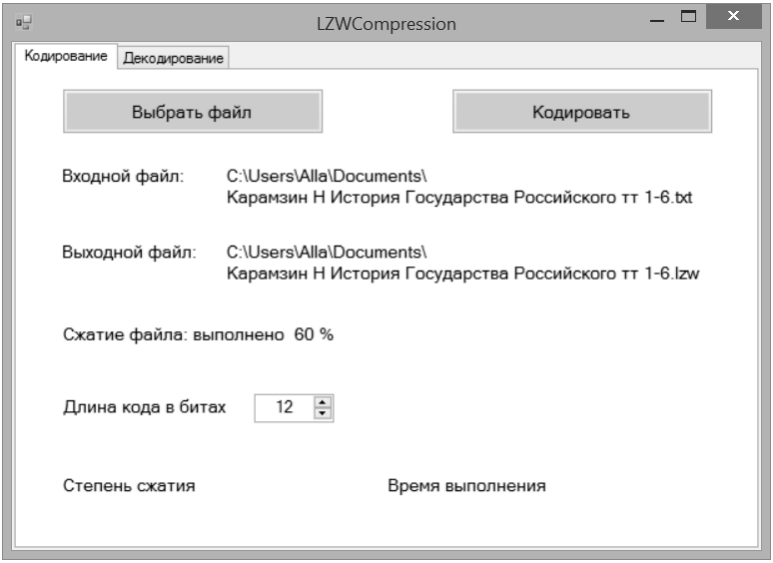
К базовым функциональным возможностям относятся:

1) выбор операции: кодирование или декодирование;
2) выбор файла для выполнения операции;
3) индикация прогресса выполнения операции.

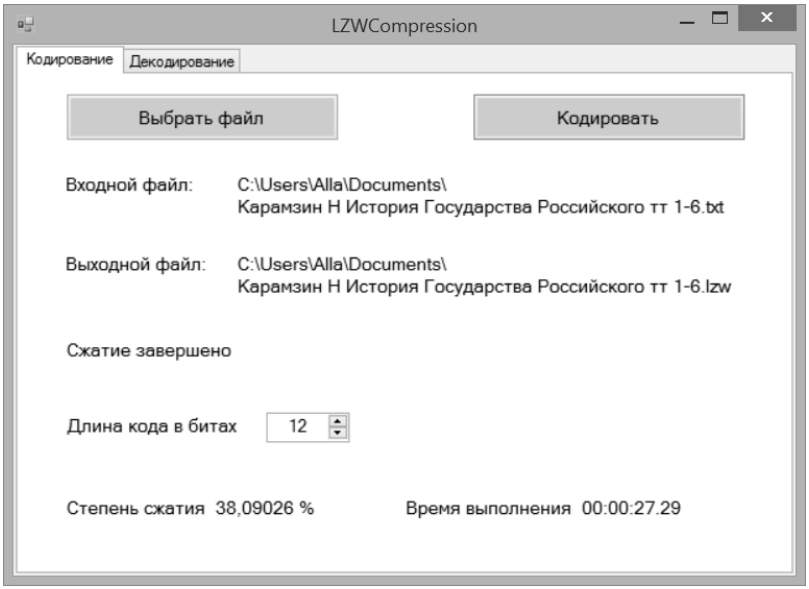
К исследовательским возможностям относятся:

1) настройка параметров алгоритма сжатия;
2) просмотр статистики выполнения операции (степень сжатия, время выполнения, максимальная и средняя длина цепочки для словарных алгоритмов, максимальная и средняя длина кода для частотных алгоритмов и т. п.).

Вариант организации интерфейса для алгоритма LZW приведен на рис. П.3. Выбор операции кодирования/декодирования осуществляется переключением на соответствующую закладку. Выбор файла для выполнения операции реализован на базе стандартных окон файлового диалога. При этом предполагается, что файл результата записывается в тот же каталог и под тем же именем и отличается от исходного только



а) интерфейс выполнения операции



б) итоговая статистика

Рис. П.3
Оконный интерфейс программы сжатия

расширением *.lzw, принятым для данного алгоритма. Индикация прогресса выполнения операции реализована в виде стандартного элемента управления Label, содержимое которого (процент от общего объема выполненной работы) динамически обновляется (рис. П.3*а*). Интерфейс настройки параметров на закладке «Кодирование» представлен скроллером для выбора длины lzw-кода и, соответственно, размера таблицы цепочек. Статистика операции включает время ее выполнения и достигнутую степень сжатия (рис. П.3*б*).

Интерфейс выполнен на языке C# с применением визуальных элементов управления из пакета средств разработки программного обеспечения (SDK) Microsoft .NET Framework. Программный код, реализующий все необходимые операции, связанные с кодированием и декодированием файла по алгоритму Лемпеля — Зива — Велча, представлен классом LZWCoder. Описание класса приведено в табл. П.3. Алгоритмы кодирования/декодирования и отдельные методы класса LZWCoder описаны в разделе 2.2 данного пособия.

Таблица П.3

Описание членов класса LZWCoder

Наименование	Назначение
Данные	
FILE* input	входной поток
FILE* output	выходной поток
char* input_name	имя входного файла
char output_name[255]	имя выходного файла
unsigned char len_code	длина кода

chaincode* table	таблица цепочек — массив структур:
	struct chaincode{
	unsigned char* chain; // цепочка байтов
	unsigned char len; // длина цепочки
	};

unsigned short maxcount	размер таблицы
unsigned short count	текущее количество записей в таблице

Продолжение табл. П.3

Наименование	Назначение
int filesize	размер несжатого файла
Методы	
`LZWCoder(unsigned char _len_code, char* _input_name);`	конструктор, инициализирует объект
`~LZWCoder();`	деструктор, освобождает динамически выделенную память
`void Init_table();`	функция создает и инициализирует постоянную часть таблицы цепочек
`int Openfile(Mode r);`	функция открывает входной файл и создает выходной файл, определяя его расширение в зависимости от параметра `Mode r`: `enum Mode { Coder, Decoder };` в режиме `Coder` – расширение ”.lzw”, в режиме `Decoder` расширение читается из сжатого файла
`bool Found_chain(unsigned char* chain, unsigned short& code, unsigned char len);`	функция ищет в таблице цепочку `chain` длины `len`, в случае успеха возвращает `true`, а также `code` — код цепочки
`void Write_code(unsigned short outcode, unsigned short& freebits, unsigned short& packcode);`	функция записывает код `outcode` в выходной поток, упаковывая его в 16-битный `packcode`, `freebits` – количество свободных битов в `packcode`
`void Add_chain(unsigned char* chain, unsigned short lench);`	функция добавляет в таблицу цепочку `chain` длины `len`
`int Encode(callback_t* callback);`	функция кодирует входной поток, `callback` — указатель на функцию обратного вызова для отображения хода процесса
`bool Read_code(unsigned short& code, unsigned short& word, unsigned short& bitsinword, unsigned short& bits, unsigned char& freebits);`	функция извлекает код `code` из входного потока, используя 16-битный буфер `word`

Продолжение табл. П.3

Наименование	Назначение
`bool Found_code(unsigned char* chain, unsigned short code, unsigned char* len);`	функция по коду `code` ищет в таблице цепочку `chain` длины `len`, в случае успеха возвращает `true`
`int Decode(callback_t* callback);`	функция декодирует входной файл, `callback` — указатель на функцию обратного вызова для отображения хода процесса
`void Get_output_filename(char* buf, int max_len);`	функция возвращает имя выходного файла параметром `buf`, `max_len` — максимальная длина имени

Для объединения в одном проекте нативного кода C++ и управляемого кода C# класс LZWCoder был помещен в динамически подключаемую библиотеку (dll). Поскольку C# не может непосредственно импортировать классы C++, библиотека была дополнена функциями-обертками для четырех методов, вызов которых осуществляется в коде на C#[23]. Управляемый код импортирует только эти обертки.

Функция LZWcoder используется в качестве обертки конструктора, создающего экземпляр класса LZWCoder.

```
extern "C" __declspec(dllexport) void* LZWcoder(unsigned char
_len_code, char* _input_name)
{
  LZWCoder *obj = new LZWCoder(_len_code, _input_name);
  return obj;
}
```

Функции encode и decode используются в качестве обертки методов кодирования и декодирования соответственно. В качестве параметров им передаются указатель на объект класса LZWcoder, инициализированного конструктором, и указатель на функцию обратного вызова на C#, которая отображает прогресс кодирования (декодирования).

```
extern "C" __declspec(dllexport) void encode(void* obj,
callback_t * callback)
```

[23] Во избежание конфликта имен названия функций-оберток отличаются от названий соответствующих методов.

```
{
  ((LZWCoder*)obj)->Encode(callback);
}

extern "C" __declspec(dllexport) void decode(void* obj,
callback_t * callback)
{
  ((LZWCoder*)obj)->Decode(callback);
}
```

Ниже приведен код функции, отображающей прогресс кодирования. В теле функции encodePercentShow вызывается метод Invoke стандартного элемента управления Label. Делегат, содержащий метод (в данном случае он определен как анонимная функция), выполняется в контексте потока элемента управления.

```
private void encodePercentShow(int percent)
{
  code_steps.Invoke((MethodInvoker)delegate
  {
    code_steps.Text = "Сжатие файла: выполнено  " +
                      percent.ToString() + " %";
    code_steps.Visible = true;
  });
}
```

Функции encode и decode выполняются как отдельные задачи в асинхронных обработчиках событий Click кнопок «Кодировать» и «Декодировать» соответственно. Асинхронные обработчики событий позволяют избежать блокирования потока пользовательского интерфейса. Ниже приведен фрагмент кода обработчика события Click кнопки «Кодировать»:

```
obj = LZWcoder(lencode, openFileDialog1.FileName);

Stopwatch stopWatch = new Stopwatch();
stopWatch.Start();

FuncPtr funcPtr = encodePercentShow;    // делегат
await Task.Run(() => encode(obj, funcPtr));

stopWatch.Stop();
```

Объект stopWatch в приведенном коде используется для определения точного времени, затраченного на выполнение операции кодирования.

Функция get_output_filename используется в качестве обертки метода, возвращающего имя выходного файла. При декодировании расширение восстанавливаемого файла извлекается из заголовка сжатого файла (в приведенной реализации алгоритма LZW имена исходного и сжатого файлов совпадают).

```
extern "C" __declspec(dllexport) void get_output_filename(void*
obj, char* buf, int max_len)
{
  ((LZWCoder*)obj)->Get_output_filename(buf, max_len);
}
```

Рассмотренный вариант построения пользовательского интерфейса не является единственным. Существуют и другие средства организации взаимодействия с пользователем в приложениях, реализованных на C++. Среди наиболее известных следует отметить библиотеку Microsoft Foundation Class (MFC) и кроссплатформенную библиотеку Qt, включающие кроме прочего и компоненты графического интерфейса.

П.4. Объектная реализация частотных алгоритмов сжатия

Частотные алгоритмы сжатия объединены общей концепцией и поэтому используют похожие структуры данных и операции их обработки. Учитывая это, в обсуждаемой далее объектной модели каждый алгоритм реализован в отдельном классе, но классы образуют единую иерархию (рис. П.4).

Рассмотрим каждый из этих классов.

1. Базовый класс BinaryTree является абстрактным шаблонным классом, реализующим методы, общие для бинарных деревьев. Дерево рассматривается в его традиционном представлении, когда каждый узел содержит в себе указатели на два дочерних поддерева, что определено в структуре Node. Информационное наполнение узла может изменяться в зависимости от ситуации, в которой используется шаблон, поэтому тип данных в узле является параметром шаблона T. Описание данных и методов класса см. в табл. П.4.

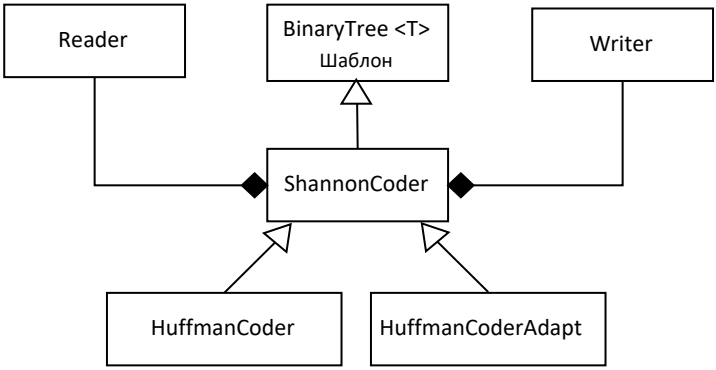


Рис. П.4. Диаграмма классов, реализующих алгоритмы Шеннона — Фано и Хаффмана

Таблица П.4

Описание членов класса BinaryTree

Наименование	Назначение
Данные	
`Node<T>* root`	указатель на корень дерева
Методы	
`BinaryTree()`	конструктор инициализирует корень дерева
`~BinaryTree()`	деструктор удаляет дерево при удалении объекта
`void DelTree(Node <T>* root)`	функция освобождает память, занимаемую узлами дерева
`Node<T>* CreateNode(T data)`	функция создает новый узел и инициализирует его, возвращает указатель на узел
`T* GetRoot()`	функция возвращает указатель на корень дерева
`virtual void CreateTree()`	абстрактная функция создания дерева

Поскольку шаблонный класс не может быть обработан компилятором для генерации объектного кода, вся реализация класса включена в заголовочный файл binarytree.h:

```
template <class T>
struct Node {
  T data;
  Node *left, *right;
```

```
};

template <class T>
class BinaryTree
{
protected:
  Node<T>* root;

public:
  BinaryTree()
  {
    root = NULL;
  }

  void DelTree(Node <T>* root)
  {
    if (root != NULL)
    {
      DelTree(root->left);
      DelTree(root->right);
      delete root;
    }
  }

  ~BinaryTree()
  {
    DelTree(root);
  }

  Node<T>* CreateNode(T data)
  {
    Node<T>* ptr;
    ptr = new Node<T>;
    ptr->data = data;
    ptr->left = NULL;
    ptr->right = NULL;
    return ptr;
  }

  T* GetRoot()
  {
    return root;
  }

  virtual void CreateTree() = 0;

};
```

2. Класс ShannonCoder реализует алгоритм Шеннона — Фано, наследует шаблон BinaryTree для реализации некоторых поведений, связанных с построением бинарного дерева. Кроме того, класс использует объекты классов Reader и Writer, для доступа к которым подключается библиотека bitstream. Структура PrefixNode используется и как информационная составляющая в дереве кодирования (передается в шаблон в качестве параметра), и как элемент таблицы частот. Структура PrefixCode определяет элемент таблицы кодов. Особенности реализации алгоритма обсуждались в разделе 3.1.1. Описание компонентов класса приведено в табл. П.5.

Таблица П.5

Описание членов класса ShannonCoder

Наименование	Назначение
Данные	
`PrefixNode *list`	указатель на динамический массив «таблица частот»
`PrefixCode *codes`	указатель на динамический массив «таблица кодов»
`unsigned short count`	количество кодируемых байт-кодов
`unsigned char sizew`	количество байтов, отправляемых в выходной поток при сохранении частоты
`Writer *writer`	объект для записи данных в выходной поток
`Reader *reader`	объект для чтения данных из входного потока
`string fnamein`	имя входного файла
`string fnameout`	имя выходного файла (определяется таким же, как и имя входного файла за исключением расширения)
`string extout`	расширение имени несжатого файла
Методы	
`ShannonCoder(string namein)`	конструктор инициализирует объект, задает расширение сжатого файла
`~ShannonCoder()`	деструктор освобождает память при удалении объекта

Продолжение табл. П.5

Наименование	Назначение
`bool PutFileExtension()`	функция создает выходной поток и записывает в него расширение несжатого файла, возвращает: true в случае успешного завершения, false — если выходной поток не создан
`bool Statistic()`	функция генерирует таблицу частот и формирует заголовок сжатого файла: расширение и длина исходного файла, таблица частот; возвращает: true в случае успешного завершения, false — если выходной поток не создан
`virtual void PutStatistic()`	функция записывает таблицу частот в выходной поток
`void Sort()`	функция сортирует таблицу частот по убыванию частоты методом пузырька
`void AddNode(unsigned long long sum, unsigned char left, unsigned char right, Node <PrefixNode> **root)`	функция реализует алгоритм построения префиксного дерева Шеннона — Фано: sum — сумма весов некоторой зоны таблицы частот; left и right — левая и правая границы зоны, **root — адрес указателя на узел дерева
`virtual void CreateTree()`	функция инициирует создание дерева Шеннона — Фано
`void Create-Code(Node<PrefixNode> *t, unsigned long long code, unsigned short depth)`	функция генерирует таблицу кодов: *t — указатель на узел дерева, code — код в виде набора битов, depth — количество битов в коде
`virtual int Encode()`	функция сжимает входной файл, возвращает: 0 — при успешном завершении; 1 — не открыт входной поток; 2 — не создан выходной поток; 3 — не создано дерево кодирования
`bool GetFileExtension()`	функция считывает из сжатого файла расширение, генерирует имя выходного файла и создает выходной поток, возвращает: true в случае успешного завершения, false — если выходной поток не создан

Продолжение табл. П.5

Наименование	Назначение
`virtual void GetStatistic()`	функция считывает из входного потока таблицу частот
`void FindSym(Node<PrefixNode> *t, unsigned char *sym)`	функция обеспечивает считывание битов из входного потока и поиск байт-кода в дереве кодирования: *t — указатель на узел дерева, sym — байт-код
`virtual int Decode()`	функция восстанавливает сжатый файл, возвращает: 0 — при успешном завершении; 1 — не открыт входной поток; 2 — не создан выходной поток; 3 — не создано дерево кодирования

Программный код сохранен в файле ”shannon.cpp”, который подключается в файлах, содержащих определение классов, реализующих алгоритмы Хаффмана.

```
#include <string.h>
#include <io.h>
#include <Windows.h>
#include <string>
#include <vector>
#include <bitstream.h>
#include "binarytree.h"

using namespace std;
#pragma comment( lib, "bitstream" )

struct PrefixNode   // структура данных в узле бинарного дерева
{                   // передается в шаблон как параметр
  unsigned long long w;
  unsigned char sym;
};

struct PrefixCode
{
  unsigned char len;  // длина кода
  unsigned long long code; // собственно код (ограничение 64 бита)
};

class ShannonCoder : public BinaryTree<PrefixNode>
{
```

```
protected:
  PrefixNode *list;     // таблица частот
  PrefixCode *codes;    // таблица кодов
  unsigned short count;// количество кодируемых элементов
  unsigned char sizew; // размер частоты повторений байт-кода
  Writer *writer;
  Reader *reader;
  string fnamein, fnameout, extout;

public:
  ShannonCoder(string namein) : BinaryTree()
  {
    fnamein = namein;
    extout = "shf";
    reader = NULL;
    writer = NULL;
    codes = NULL;
    list = NULL;
    count = 0;
  }

  ~ShannonCoder()
  {
    if (reader)
      delete reader;
    if (writer)
      delete writer;
    delete list;
    delete codes;
    fnamein.erase();
    fnameout.erase();
  }

  bool PutFileExtension()
  {
    char* path = new char[255], *ext = new char[10];
    int ind = fnamein.find('.');
   //выделение имени исходного файла
    fnamein.copy(path, ind + 1);
    path[ind + 1] = '\0';
    string pathout = string(path);
    // выделение расширения исходного файла
    fnamein.copy(ext, fnamein.size() - ind - 1, ind + 1);
    ext[fnamein.size() - ind - 1] = '\0';
    int len = fnamein.size() - ind - 1;
    // сжатый файл имеет то же имя, что и исходный файл,
```

```
  // но расширение "shf"
  fnameout = pathout + extout;
  writer = new Writer(fnameout);
  if (writer->isopen())
  {
    writer->writebyte(&len); //запись длины расширения
    //запись расширения
    writer->writebyte((unsigned char*)ext, strlen(ext));
    return true;
  }
  else return false;
}

// определение частоты повторений
bool Statistic()
{
  unsigned char c;
  // длина файла
  unsigned long long flen = reader->getfilelength();
  // подсчет количества повторений (частоты) байт-кодов
  for (long long i = 0; i < flen; i++)
  {
    reader->readbyte(&c);
    list[c].w++;
  }
  for (int i = 0; i < 256; i++)    // заполнение байт-кодов
    list[i].sym = (unsigned char)i;
  if (PutFileExtension())
  {
    writer->writebyte((long long*)&flen);
    PutStatistic();   // запись таблицы частот в выходной поток
    reader->reset();
    return true;
  }
  else return false;
}

// запись таблицы частот в выходной поток
virtual void PutStatistic()
{
  writer->writebyte(&sizew);
  for (int i = 0; i < 256; i++)
    writer->writebyte((int*)&list[i].w, sizew);
}

// сортировка таблицы частот по убыванию частоты
```

```
void Sort()
{
  unsigned short i, j;
  unsigned char ch;
  bool sort = false;
  PrefixNode t;
  // сортировка методом "пузырька"
  for (i = 0; i < 255 && !sort; i++)
  {
    sort = true;
    for (j = 255; j > i; j--)
    if (list[j - 1].w < list[j].w)
    {
      t = list[j];
      list[j] = list[j - 1];
      list[j - 1] = t;
      sort = false;
    }
  }
  // определение количества значимых байт-кодов
  for (count = 255; count >= 0 && !list[count].w; count--);
  count++;
}

// рекурсивная функция построения дерева Шеннона-Фано
void AddNode(unsigned long long sum, unsigned char left,
             unsigned char right, Node <PrefixNode> **root)
{
  if (left < right)
  {
    long long halfsum = sum / 2, currsum = 0;
    unsigned short i, middle;
    *root = CreateNode({ sum, 0 });
    for (i = left; currsum < halfsum; i++)
      currsum += list[i].w;
    if (halfsum - (currsum - list[i - 1].w) < currsum - halfsum)
    {
      middle = i - 2;
      currsum -= list[i - 1].w;
    }
    else middle = i - 1;
    if (middle == right) middle--;
    AddNode(currsum, left, middle, &((*root)->left));
    AddNode(sum - currsum, middle+1, right, &((*root)->right));
  }
  else
```

```
    *root = CreateNode({ sum, list[left].sym });
  }

  // функция, инициирующая создание дерева Шеннона-Фано
  virtual void CreateTree()
  {
    Sort();
    long long sum = 0;
    for (int i = 0; i < count; i++)
      sum += list[i].w;
    if (count > 0)
    {
      unsigned char left = 0, right = count - 1;
      AddNode(sum, left, right, &root);
    }
  }

  // генерация кодов
  void CreateCode(Node<PrefixNode> *t, unsigned long long code,
                  unsigned short depth)
  {
    if (t != NULL)
    {
      if (!t->left && !t->right)
      {
        codes[t->data.sym].len = depth;
        codes[t->data.sym].code = code;
      }
      CreateCode(t->left, code, depth + 1);
      code += 1 << depth;
      CreateCode(t->right, code, depth + 1);
    }
  }

  // сжатие файла
  virtual int Encode()
  {
    reader = new Reader(fnamein);
    if (reader->isopen())
    {
      list = new PrefixNode[256];    // таблица частот
      codes = new PrefixCode[256];   // таблица кодов
      memset(codes, 0, sizeof(PrefixCode) * 256);
      memset(list, 0, sizeof(PrefixNode) * 256);
      sizew = 4;
      if (Statistic())  // сбор статистики
```

```
    {
      CreateTree();    // построение дерева
      if (root)
      {
        CreateCode(root, 0, 0);   // построение таблицы кодов
        unsigned long long flen = reader->getfilelength();
        unsigned char c;
        // кодирование
        for (long long i = 0; i < flen; i++)
        {
          reader->readbyte(&c);
          for (int k = 0; k < codes[c].len; k++)
          {
            // запись кода в файл, начиная с младшего бита
            unsigned char bit = (codes[c].code << (63 - k)) >> 63;
            writer->writebit(bit);
          }
        }
        writer->flush();
        writer->close();
        return 0;
      }
      return 3;
    }
    return 2;
  }
  return 1;
}

// считывание из архива расширения исходного файла
bool GetFileExtension()
{
  int ex;
  int k = reader->readbyte(&ex);
  char* path = new char[255], *ext = new char[ex + 1];
  int ind = fnamein.find('.');
  fnamein.copy(path, ind + 1);
  path[ind + 1] = '\0';
  reader->readbyte((unsigned char*)ext, ex);
  ext[ex] = '\0';
  strcat(path, ext);
  writer = new Writer(fnameout);
  if (writer->isopen()) return true;
  else return false;
}
```

```
// считывание из архива таблицы частот
virtual void GetStatistic()
{
  reader->readbyte(&sizew);
  for (int i = 0; i < 256; i++)
  {
    reader->readbyte((int*)&list[i].w, sizew);
    list[i].sym = (unsigned char)i;
  }
}

// поиск байт-кода в дереве кодирования
void FindSym(Node<PrefixNode> *t, unsigned char *sym)
{
  if (!t->left && !t->right)
  {
    *sym = t->data.sym;
  }
  else
  {
    unsigned char c = reader->readbit();
    if (!c)
      FindSym(t->left, sym);
    else FindSym(t->right, sym);
  }
}

// декодирование файла
virtual int Decode()
{
  reader = new Reader(fnamein);
  if (reader->isopen())
  {
    if (GetFileExtension())
    {
      list = new PrefixNode[256];    // таблица частот
      codes = new PrefixCode[256];   // таблица кодов
      memset(codes, 0, sizeof(PrefixCode) * 256);
      memset(list, 0, sizeof(PrefixNode) * 256);
      unsigned long long flen, count = 0;
      reader->readbyte((long long*)&flen);
      GetStatistic();
      CreateTree();
      if (root)
      {
        unsigned char c;
```

```
        while (count < flen)
        {
          FindSym(root, &c);
          writer->writebyte(&c);
          count++;
        }
        writer->flush();
        writer->close();
        return 0;
       }
       else return 3;
     }
     else return 2;
   }
   else return 1;
  }
};
```

3. Класс HuffmanCoder реализует алгоритм Хаффмана построения префиксного дерева кодирования, является производным от класса ShannonCoder и наследует от него значительную часть своего поведения. Класс переопределяет только одну функцию родительского класса — функцию построения дерева кодирования CreateTree, при этом таблица частот трансформируется в список узлов NodeList nodelist[256], каждый элемент которого представлен структурой NodeList и кроме байт-кода и частоты его повторения (структура PrefixNode) содержит указатель на соответствующий узел дерева. Ниже приведен программный код класса HuffmanCoder.

```
#include "shannon.cpp"
struct NodeList
{
  PrefixNode data;
  Node <PrefixNode> *ptr;// указатель на узел дерева
};

class HuffmanCoder : public ShannonCoder
{
protected:
  NodeList nodelist[256];     // список узлов дерева Хаффмана

public:
  HuffmanCoder(string namein) : ShannonCoder(namein)
```

```
  {
    memset(nodelist, 0, sizeof(nodelist));
    extout = "hfm";
  }
  virtual void CreateTree()
  {
    Sort();
    for (int i = 0; i < count; i++)
      nodelist[i].data = list[i];
    int nodecount = count - 1;
    while (nodecount)
    {
      if (!nodelist[nodecount - 1].ptr)
        // узел (nodecount - 1) в дерево не добавлен
        nodelist[nodecount - 1].ptr = CreateNode(
                                { nodelist[nodecount - 1].data.w,
                                nodelist[nodecount - 1].data.sym });
      if (!nodelist[nodecount].ptr)
      // узел (nodecount) в дерево не добавлен
        nodelist[nodecount].ptr = CreateNode(
                                { nodelist[nodecount].data.w, nodel-
                               ist[nodecount].data.sym });
      // создание родительского узла
      Node <PrefixNode> *tmp = CreateNode(
                                { nodelist[nodecount - 1].data.w +
                               nodelist[nodecount].data.w, 0 });
      tmp->left = nodelist[nodecount - 1].ptr;
      tmp->right = nodelist[nodecount].ptr;
      int i;
      // вставка родительского узла в список узлов
      // с сохранением упорядоченности
      for (i = nodecount - 2; i >= 0 &&
                  nodelist[i].data.w < tmp->data.w; i--)
        nodelist[i + 1] = nodelist[i];
      nodelist[i + 1].data.w = tmp->data.w;
      nodelist[i + 1].data.sym = 0;
      nodelist[i + 1].ptr = tmp;
      nodecount--;
    }
    root = nodelist[0].ptr;
  }
};
```

Как было справедливо замечено (см. раздел 3.1.2.1), реализации классов ShannonCoder и HuffmanCoder страдают одним общим недостат-

ком — ограничением длины кода 64 битами. Самый простой способ снять ограничение — заменить в структуре PrefixCode компонент unsigned long long code на байтовый массив в 255 элементов (максимально возможная длина кода):

```
struct HuffmanCode
{
  unsigned char len;       // длина кода
  unsigned char code[255]; // собственно код
};
```

В памяти такая структура занимает 256 байт, а таблица кодов для 256 элементов — 64 Кб, что вполне допустимо для современных систем программирования. Запись кода в файл при этом не потребует битовых операций, а функция генерации кода примет вид:

```
  void CreateCode(Node<HuffmanNode> *t, unsigned char code[255],
                  unsigned short depth)
  {
    if (t != NULL)
    {
      if (!t->left && !t->right)
      {
        codes[t->data.sym].len = depth;
        memcpy(codes[t->data.sym].code, code, 255);
      }
      CreateCode(t->left, code, depth + 1);
      if (t->right)
      {
        code[depth] = 1;
        CreateCode(t->right, code, depth + 1);
        code[depth] = 0;
      }
    }
  }
```

Желающим упаковать код максимальной длины в 32 байта предлагаем сделать это самостоятельно.

При обсуждении статического алгоритма Хаффмана (см. раздел 3.1.2.1) было предложено сохранять в сжатом файле не таблицу частот байт-кодов, а вектор длин префиксных кодов. Такая оптимизация потребует предварительного построения вектора длин. Решение этой

задачи во многом повторяет алгоритм построения дерева Хаффмана, описанный в разделе 3.1.2, с небольшим дополнением: при каждом добавлении узла выполняется пересчет длин кодов для листьев, находящихся в поддеревьях этого узла. Дерево представлено массивом узлов NodeList nodelist[511], каждый элемент которого является структурой (инициализирован массив нулевыми значениями):

```
struct NodeList
{
  unsigned long long w;  // вес узла
  unsigned char sym;     // байт-код
  unsigned char len;     // длина кода
  unsigned char child;   // индекс правого потомка (для листа – 0)
};
```

Реализуется задача двумя функциями. Функция Calculation_length эмулирует процесс построения дерева Хаффмана с одновременным подсчетом длин префиксных кодов. Очевидно, что добавление каждого родительского узла увеличивает глубину дерева на 1, вместе с этим увеличивается и длина префиксного кода для каждого байт-кода, помещенного в дерево. Рекурсивная функция inclen обеспечивает полный обход построенной части дерева, и инкрементирует длину кода для каждого узла.

```
  void Calculation_length()
  {
    Sort();   // сортировка таблицы частот
    for (int i = 0; i < count; i++) // инициализация списка узлов
    {
      nodelist[i].sym = list[i].sym; //байт-код
      nodelist[i].w = list[i].w;     //частота
    }
    int countnode = count;     // количество узлов в дереве
    int currnode = count - 1; // текущий узел
    int start = currnode;      // индекс узла, который на текущий
                               // момент является корнем дерева
    while (currnode)  //пока не поднялись до нулевого элемента
    {
      // сдвигаем в массиве все узлы, начиная с nodecount - 1,
      // на 1 элемент вправо
      memcpy(nodelist + currnode, nodelist + currnode - 1,
             (countnode - currnode + 1) * sizeof(NodeList));
```

```
    countnode ++; // количество узлов в дереве
    // вставляем внутренний узел с суммарным весом
    nodelist[currnode - 1].w = nodelist[currnode].w +
                               nodelist[currnode + 1].w;
    // определяем его правый дочерний узел
    nodelist[currnode - 1].child = currnode;
    // увеличиваем номера дочерних узлов
    // у всех потомков узла nodecount
    for (int i = start; i < countnode; i++)
      if (nodelist[i].child > currnode) nodelist[i].child++;
    inclen(currnode);   // пересчитываем длину кода
    int i;
    // вставка родительского узла в список узлов
    // с сохранением упорядоченности
    NodeList tmp = nodelist[currnode - 1];
    for (i = currnode - 2; i >= 0 && nodelist[i].w < tmp.w; i--)
      nodelist[i + 1] = nodelist[i];
    nodelist[i + 1] = tmp;
    start = i + 1;
    currnode --;
  }
  // записываем длины в таблицу кодов по возрастанию
  // перезаписываем байт-коды в таблице частот в соответствии с
  // длиной (используется при построении дерева),
  // таблица частот в дальнейшем не используется
  for (int i = 0, j = 0; i < count; i++)
    if (!nodelist[i].child)
    {
      codes[j].len = nodelist[i].len;
      list[j++].sym = nodelist[i].sym;
    }
}

void inclen(int currnode)
{
  nodelist[currnode].len++; //увеличение длины кода
  nodelist[currnode +1].len++;
  if (nodelist[currnode].child)// переход к правому поддереву
    inclen(nodelist[currnode].child);
  if (nodelist[currnode + 1].child)// переход к левому поддереву
    inclen(nodelist[currnode + 1].child);
}
```

Поскольку декодеру теперь передается только вектор длин, то и дерево необходимо строить по этому вектору, как в кодере, так и в декодере. Принцип построения дерева рассмотрен в разделе 3.1.2.1. Ниже

приведена функция построения дерева. Функция создает корень дерева кодирования, а затем добавляет в него пути с помощью функции insert, начиная с пути максимальной длины.

```
virtual void CreateTree()
{
  int i;
  root = CreateNode({ 0, 0 });     // создать корень дерева
  for (i = count - 1; i >= 0; i--) // построить дерево кодов
    insert(&root, codes[i].len, list[i].sym);
}
```

Функция insert добавляет путь длины len в дерево кодирования. Поскольку вес узла в данной реализации не актуален, компонент *w* используется как ключ для определения направления построения пути: при нулевом значении путь строится влево, при значении –1 — вправо, –2 означает, что построение пути из данного узла невозможно. Изменяется значение *w* за счет добавления возвращаемого функцией значения: функция возвращает –1 при добавлении в дерево листа и при обнаружении узла, для которого построены и левое и правое поддеревья.

```
  short insert(Node<PrefixNode>** t, short len, char sym)
  {
    if (len > 0) // путь еще не построен
    {
      if ((*t)->data.w == 0) // от узла можно построить путь
      {
        if ((*t)->left == NULL)   // если нет левого поддерева
          (*t)->left = CreateNode({ 0, 0 });  // создаем узел
        // переходим к левому поддереву
        (*t)->data.w += insert(&(*t)->left, len - 1, sym);
        // если узел занят (есть левое и правое поддеревья)
        if ((*t)->data.w == -2)
          return -1;
        else return 0;
      }
      else
        if ((*t)->data.w == -1)
        {
          if ((*t)->right == NULL)
            (*t)->right = CreateNode({ 0, 0 });
          (*t)->data.w += insert(&(*t)->right, len - 1, sym);
          if ((*t)->data.w == -2)
```

```
        return -1;
      else return 0;
    }
  }
  else // путь построен
  {
    (*t)->data.sym = sym;// сохраняем байт-код
    return -1;
  }
}
```

4. Класс HuffmanCoderAdapt реализует адаптивный алгоритм Хаффмана (детальное описание алгоритма см. в разделе 3.2). Класс определен как производный от класса ShannonCoder. В силу особенностей самого алгоритма дерево представляется массивом узлов, каждый элемент которого является структурой типа HuffmanNode, состав структуры можно увидеть ниже в программном коде. Описание членов класса приведено в табл. П.6 (унаследованные члены, не используемые в реализации алгоритма, исключены).

Таблица П.6

Описание членов класса HuffmanCoderAdapt

Наименование	Назначение
Данные	
`short symbol_nodes[258]`	массив индексов листьев в массиве узлов
`HuffmanNode nodes[515]`	массив узлов
`unsigned short free_node`	массив индексов свободного узла в массиве узлов
`Writer *writer`	объект для записи данных в выходной поток
`Reader *reader`	объект для чтения данных из входного потока
`string fnamein`	имя входного файла
`string fnameout`	имя выходного файла (определяется таким же, как и имя входного файла за исключением расширения)
`string extout`	расширение имени несжатого файла
Методы	
`HuffmanCoderAdapt(string namein)`	конструктор инициализирует объект, задает расширение сжатого файла
`void init_tree()`	функция инициализирует дерево

Продолжение табл. П.6

Наименование	Назначение
`bool PutFileExtension()`	функция создает выходной поток и записывает в него расширение несжатого файла, возвращает: true в случае успешного завершения, false — если выходной поток не создан (унаследовано от ShannonCoder)
`void AddList(unsigned char c, bool encode)`	функция добавляет лист в дерево кодирования: c — значение байт-кода, encode — режим использования функции (кодирование — true, декодирование — false)
`void UpdateTree(short curr_node)`	функция обновляет дерево, восстанавливая свойство упорядоченности: curr_node — индекс текущего узла
`virtual int Encode()`	функция сжимает входной файл, возвращает: 0 — при успешном завершении; 1 — не открыт входной поток; 2 — не создан выходной поток
`bool GetFileExtension()`	функция считывает из сжатого файла расширение, генерирует имя выходного файла и создает выходной поток, возвращает: true в случае успешного завершения, false — если выходной поток не создан (унаследовано от ShannonCoder)
`virtual int Decode()`	функция восстанавливает сжатый файл, возвращает: 0 — при успешном завершении; 1 — не открыт входной поток; 2 — не создан выходной поток; 3 — восстановление не было успешным

Программный код класса HuffmanCoderAdapt имеет вид:

```
#include "shannon.cpp"
struct HuffmanNode
{
  unsigned long long w;// вес узла
  short sym;   // байт-код
  short parent;// индекс родительского узла
  short child; // индекс правого потомка
               // (если узел является листом, child = -1)
};

class HuffmanCoderAdapt : public ShannonCoder
{
  const short esc = 257;    // код esc
```

```
  const short eof = 256;    // код eof
protected:
  short symbol_nodes[258];    // массив листьев
  HuffmanNode nodes[515];     // массив узлов
  unsigned short free_node;   // индекс свободного узла

public:
  HuffmanCoderAdapt(string namein) : ShannonCoder(namein)
  {
    fnamein = namein;
    extout = "hfm";
    memset(nodes, 0, sizeof(nodes));
    init_tree();
  }

  void init_tree()// инициализация дерева
  {
    for (int i = 0; i < 256; i++)
      symbol_nodes[i] = -1;
    nodes[0] = { 0, -1, -1, 1 };
    nodes[1] = { 0, eof, 0, -1 };
    nodes[2] = { 0, esc, 0, -1 };
    symbol_nodes[eof] = 1;
    symbol_nodes[esc] = 2;
    free_node = 3;
  }

  void UpdateTree(short curr_node) // обновление дерева
  {
    while (curr_node >= 0)   // пока не поднимемся по дереву
    {                        // до корневой вершины
      nodes[curr_node].w++;  // увеличиваем вес текущего узла
      int curr = curr_node - 1;
      // ищем место для перемещения узла
      while (curr>0 && nodes[curr_node].w > nodes[curr].w) curr--;
      curr++;
      if (curr != curr_node && curr != nodes[curr_node].parent)
      {
        // перемещаем узел, переопределяем связи
        if (nodes[curr].child == -1)
          symbol_nodes[nodes[curr].sym] = curr_node;
        else
        {
          nodes[nodes[curr].child].parent = curr_node;
          nodes[nodes[curr].child + 1].parent = curr_node;
        }
```

```
      if (nodes[curr_node].child == -1)   // если узел – лист
        // записываем узел в массив листьев
        symbol_nodes[nodes[curr_node].sym] = curr;
      else
      {
        nodes[nodes[curr_node].child].parent = curr;
        nodes[nodes[curr_node].child + 1].parent = curr;
      }
      HuffmanNode buf = nodes[curr];    // меняем узлы местами
      nodes[curr] = nodes[curr_node];
      nodes[curr_node] = buf;
      // переопределяем родителей
      short bufparent = nodes[curr].parent;
      nodes[curr].parent = nodes[curr_node].parent;
      nodes[curr_node].parent = bufparent;
    }
    if (curr != nodes[curr_node].parent) // если узел перемещен
      // переходим к родителю перемещенного узла
      curr_node = nodes[curr].parent;
    else
      curr_node = nodes[curr_node].parent;
  }
}

virtual int Encode()   // сжатие файла
{
  short current_node;
  unsigned char code[256];
  reader = new Reader(fnamein);
  if (reader->isopen())
  {
    if (PutFileExtension())
    {
      unsigned long long flen = reader->getfilelength();
      // записываем длину исходного файла в архив
      writer->writebyte((long long*)&flen);
      unsigned char c;
      bool end_of_file = false;
      do
      {
        if (!reader->readbyte(&c))
        {
          current_node = symbol_nodes[eof];
          end_of_file = true;
        }
        else
```

```
        if (symbol_nodes[c] == -1)
          current_node = symbol_nodes[esc]; // номер узла esc
        else
          current_node = symbol_nodes[c];   // номер узла c
       short depth = 0;
       do                // получаем код Хаффмана
       {
        if (current_node & 1)
          code[depth++] = 1;
        else
          code[depth++] = 0;
        current_node = nodes[current_node].parent;
       } while (current_node);
       depth--;
       while (depth >= 0)          // записываем код Хаффмана
       {
        writer->writebit(code[depth]);
        depth--;
       }
       // если для байт-кода узел не создан (записан esc)
       if (symbol_nodes[c] == -1)
       {
        // записываем 8 битов байт-кода c, начиная со старшего
        for (int i = 0; i < 8; i++)
        {
          writer->writebit((c >> (7 - i)) & 1);
        }
        AddList(c, true);
       }
       UpdateTree(symbol_nodes[c]);
      } while (!end_of_file);
      writer->flush();
      writer->close();
      return 0;
    }
    return 2;
  }
  return 1;
}

void AddList(unsigned char c, bool encode)  // добавление листа
{
  // сдвигаем три последних узла на две позиции
  nodes[free_node + 1] = nodes[free_node - 1];
  nodes[free_node] = nodes[free_node - 2];
  nodes[free_node - 1] = nodes[free_node - 3];
```

```
  // вставляем новый внутренний узел
  nodes[free_node - 3] = { 0, -1, nodes[free_node - 1].parent,
                           free_node - 2 };
  // вставляем новый лист для байт-кода c
  nodes[free_node - 2] = { 0, c, free_node - 3, -1 };
  // переопределяем связи
  nodes[free_node - 1].parent = free_node - 3;
  nodes[free_node - 1].child = free_node;
  nodes[free_node].parent = free_node - 1;
  nodes[free_node + 1].parent = free_node - 1;
  if (encode)  //только в режиме сжатия
  {
    // записываем новые индексы листьев
    symbol_nodes[nodes[free_node].sym] = free_node;
    symbol_nodes[nodes[free_node + 1].sym] = free_node + 1;
    symbol_nodes[c] = free_node - 2;
  }
  free_node += 2;
}

virtual int Decode()   // декодирование файла
{
  unsigned char bit, code;
  long long count = 0;
  reader = new Reader(fnamein);
  if (reader->isopen())
  {
    if (GetFileExtension())
    {
      unsigned long long flen, count = 0;
      reader->readbyte((long long*)&flen);
      short node = 0;
      while (true)
      {
        bit = reader->readbit();    // читаем из архива бит
        if (bit == 1)               // спускаемся по дереву
          node = nodes[node].child;
        else node = nodes[node].child + 1;
        if (nodes[node].child == -1)   // спустились до листа
        {
          if (nodes[node].sym == eof)  // если лист "eof",
            break;                     // завершаем цикл
          else
          if (nodes[node].sym == esc)  // если лист "esc",
          {                            // читаем байт-код
            code = 0;
```

```
            for (int i = 0; i < 8; i++)
            {
              bit = reader->readbit();
              code = (code << 1) + bit;
            }
            writer->writebyte(&code);
            count++;
            AddList(code, false);
            UpdateTree(free_node - 4);
          }
          else                          // записываем байт-код
          {
            writer->writebyte((unsigned char *)&nodes[node].sym);
            count++;
            UpdateTree(node);
          }
          node = 0;
        }
      }
      writer->flush();
      writer->close();
      if (count == flen)  // длина полученного файла совпадает
        return 0;         // с длиной исходного файла
      else
        return 3;
    }
    return 2;
  }
  return 1;
 }
};
```

Список литературы

1. *Nelson*, *M*. The Data Compression Book / M. Nelson, J.-L. Gailly. — 2nd ed. —New York : M&T Books, 1995. — 541 p.
2. *Berkun*, *S*. The Myths of Innovation. — O'Reilly Media, Inc., 2010. — 180 p.
3. *Burrows*, *M*. A Block-sorting Lossless Data Compression Algorithm / M. Burrows, D. J. Wheeler. — 1994. — 18 p.
4. *Пантелеев*, *Е*. *Р*. Методы сортировки и поиска : учеб. пособие. — Иваново : Ивановский гос. энергетический ун-т, 2006. — 80 с.
5. *Huffman, David A.* A Method for the Construction of Minimum-Redundancy Codes // Proceedings of the IRE. — Vol. 40. — Sept. 1952. — P. 1098–1101. — DOI: 10.1109/JRPROC.1952.273898.
6. *Мастрюков, Д.* Алгоритмы сжатия информации. Часть 1. Сжатие по Хаффмену // Монитор. — 1994. — № 1. — С. 20–26.
7. *Knuth, Donald E.* Dynamic Huffman Coding // Journal of Algorithm. — 6(2). — 1985. — P. 163–180.
8. *Vitter, J. S.* Design and analysis of dynamic Huffman codes // Journal of ACM. — 34, 4. — Oct. 1987. — P. 825-845.
9. *Abramson, N.* Information Theory and Coding. — New York : McGraw-Hill, 1963. — [This textbook contains the first reference to what was to become the method of arithmetic coding (pp. 61–62)].
10. *Мастрюков, Д.* Алгоритмы сжатия информации. Часть 2. Арифметическое кодирование // Монитор. — № 7–8. — 1993. — С. 20–26.
11. *Иринёв А., Каширин В*. Арифметическое кодирование // Computer technologies department, ITMO University. — URL: http://rain.ifmo.ru/cat/data/theory/data-compression/arithmetic-coding-2006/article.pdf (дата обращения 04.02.2018).
12. *Di Pierro, M.* What is the Blockchain? // Computing in Science & Engineering. — 2017. — Т. 19. — № 5. — P. 92–95.

Евгений Рафаилович ПАНТЕЛЕЕВ,
Алевтина Леонидовна АЛЫКОВА

АЛГОРИТМЫ СЖАТИЯ ДАННЫХ БЕЗ ПОТЕРЬ

Учебное пособие

Издание второе, стереотипное

Зав. редакцией
литературы по информационным технологиям
и системам связи *О. Е. Гайнутдинова*

ЛР № 065466 от 21.10.97
Гигиенический сертификат 78.01.10.953.П.1028
от 14.04.2016 г., выдан ЦГСЭН в СПб

Издательство «ЛАНЬ»
lan@lanbook.ru; www.lanbook.com;
196105, Санкт-Петербург, пр. Юрия Гагарина, 1, лит. А.
Тел.: (812) 412-92-72, 336-25-09.
Бесплатный звонок по России: 8-800-700-40-71

Подписано в печать 05.08.22.
Бумага офсетная. Гарнитура Школьная. Формат 60×90 $^{1}/_{16}$.
Печать офсетная/цифровая. Усл. п. л. 10,75. Тираж 30 экз.

Заказ № 1119-22.

Отпечатано в полном соответствии
с качеством предоставленного оригинал-макета
в АО «Т8 Издательские Технологии».
109316, г. Москва, Волгоградский пр., д. 42, к. 5.