МИНИСТЕРСТВО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

УРАЛЬСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

кафедра молекулярной физики

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНОЙ РАБОТЕ

# КОМПРЕССИЯ ДАННЫХ ИЛИ ИЗМЕРЕНИЕ И ИЗБЫТОЧНОСТЬ ИНФОРМАЦИИ. МЕТОД ХАФФМАНА

ПО КУРСУ «ТЕОРИЯ ИНФОРМАЦИОННЫХ СИСТЕМ»

ДЛЯ СТУДЕНТОВ ДНЕВНОЙ ФОРМЫ ОБУЧЕНИЯ СПЕЦИАЛЬНОСТЕЙ:

07.19.00 - ИНФОРМАЦИОННЫЕ СИСТЕМЫ В ТЕХНИЧЕСКОЙ ФИЗИКЕ

Екатеринбург 2000

УДК 774:002:006.354

Составители: О. Е. Александров, Попков В.И.

Научный редактор: канд. физ.-мат. наук О. Е. Александров

КОМПРЕССИЯ ДАННЫХ ИЛИ ИЗМЕРЕНИЕ И ИЗБЫТОЧНОСТЬ ИНФОРМАЦИИ. Метод Хаффмана: Методические указания к лабораторной работе / О. Е. Александров, Попков В.И. Екатеринбург: УГТУ-УПИ, 2000. 52 с.

Библиогр. 1 назв. Рис. 23. Табл. 1. Прил. 1.

Изложена краткая теория сжатия информации и понятие избыточности информации. Приведена классификация методов сжатия. Описаны несколько алгоритмов полностью обратимого сжатия данных (сжатие без потерь).

Приведено подробное описание алгоритма метода Хаффмана.

Исходный код программы для лабораторной работы доступен по адресу «http://www.mp.dpt.ustu.ru/InformationSystemsTheory».

Материалы предназначены для студентов кафедры «Молекулярная физика».

Подготовлено кафедрой «Молекулярная физика».

**СОДЕРЖАНИЕ**

## ПЕРЕЧЕНЬ УСЛОВНЫХ ОБОЗНАЧЕНИЙ СИМВОЛОВ, ЕДИНИЦ И ТЕРМИНОВ

MPEG – Motion Pictures Expert Group;
JPEG – Joint Pictures Expert Group;
0111b – суффикс «b» означает число, записанное по основанию 2 — двоичное число;
0ABCh – суффикс «h» означает число, записанное по основанию 16 — шестнадцатиричное число;
111. – суффикс «.» означает число, записанное по основанию 10 — десятичное число;
$N_1..N_2$ – диапазон целых чисел от $N_1$ до $N_2$;

## ВВЕДЕНИЕ

Любая информационная система должна обеспечивать выполнение следующих основных функций: прием, хранение, передача, обработка и выдача информации. Причем хранение и передача информации занимает важное место.

Нынешний век называют информационным веком, ибо информация играет все более и более важную роль в современной жизни. Ее объемы постоянно возрастают, и, таким образом, требуются все большие и большие накопители и все более быстрые каналы связи для передачи. Но повышение емкости хранилищ и скорости линий передачи либо невозможно технически, либо не оправдано экономически. Таким образом, приходится подстраиваться под существующие возможности. Но поскольку просто уменьшать объем информации нежелательно, то приходится искать другие способы уменьшения. То есть надо как-то уменьшить объем информации, не изменяя ее. Такой процесс называется архивацией, компрессией или сжатием данных.

На практике возможно сжатие практически любой, т.н. «обычной» информации. Почему? Прежде всего, потому, что «обычное» представление информации, которым люди привыкли пользоваться, почти всегда избыточно. Избыточность присутствует текстах, так как в них обязательно есть повторяющиеся слова, фразы, а то и целые абзацы. Избыточность информации присуща звукам речи, так как в ней обязательно есть частоты, не слышимые человеком, или несущественные для восприятия. Аналогично, избыточно представление информации в электронном виде, обязательно есть какие-то повто-

ряющиеся символы, цепочки символов. Удалив избыточность мы можем уменьшить потребности в информационных емкостях, необходимых для хранения информации и при этом не уменьшив содержательную сторону информации, т.е. сохранив возможность восстановления ее к исходному виду. Таким образом, удаляя избыточность информации, можно уменьшить ресурсы, необходимые для хранения и передачи данных.

Исходный код программы для лабораторной работы доступен по адресу «http://www.mp.dpt.ustu.ru/InformationSystemsTheory».

## 1. СПОСОБЫ СЖАТИЯ ИНФОРМАЦИИ

Сжатие данных можно разделить на два основных типа:

1) сжатие без потерь (или полностью обратимое);
2) сжатие с потерями (когда несущественая часть данных утрачивается и полное восстановление невозможно).

Первый тип сжатия применяют, когда данные важно восстановить после сжатия в неискаженном виде, это важно для текстов, числовых данных и т.п. Полностью обратимое сжатие, по-определению, ничего не удаляет из исходных данных. Сжатие достигается только за счет иного, более экономичного, представления данных.

Второй тип сжатия применяют, в основном, для видеоизображений и звука. За счет потерь может быть достигнута более высокая степень сжатия. В этом случае потери при сжатии означают несущественное искажение изображения (звука) которые не препятствуют нормальному восприятию, но при сличении оригинала и восстановленной после сжатия копии могут быть замечены.

Ниже более подробно будут рассмотрены алгоритмы сжатия без потерь. Алгоритмы сжатия с потерями рассмотрим поверхностно.

Как уже было сказано алгоритмы с потерями применяются в основном для сжатия звука и изображений. Дело в том, что при сжатии такого рода информации, теряя какую-то часть данных при сжатии, мало теряем собственно полезных данных, а объем уменьшается существенно. В результате получаются какие-то искажения исходных данных, зависящие от того, какую долю данных мы теряем. Чем больше потери, тем выше степень сжатия и тем сильнее искажения. Но умеренных искажений, как правило, человек не замечает. Это связано с особенностями его восприятия звука и изображений. Алгоритмы построены так, чтобы при разумной степени компрессии искажения были практически незаметны. Но, разумеется, профессиональный фотограф, к примеру, или человек близкой к нему профессии сразу опознает искажения фотографии, музыкант

или человек с хорошим музыкальным слухом отличит предварительно сжатый звук от оригинала.

Существует много форматов графических файлов. Наиболее известны BMP, PCX, GIF, JPEG. Некоторые из них предусматривают сжатие.

Алгоритм GIF является примером тривиального алгоритма с потерями и позволяет достичь хорошей степени сжатия. Сжатие достигается за счет урезания количества цветов до 256, поэтому при сохранении фотографии резко ухудшается цветопередача.

Более сложным является алгоритм с потерями JPEG. Достоинством стандарта является возможность изменения качества изображения в зависимости от требований к качеству и необходимого объема изображения.

Каждое изображение на входе делится на неперекрываемые блоки пикселей размером 8×8, затем каждый блок подвергается дискретно-косинусному преобразованию (ДКП), аналогичному преобразованию Фурье. Так как это преобразование двухкоординатное, то базисом ДКП являются функции с увеличивающимися частотами в горизонтальной и вертикальной плоскостях. Если после преобразования на приемную сторону были переданы все коэффициенты, то восстановленное изображение не уступает по качеству исходному. Когда абсолютно подлинного качества не требуется, то передаются не все коэффициенты, а лишь то их количество, которое обеспечит нужное качество изображения. В этом случае после ДКП каждый из 64 полученных коэффициентов по однообразной схеме квантируется с определенным квантирующим значением, выражающим весовое значение каждого коэффициента, субъективно влияющего на качество восприятия изображения человеческим глазом. Поскольку значения коэффициентов у ближайших блоков сильно коррелированны, то кодируются не сами коэффициенты, а разница в значениях коэффициентов предыдущего и текущего блоков. Ненулевые квантированные значения оставшихся коэффициентов и их местоположение сканируются зигзаговым проходом блока, а затем кодируются кодовыми словами переменной длины.

Декодер на приемной стороне осуществляет обратное преобразование. Сначала для каждого блока пикселов декодируется битовый поток с целью определения значения и местоположения ненулевых значений коэффициентов ДКП. После восстановления всех ДКП – коэффициентов производится обратное преобразование, в результате чего получаем блок пикселов.

Чтобы предоставить пользователю возможность управлять характеристиками сжатия, в стандарте JPEG реализована концепция 64-элементной таблицы квантирования. Процесс квантирования заключается в делении каждого коэффициента ДКП на соответствующее ему значение из таблицы и последующего округления результата до ближайшего целого значения. Видно, что степень сжатия может быть задана квантируемым значением. Сильное увеличение

степени сжатия приведет к появлению искажений. Поэтому необходимо достигать баланса между степенью сжатия и качеством изображения.

Рассмотрим теперь сжатие видео. Для начала примерно оценим объем одной секунды видеоданных. Пусть кадр имеет размер 320×240 и глубину цвета 16 бит. При таком разрешении и цветности качество изображения довольно низкое. Размер одного кадра 320×240×2 = 150 *Кбайт*. В секунду необходимо не менее 30 кадров для плавной их смены. Таким образом, 1 секунда видео такого формата занимает 150×30=4.5 *Мбайт*. При этом мы не учитываем звук, а также то, что для более высокого качества необходимо больший размер кадра.

Видно, что размер обычного фильма получается огромным. Но, к счастью, разработаны методы сжатия, которые позволяют сжать видеопоток в десятки раз. Например, это формат MPEG.

В основе семейства MPEG лежит тот факт, что видеопоток содержит избыточность в двух направлениях – пространственном и временном. Поэтому используется внутрикадровое и межкадровое кодирование —корреляция отдельных элементов изображения и отдельных кадров.

Кодирование по MPEG начинается с интерполяции видеоинформации, убирающей субъективную избыточность, содержащуюся в видео. После этого используются различные методы сжатия.

Например, энтропийное кодирование. Это тот самый алгоритм Хаффмана, о котором я уже говорил, только здесь используются не отдельные символы, а цвета.

Другой метод – кодирование с предсказанием. Уменьшение избыточности осуществляется за счет определения окружения пиксела внутри кадра или между кадрами. Этот метод дает хороший результат, если существует хорошая пространственная корреляция ближайших друг к другу пикселов. Некоторые пиксели аппроксимируются с помощью ближайших к ним, а разница между действительным значением цвета и предсказанным – небольшая величина — подвергается энтропийному кодированию.

## 2. АЛГОРИТМЫ СЖАТИЯ БЕЗ ПОТЕРЬ

В настоящее время существует значительное количество алгоритмов сжатия без потерь, частично это открытые[1] алгоритмы, частично коммерческие[2]

[1] Открытые алгоритмы обычно рассматривают только саму идею, не останавливаясь на проблемах ее реализации в виде программы.

[2] Коммерческий алгоритм обычно включает в себя не только идею алгоритма сжатия, но и эффективную реализацию алгоритма в виде программы.

алгоритмы. Коммерческие алгоритмы не публикуются и познакомится с ними невозможно, за исключением ознакомления с результатами работы программ на базе этих алгоритмов. Соответствующие программы (ZIP, ARJ, RAR и др.) достаточно известны и с ними можно познакомиться самостоятельно.

### 2.1. Алгоритм Running

Один из самых простых алгоритмов сжатия без потерь — так называемый алгоритм *Running* или *RLE*. Допустим, мы имеем цепочку одинаковых символов. Налицо явная избыточность информации. Сжатие осуществляется следующим образом:

1) подсчитываем количество одинаковых символов в цепочке;
2) вместо цепочки записываем, символ и сколько раз повторяется этот символ.

Например, строка из 40 пробелов. Записываем байт-код, показывающий что идет повторяющийся символ, затем число 40 (сколько раз повторяется), и, наконец, пробел (повторяющийся символ). Строка длиной 40 сжимается до 3 байт.

Данный алгоритм прост и в понимании, и в реализации, но он малоэффективен при применении его в одиночку. Существуют файлы, которые содержат повторяющиеся символы, но не могут быть сжаты описанным выше алгоритмом. Это происходит, когда повторяется не символы, а последовательность символов. Например: «абабабабабаб». Повторяющихся друг за другом символов нет, и сжать предыдущим алгоритмом нельзя. Приходится пользоваться расширенным алгоритмом. Он заключается в следующем:

1) Ищем не повторяющиеся символы, а повторяющиеся последовательности символов (в данном случае это «аб»).
2) Затем подсчитываем, сколько раз повторяется эта последовательность.
3) Записываем по следующему правилу: байт-код сигнала начала последовательности, количество повторяющихся строк (в нашем случае 6), и сама строка (в нашем случае «аб»).

Но и это не предел. Дело в том, что могут существовать одинаковые фрагменты текста, располагающиеся в разных местах файла. Основная проблема состоит в том, чтобы найти эти фрагменты. Далее его необходимо записать в кодовую таблицу и приписать ему определенное значение — код. После этого мы можем заменять все фрагменты этим кодом-ссылкой.

Реализация такого поиска — весма сложная и трудоемкая задача. Хотя существуют особые ситуации, когда подобный поиск повторяющихся фрагментов может быть ускорен. Например, при передаче видеоизображения данные представляют собой последовательность кадров — картинок, состоящих из массива точек. Эти картинки имеют одинаковый размер и, что гораздо более важно, часто предыдущая картинка слабо отличаются от последующей (простейший

пример, снимается неподвижный объект). В этом случае реализация *Running*-подобного алгоритма достатояно проста. Достаточно записывать для каждого следующего кадра только его отличия от предыдущего и можно добыться существенного сжатия данных для медленно меняющихся изображений.

Вышеперечисленные алгоритмы работают, только если в файле имеются повторяющиеся фрагменты и символы.

Существуют и алгоритмы другой природы. Рассмотрим довольно интересный алгоритм, которому не требуется этих последовательностей.

### 2.2. Алгоритм Хаффмана

Данные для электронной обработки представляют собой некоторую последовательность чисел. Обычно, каждое число в этой последовательности представляется битовой цепочкой фиксированной длины. Например, если рассматривать данные как последовательность байтов (8-и битных кусочков), то битовая цепочка[3] 0000000b = 0, 0000001b = 1, 11111111b = 255 и т.д.

Идея алгоритма Хаффмана заключается в следующем. Если некоторые данные содержат различные числа, представленные в виде битовых цепочек фиксированной длины, и частота[4], с которой различные числа присутствуют в этих данных, существенно различается, то заменив битовые цепочки фиксированной длины на битовые цепочки различной длины, причем так, чтобы более часто встречающимся числам соответствовали бы более короткие цепочки, можно получить уменьшение объема данных.

Далее будем предполагать данные последовательностью байт (8-и битных кусочков), хотя это необязательно и можно применять алгоритм Хаффмана к битовым цепочкам любой фиксированной длины.

Рассмотрим пример: имеем данные длиной в 100 байт, включающие шесть различных чисел: 1, 2, 3, 4, 5, 7.

Сжимая данные по алгоритму Хаффмана, первое, что необходимо сделать — это подсчитать сколько раз встречается каждое число в данных.

После подсчета частоты вхождения каждого числа мы получаем таблицу частот (рис. 2.1.). Таблица имеет ненулевые значения частот для чисел 1, 2, 3, 4, 5, 7. Числа с нулевой частотой далее не участвуют в алгоритме, о них можно просто забыть. Ячейки таблицы будем называть «узлами».

Найдем в таблице наименьшие частоты. Для этого удобно отсортировать

---

[3] Суффикс «b» означает, что предшествующее число записано в двоичном виде, а суффикс «.» — число записано в десятичном виде.

[4] Частота — количество раз, которое данный байт присутствует в данных отнесенное к полному числу байт в данных. Здесь и далее под частотой будет подразумеваться просто количество раз, которое данный байт присутствует в данных, без отнесения к к полному числу байт.

таблицу по возрастанию. В нашем случае это 5 и 10. Сформируем из узлов 5 и 10 новый узел, частота вхождения для которого будет равна сумме частот, рис. 2.2. Сами узлы, образовавшие новый узел, более не участвуют в создании новых узлов.

Новый узел и оставшиеся узлы таблицы частот образуют новую таблицу, для которой повторяют операцию добавления узла. Самая низкая частота 10 (узел для числа 7) и 15 (новый узел). Снова добавим узел, см. рис. 2.3.

Продолжаем добавлять узлы, пока не останется единственный узел (корень), рис. 2.4. Структура, изображенная на рис. 2.4, где каждый нижележащий узел имеет связь с двумя вышележащими узлами носит название «двоичное дерево» (далее просто дерево).

Теперь когда дерево создано, можно вычислить коды (битовые цепочки для кодирования исходных чисел) и закодировать данные. Вычисление кода числа начинается от корня дерева. Для вычисления кода, необходимо, двигаясь по дереву от корня к числу в исходной таблице, подсчитать число пройденных узлов. Это значение будет равно длине битовой цепочки кода. И проследить для каждого узла повороты, если поворот в узле осуществляется налево, то в цепочке битов устанавливается значение 0, если направо — 1.

Например, для числа 3 от корня до исходного узла пройдем два узла, т.е. длина кода два бита. Первый от корня поворот направо 1 и второй тоже направо 1 — код Хаффмана для числа равен 11b. Выполнив вычисление для всех чисел, получим следующие коды Хаффмана:

ТАБЛИЦА ЧАСТОТ (ИСХОДНАЯ ТАБЛИЦА) МЕТОДА ХАФФМАНА

Полная частотная таблица

| Число | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | … | 254 | 255 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Частота | 0 | 10 | 20 | 30 | 5 | 25 | 0 | 10 | … | 0 | 0 |

Ненулевая часть частотной таблицы

| Число | 1 | 2 | 3 | 4 | 5 | 7 |
|---|---|---|---|---|---|---|
| Частота | 10 | 20 | 30 | 5 | 25 | 10 |

Рис. 2.1

ФОРМИРОВАНИЕ ПЕРВОГО УЗЛА В МЕТОДЕ ХАФФМАНА

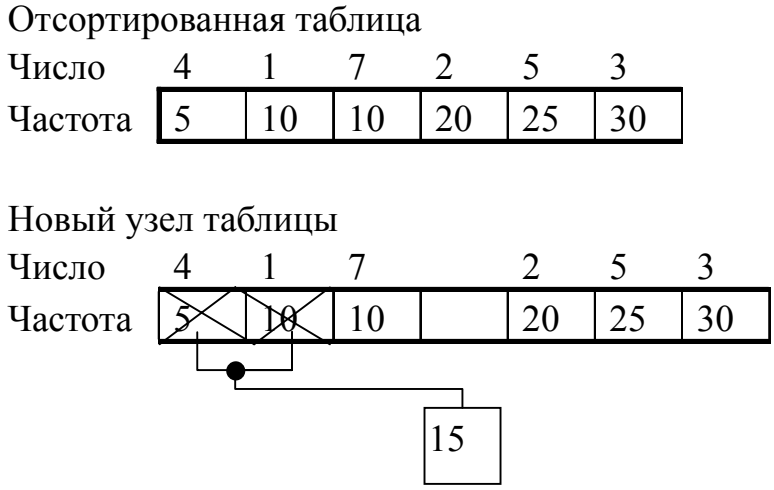



Рис. 2.2

1. $\rightarrow$ 0111b (4 бита );
2. $\rightarrow$ 00b ( 2 бита );
3. $\rightarrow$ 11b ( 2 бита );
4. $\rightarrow$ 0110b ( 4 бита );
5. $\rightarrow$ 10b ( 2 бита );
7. $\rightarrow$ 010b ( 3 бита ). (2.1)

Каждый символ изначально представлялся 8-ю битами, и так как мы уменьшили число битов необходимых для представления каждого символа, мы, следовательно, уменьшили размер выходного файла. Результирующее сжатие может быть вычислено следующим образом, см. табл. 2.1.

ФОРМИРОВАНИЕ ВТОРОГО УЗЛА В МЕТОДА ХАФФМАНА

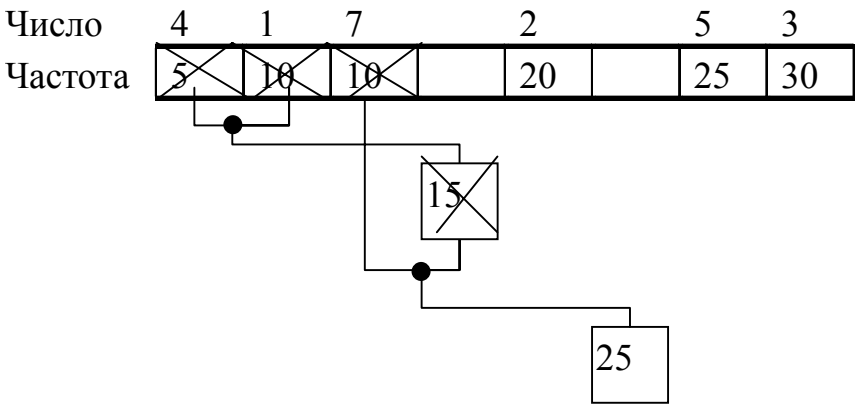



Рис. 2.3

Дерево узлов в методе Хаффмана

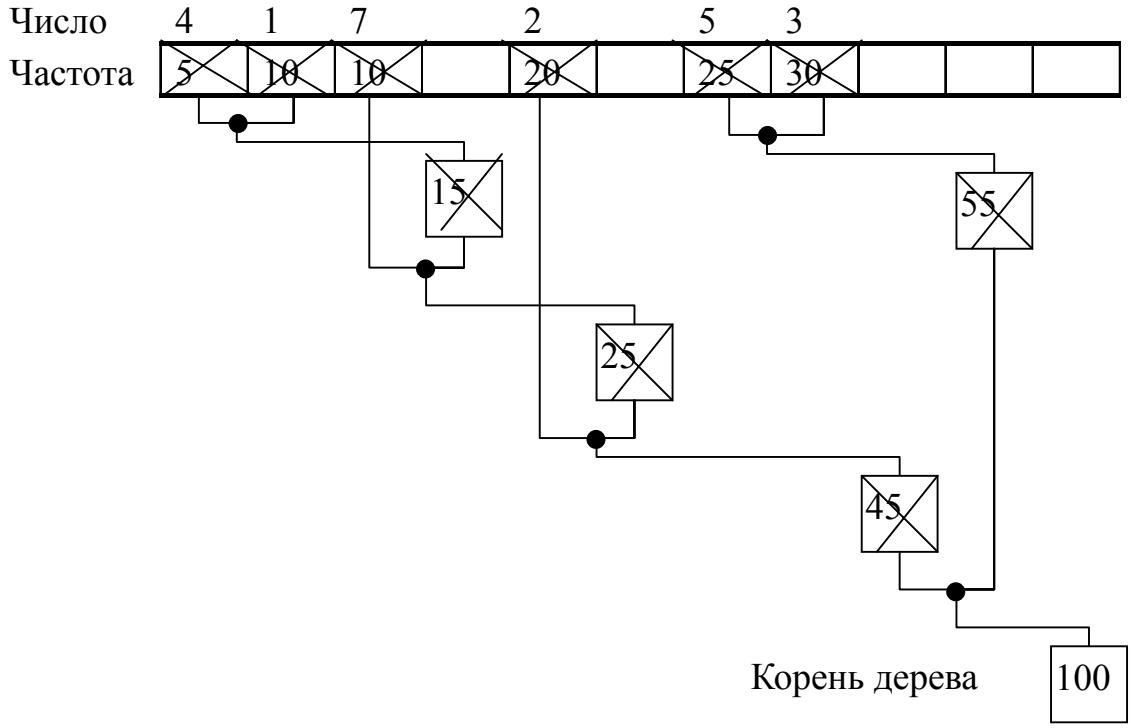



Рис. 2.4

Первоначальный размер данных: 100 *байт* или 800 *бит*; размер сжатых данных: 30 *байт* или 240 *бит*; так что получили размер данных 0,3 исходного. Заметим, что в той же пропорции же будет уменьшен размер данных для более длинной порции данных, если относительные частотные характеристики не изменятся.

Следует помнить, что для восстановления первоначального данных, необходимо иметь декодирующее дерево. Следовательно, мы должны сохранить дерево вместе с данными. Это приводит в итоге к увеличению размеров сжатых данных. Если для ссылки на узлы использоват номер узла в массиве узлов, то такой номер может изменяться от 0 до 511 (для байта), т.е. для хранения ссылки необходимо 9 бит. В рассмотренном примере дерево состоит из 5 узлов по две ссылки (18 бит), итого 90 бит необходимо для сохранения информации по дереву.

Правильный подсчет степени сжатия: первоначальный размер данных: 100 *байт* или 800 *бит*; размер сжатых данных 240 *бит* и 90 *бит* данных о дереве — получили суммарнай размер данных 0,41 исходного.

Метод Хаффмана не всегда способен уменьшить размер данных. Очевидно, что для данных, где встречаются все числа (0÷255) и частоты их равны

Таблица 2.1

Вычисление степени сжатия даных

| Число | Частота | Первоначальный размер, *бит* | Уплотненный размер, *бит* | Уменьшение размера, *бит* |
|---|---|---|---|---|
| 1. | 10 | 80 | 40 | 40 |
| 2. | 20 | 160 | 40 | 120 |
| 3. | 30 | 240 | 60 | 180 |
| 4. | 5 | 40 | 20 | 20 |
| 5. | 25 | 200 | 50 | 150 |
| 7. | 10 | 80 | 30 | 50 |
| ВСЕГО | 100 | 800 | 240 | 560 |

получится длина кода для каждого числа 8 *бит* и никакого сжатия. Вероятно можно придумать и другие сочетания приводящие к аналогичному результату.

Тут следует заметить, что невозможность сжать данные по алгоритму Хаффмана не означает отсутствие возможности сжатия вообще. Например, очевидно, что любая последовательность, где встречаются все числа (0÷255) с равными частотами несжимаема по Хаффману. Но, если числа в данных расположены группами состоящими из одинаковах чисел, например: 1,1,1,1,7,7,7,7 и т.д., то такие данные легко сжать методом *Running*.

Вопрос о степени избыточности информации в произвольных данных в общем случае не решен.

### 2.3. Статический и адаптивный (динамический) алгоритм Хаффмана

Описанный выше алгоритм требует предварительной обработки всех исходных данных — должны быть вычислены частоты вхождения для байтов. После чего данные могут быть упакованы, т.е. необходимо два просмотра исходных данных. Кроме того, этот алгоритм требует сохранения данных о частотах (или двоичном дереве) вместе с упакованными данными, иначе распаковка невозможна. Алгоритмы сжатия, которые обладают такими свойствами, называют **статическими**. Эти алгоритмы трудно использовать при кодировке потока данных, например, при передаче данных по сети.

Существуют алгоритмы сжатия, которые не требуют предварительной обработки всего массива данных, а способны кодировать (упаковывать) данные по мере их поступления (поток данных). Этим алгоритмам достаточно однократного просмотра данных, например, к таким алгоритмам сжатия относится

метод Лемпеля-Зива. Алгоритмы этого типа называют **адаптивными (динамическими)**.

Хотя метод Хаффмана по своей сути статический, но существует возможность реализовать адаптивный вариант этого метода.

# 3. ВОЗМОЖНАЯ РЕАЛИЗАЦИЯ МЕТОДА ХАФФМАНА

## 3.1. Общие замечания

Исходный код программы для лабораторной работы доступен по адресу «http://www.mp.dpt.ustu.ru/InformationSystemsTheory».

Прежде всего, следует обеспечить автономность и возможность и повторного использования кода, который будет написан. Для этого следует реализовать алгоритм в виде процедуры или набора процедур.

Далее, следует определиться с основными процедурами, которые нам нужны при компрессии/декомпрессии данных. В общем случае, нас бы устроили, по-видимому, всего две процедуры:

1) *Закодировать*.
2) *Декодировать.*

Первая процедура должна осуществлять сжатие по методу Хаффмана, а вторая — декодировать сжатые данные к исходному виду.

Потом, следует определиться с аргументами (данными) над которыми будут проводить операции эти две процедуры. Здесь можно указать на следующие соображения: 1) компрессия/декомпрессия должны осуществляться с максимальной скоростью; 2) аргументы процедур должны подходить для любого типа информации (или возможно более широкого типа исходных данных), т.е. быть достаточно универсальными.

Исходя из этих соображений можно предложить ограничиться операцией компрессия/декомпрессия над участком оперативной памяти ЭВМ (далее этот участк оперативной памяти ЭВМ будет именоваться *буфером*). Это даст достаточно гибкий механизм компрессии и для файлов, файл можно последовательно считывать в буфер, упаковывать и записывать результат в новый файл — файл-архив. Таким образом наши процедуры должны иметь следующие аргументы:

1) *Закодировать*(*ИсходныйБуфер*; *РазмерИсходногоБуфера*; var *ЗакодированныйБуфер*; var *РазмерЗакодированногоБуфера*).
2) *Декодировать*(*ЗакодированныйБуфер*; *РазмерДекодированныхДанных*; var *ДекодированныйБуфер*).

Переменные (точнее их содержимое), помеченные «var» изменяются в результате работы процедуры.

Следует определиться с типами переменных, обеспечив максимальную гибкость. Для *Буфера* наиболее подходящим будет тип открытый массив байтов — array of *byte*. Заботу о выделении этих участков памяти (буферов) мы возложим на внешнюю программу. *Размер* буфера может изменяться от 0 до 65535 для DOS-программы, поэтому можно определить его размер как *Размер:word* — слово (16 битное беззнаковое целое). В Delphi размер буфера может быть существенно больше и там тип переменной можно определить как *Размер: dword* — двойное слово (32 битное беззнаковое целое).

Окончательно, мы получили — необходимо создать две процедуры:

1) *Закодировать*(*ИсходныйБуфер*: *array of Byte*; *РазмерИсходногоБуфера*: *word*; var *ЗакодированныйБуфер*: *array of Byte*; var *РазмерЗакодированногоБуфера*: *word*).
2) *Декодировать*(*ЗакодированныйБуфер*: *array of Byte*; *РазмерДекодированныхДанных*: *word*; var *ДекодированныйБуфер*: *array of Byte*;).

Создание этих процедур сразу и целиком представляет сложную задачу. Для упрощения следует разбить каждую процедуру на более простые операции (процедуры). Например, процедура *Закодировать* должна выполнить следующие действия:

1) Подсчет частот вхождений байт в буфере.
2) Построение дерева для буфера.
3) Вычисление по дереву кодов Хаффмана.
4) Запись данных для декодировки буфера.
5) Запись кодированного буфера.

Аналогично, процедура *Декодировать* должна выполнить следующие действия:

1) Чтение данных для декодировки буфера.
2) Запись декодированного буфера.

В процессе работы эти процедуры должны хранить свои данные и форма хранения должна обеспечивать:

1) Хранение данных как единого целого в удобном виде.
2) Удобный доступ к элементам данных.
3) Блокировку (по возможности) непреднамеренной ошибочной модификации данных пользователем.

### 3.2. Структуры данных для метода Хаффмана

Проектирование данных (т.е. типов данных, места хранения и способа доступа) при создании любой программы является зачастую не менее важной

задачей, чем разработка самого алгоритма. Правильно и эффективно спроектированные данные позволят быстрее написать код алгоритма и сделать этот код более эффективным, понятным и безошибочным.

Рассмотрим возможную структуру данных для реализации метода Хаффмана. Начнем с задачи построения и хранения информации о двоичном дереве.

Двоичное дерево метода Хаффмана (далее просто дерево) состоит из узлов. Максимальное возможное число узлов дерева равно удвоенной длине полной исходной таблицы символов минус 1. Для байта (0..255 — всего 256 различных значений) максимальное число узлов дерева равно 2×256 - 1 или 511 узлов.

Поскольку при генерации дерева желательно максимальное быстродействие и размер максимального дерева невелик, то программа может разместить статический массив узлов максимального размера. Элементы массива будут использоваться для построения дерева.

Так как для декодировки необходима информация о дереве и желательно минимального размера (ее приходится записывать вместе с кодированным буфером), то реально данные лучше разместить в виде двух массивов (определения типов и данных см. в файлах, перечисленных в п. 4.1):

1) минимальная часть, необходимая для декодировки (см. типы данных[5] *tNode*, *tNodes* и массив *Nodes* в объекте *tHuffman*);
2) доп. данные, необходимые для построения дерева и вычисления кода (см. типы данных[5] *tNodeData*, *tNodesData* и массив *NodesData* в объекте *tHuffman*).

Таким образом данные по дереву состоят из двух массивов, как это изображено на рис. 3.1.


Рис. 3.1

[5] Все ссылки на определения типов имеют ввиду следующие файлы: «HufTypes.pas» и «Huffman0.pas».

ДАННЫЕ ДЛЯ ДВОИЧНОГО ДЕРЕВА

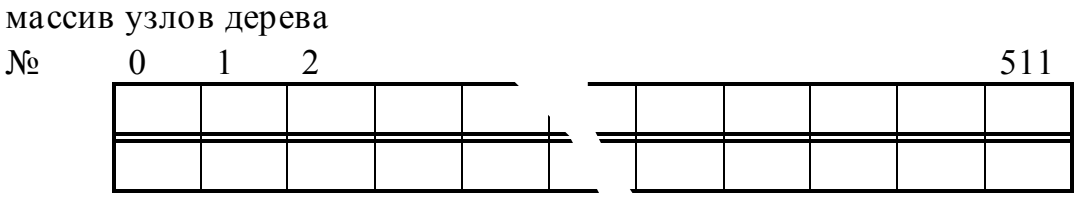



Рис. 3.2

Далее мы будем называть оба этих массива «массив узлов дерева», не различая частей (рис. 3.2). Узлы с 0 по 255 используются как исходная таблица метода Хаффмана, а узлы с 256 по 511 — для построения собственно двоичного дерева. Ссылка на узел производится по его номеру, узлы в массиве никогда не изменяют своего порядка. Каждый узел содержит данные, как это изображено на рис. 3.3.

В процессе построения дерева удобнее работать с упорядоченной по возрастанию частоты вхождений последовательностью узлов, с одной стороны, и иметь доступ к исходной последовательности узлов (исходной таблице), с другой стороны.

Для организации упорядоченного списка узлов без изменения порядка в исходном массиве узлов используется индексный массив *Indexes* или массив ссылок (см. типы данных *tIndexes* и массив *Indexes* в *tHuffman*). Массив индексов представляет собой последовательность ссылок на узлы (номеров узлов в массиве узлов), расположенных в порядке возрастания частот вхождений, рис. 3.4.

Проще говоря для узлов любых двух индексов $i<j$ выполняется: частота вхождения для *Nodes*[ *Indexes*[*i*] ] меньше или равна частоте вхождения для *Nodes*[ *Indexes*[*j*] ].

ДАННЫЕ ДЛЯ УЗЛА ДВОИЧНОГО ДЕРЕВА

узел дерева (*tNode*)

| | |
|---|---|
| ссылка на предыдущий левый узел | для узлов исходной таблицы = -1, что интерпретируется программой как «отсутствует предыдущий узел». |
| ссылка на предыдущий правый узел | |
| счетчик частоты вхождений для узла | |
| ссылка на следующий узел | не обязательно, можно обойтись и без этого параметра |

Рис. 3.3

ИНДЕКСНЫЙ МАССИВ ДЛЯ УЗЛОВ ДВОИЧНОГО ДЕРЕВА

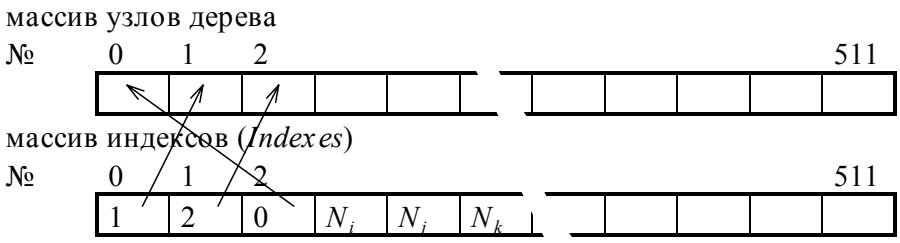



Рис. 3.4

### 3.3. АЛГОРИТМ ПОСТРОЕНИЯ ДЕРЕВА

1) Массив узлов заполняется начальными значениями, рис. 3.5.

ИСХОДНЫЙ МАССИВ ДЛЯ ДВОИЧНОГО ДЕРЕВА

массив узлов дерева (*Nodes+NodesData)*

| № | 0 | 1 | 2 | | | | | | | | 511 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | -1 | -1 | -1 | -1 | -1 | | -1 | -1 | -1 | -1 | -1 |
| | -1 | -1 | -1 | -1 | -1 | | -1 | -1 | -1 | -1 | -1 |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |

Рис. 3.5

2) Массив индексов заполняется начальными значениями, рис. 3.6.

ИСХОДНЫЙ ИНДЕКСНЫЙ МАССИВ ДЛЯ УЗЛОВ ДВОИЧНОГО ДЕРЕВА

массив индексов (*Indexes*)

| № | 0 | 1 | 2 | 3 | | | | | | 510 | 511 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | | | | | | 510 | 511 |

Рис. 3.6

3) Подсчитываются частоты вхождений для байтов в исходной таблице, рис. 3.7.

МАССИВ ИСХОДНЫХ ДАННЫХ С ЧАСТОТАМИ

массив узлов дерева (*Nodes+NodesData*)

| № | 0 | 1 | 2 | | | | 255 | | | | 511 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | -1 | -1 | -1 | -1 | -1 | | -1 | -1 | -1 | -1 | -1 |
| | -1 | -1 | -1 | -1 | -1 | | -1 | -1 | -1 | -1 | -1 |
| | $F_0$ | $F_1$ | $F_2$ | $F_3$ | $F_4$ | | $F_{255}$ | 0 | 0 | 0 | 0 |
| | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |

Рис. 3.7

4) Сортируется по возрастанию частот вхождений часть массива индексов, соответствующая исходной таблице (с 0 по 255 индекс), рис. 3.8.

ИНДЕКСЫ ДЛЯ ДВОИЧНОГО ДЕРЕВА С ЧАСТОТАМИ

массив узлов дерева (*Nodes+NodesData*)

| № | 0 | 1 | 2 | 3 | 4 | | 255 | | | | 511 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | -1 | -1 | -1 | -1 | -1 | | -1 | -1 | -1 | -1 | -1 |
| | -1 | -1 | -1 | -1 | -1 | | -1 | -1 | -1 | -1 | -1 |
| | $F_0$ | $F_1$ | $F_2$ | $F_3$ | $F_4$ | | $F_{255}$ | 0 | 0 | 0 | 0 |
| | -1 | -1 | -1 | -1 | -1 | - | -1 | -1 | -1 | -1 | -1 |

массив индексов (*Indexes*)

| № | 0 | 1 | 2 | | | | | | | | 511 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 4 | 0 | $N_k$ | | | | | | | | |

$F_4 \le F_0$ - условие сортировки.

Рис. 3.8

5) Устанавливаются указатели (индексы):
   а) На первый ненулевой индекс (ссылающийся на узел со счетчиком вхождений большим 0) — далее *IndexFirst*.
   б) На максимальный индекс (ссылающийся на узел с максимальным счетчиком вхождений) — далее *IndexLast*.

Данные индексы соответствуют началу и концу исходной таблицы. Ссылки на нулевые узлы более не участвуют в построении дерева.

ИСХОДНАЯ ИНДЕКСНАЯ ТАБЛИЦА ДВОИЧНОГО ДЕРЕВА

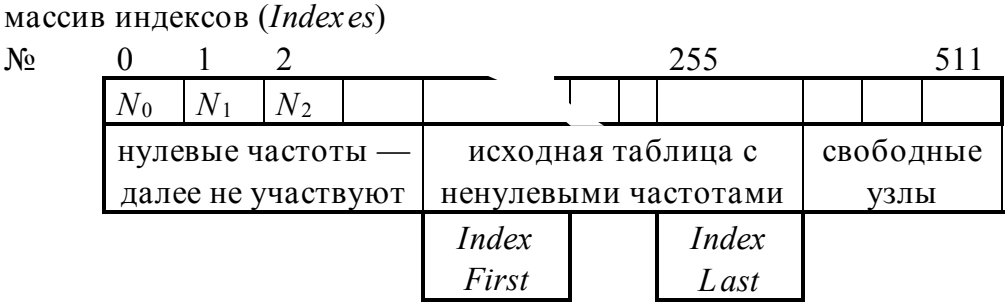



Рис. 3.9

4) Из исходной таблицы выбираются два левых узла: *индекс*[*IndexFirst*] и *индекс*[*IndexFirst*+1] — это узлы с минимальными частотами вхождений — и объединяются в новый узел. В качестве нового узла используется свободный узел с номером *индекс*[*IndexLast*+1].

ДОБАВЛЕНИЕ НОВОГО УЗЛА В ДВОИЧНОЕ ДЕРЕВО

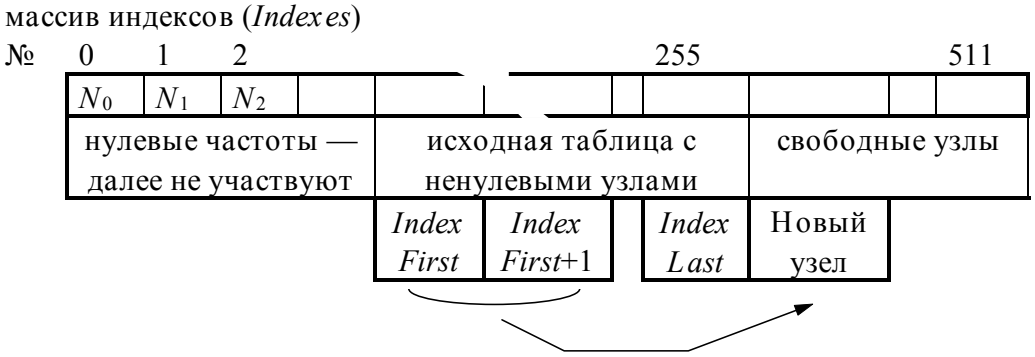



Рис. 3.10

5) Создается новая таблица частот, рис. 3.11.
   а) Два левых узла исходной таблицы удаляются, путем установки указателя *IndexFirst* = *IndexFirst* + 2.
   б) Новый узел добавляется в таблицу, путем установки указателя *IndexLast* = *IndexLast* + 1.
   в) Новая таблица сортируется по возрастанию частот вхождений.

СОЗДАНИЕ НОВОЙ ТАБЛИЦЫ ЧАСТОТ

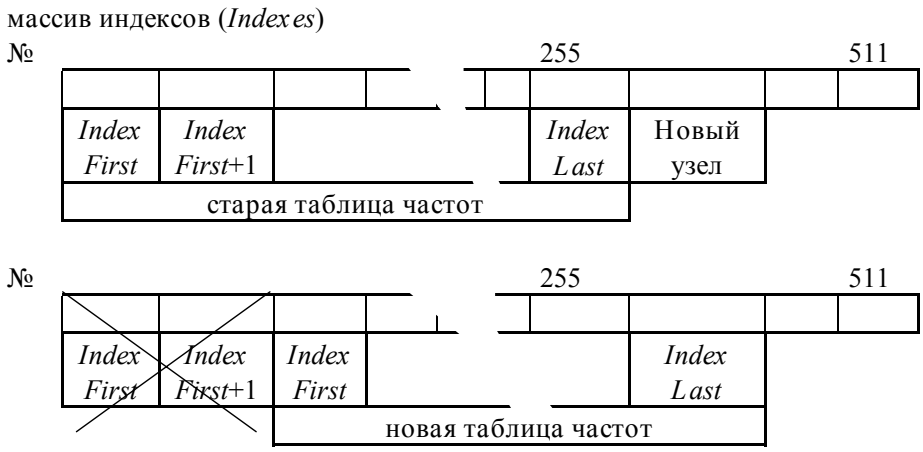



Рис. 3.11

6) Проверяется, количество узлов в новой таблице, если их больше одного, то повторяются шаги с пункта 4), иначе построение дерева считается законченным и *индекс*[*IndexLast*] указывает на корневой узел двоичного дерева. Узлы (0..255) содержат информацию по первоначальной таблице, а узлы (256..*IndexLast*) содержат минимальную информацию по двоичному дереву, необходимую для декодировки.

Построение дерева закончено.

### 3.4. БОЛЕЕ БЫСТРЫЙ АЛГОРИТМ ПОСТРОЕНИЯ ДЕРЕВА

Можно строить дерево еще быстрее. Для этого следует обратить внимание на то, что вновь генерируемые узлы с 256 по 511 (т.е. новые узлы, которые мы получаем последовательно объединяя по два узла с минимальными частотами вхождений) образуют упорядоченную по возрастанию частот вхождений последовательность. Это позволяет ограничиться сортировкой только базовой последовательности узлов с 0 по 255, а затем для выбора узлов с минимальными частотами вхождений использовать разновидность сортировки называемой «сортировка слиянием». В этом варианте сравниваются очередной узел базовой последовательности узлов с 0 по 255 и очередной узел вновь генерируемой последовательности узлов с 256 по 511 и берется минимальных из них. Для выбора двух узлов требуется только два сравнения, что заведомо меньше, чем необходимо сравнений для сортировки. Пример реализации быстрого алгоритма построения дерева можно посмотреть в файле «HuffBase.pas», см. процедуру *BuildTreeAdvanced*.

### 3.5. Алгоритм вычисления кодов Хаффмана

Вычисление кодов для элементов первоначальной таблицы (узлы 0..255) можно провести используя либо поле «ссылка на следующий узел» (прямой проход по дереву от вершины к корню), см. рис. 3.3, либо рекурсивным проходом, используя поля «ссылка на предыдущий левый узел» и «ссылка на предыдущий правый узел».

Само вычисление не представляет сложности и рассматриваться здесь не будет, см. пример вычисления в пункте 2.2. После вычисления получим массив кодов для каждого из чисел исходной таблицы, как это показано на рис. 3.12.

Размер данных, выделяемых под код Хаффмана, должен обеспечивать хранение:

1) Длины кода.
2) Кода.

Код Хаффмана может занимать от 1 бита (минимальная длина кода) до 256 бит (максимальная длина кода), т.е. необходимо зарезервировать 32 байта под хранение кода.

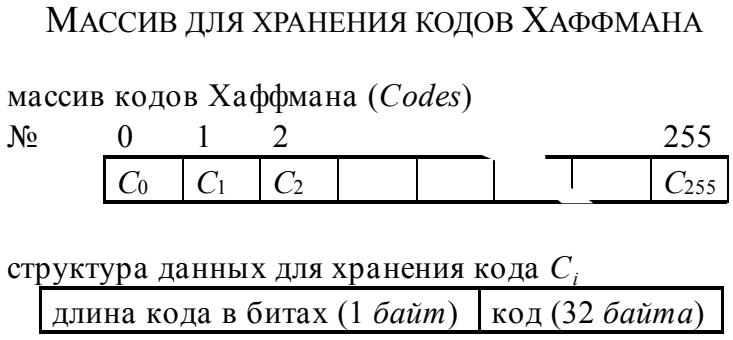



Рис. 3.12

### 3.6. Размер и формат записи упакованных данных

Прежде чем приступать к кодированию, после построения дерева, разумно подсчитать будущий размер кодированных данных. Если размер кодированных данных меньше размеров исходных данных, то можно будет приступить к кодированию.

Размер $S_d$ кодированных данных (без дерева и вспомогательной информации) в байтах можно вычислить по формуле[6]:

[6] Здесь и далее размер порций данных округляется до полного байта.

$$S_d = \left[ \frac{\left( \sum_{i=0}^{255} Частота_i \cdot Длина\_кода\_в\_битах_i \right) + 7}{8} \right] байт, \qquad (3.1)$$

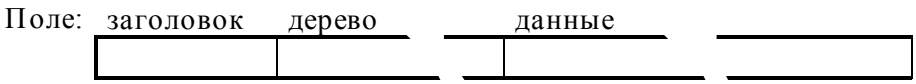
здесь квадратные скобки означают взятие целой части числа, [*X*] = (*целая часть X*). Размер упакованного дерева $S_t$ можно вычислить так

$$S_t = \left[ \frac{(Число\_узлов\_в\_дереве - 256) \cdot 2 \cdot 9 + 8 + 7}{8} \right] байт, \qquad (3.2)$$

здесь (*Число_узлов_в_дереве* - 256) - число узлов данные которых нужно сохранить, 256 узлов исходной таблицы сохранять нет необходимости. Каждый узел содержит две ссылки, максимальная ссылка 511 или 9 бит, плюс сведения о длине дерева — 8 бит.

Размер вспомогательных данных зависит от формата записи упакованных данных. Можно, например, использовать следующий формат для упакованного буфера: *заголовок*, *упакованное дерево*, *упакованные данные*, рис. 3.13.

ФОРМАТ УПАКОВАННЫХ ДАННЫХ




Структура данных заголовка:

| | |
|---|---|
| Полная длина буфера, байт.<br><br>Размер поля *word* (2 *байта*). | Полная длина распакованных данных, байт.<br>Размер поля *word* (2 *байта*). |

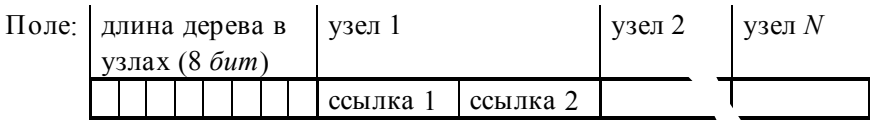
Рис. 3.13

Заголовок (см. рис. 3.13) содержит сведения о длине упакованных данных, необходимые для их считывания и распаковки. Размер заголовка

$$S_h = 4 \text{ } байта. \qquad (3.3)$$

Упакованное дерево может быть записано в формате показанном на рис. 3.14.

ФОРМАТ ДАННЫХ УПАКОВАННОГО ДЕРЕВА




Структура узла:

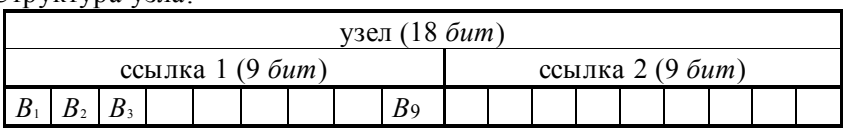



Рис. 3.14

Размер дерева (количество узлов в дереве) записывается в первые 8 бит, далее следуют соответствующее количество записей узлов. Каждый узел содержит 18 бит, по 9 бит на ссылку. Полная длина упакованного дерева может быть вычислена по формуле (3.2).

Если суммарный размер (см. формулы (3.1), (3.2) и (3.3))

$$S = S_d + S_t + S_h, \qquad (3.4)$$

меньше, чем размер исходных данных, то желаемый результат достигнут — данные могут быть упакованы.

### 3.7. Кодирование данных

Для кодирования данных необходимо два буфера:

1) Буфер с исходными данными.
2) Буфер для закодированных данных.

Поскольку размер закодированных данных не будет превосходить размер исходных данных, то разумно в программе выделить сразу два одинаковых буфера — один используется как «*буфер с исходными данными*», а второй как «*буфер для закодированных данных*».

Прежде всего в «*буфер для закодированных данных*» записывают заголовок и упакованное дерево. Обсуждать детали этого процесса нет смысла (см. процедуру *EncodeBufEx1* в файле HuffmanE.pas).

Далее в «*буфер для закодированных данных*» записывают кодированные данные, полученные из исходных данных. Процесс кодирования байта исходных данных состоит из следующих шагов:

1) чтение из «*буфера с исходными данными*» очередного байта;
2) выбор из кодовой таблицы соответствующего кода;
3) копирование битов выбранного кода в «*буфер для закодированных данных*».

Схематически процесс кодирования байта исходных данных можно представить как это показано на рис. 3.15. Кодирование повторяется для всех байтов «*буфера с исходными данными*».

Теперь «*буфер для закодированных данных*» содержит закодированные данные с полной длиной *S*, которую можно вычислить по формуле (3.4). Закодированные данные можно записать в файл-архив, а в «*буфер с исходными данными*» считать новую порцию данных и продолжить архивацию.

ПРОЦЕСС КОДИРОВАНИЯ БАЙТА ИСХОДНЫХ ДАННЫХ

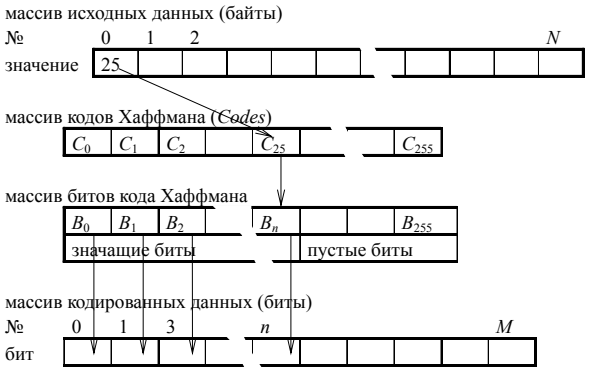



Рис. 3.15

Здесь следует заметить, что хотя исходные данные имеют всегда фиксированную длину, размер кодированных данных может быть различным для различных исходных данных. Поэтому при сохранении кодированного буфера следует сохранить и сведения о его длине. В нашем случае (см. рис. 3.13) полная длина закодированного буфера может быть считана из заголовка.

Пример процедуры кодирования приведен в ПРИЛОЖЕНИИ.

## 3.8. Декодирование данных

### 3.8.1. Простой алгоритм декодирования проходом по дереву

Для декодирования данных необходимо:

1) Буфер с закодированными данными.
2) Буфер для декодированных данных.
3) Данные по дереву.

Может показаться, что для декодировки достаточно знания кодов Хаффмана. Но взглянув на пример кодов Хаффмана (см. (2.1)), становится очевидным, что прямое сравнение порции закодированных данных с кодами Хаффмана затруд-

нительно из-за различной длины кодов. Хотя такое решение проблемы декодировки, по-видимому, возможно.

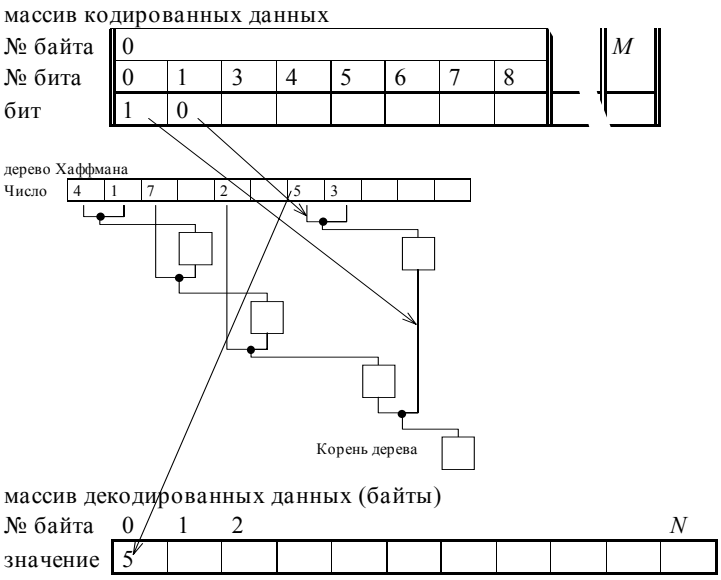
Проще использовать следующий алгоритм декодировки байта:

1) Восстанавливается структура дерева в *Nodes*. Достаточно восстановить ссылки на вышележащие узлы, частотные данные для декодировки не нужны.
2) Текущим узлом выбирается корень дерева (*n* = *IndexLast*).
3) Считывается один бит из «*буфера с закодированными данными*», если бит равен 1, то устанавливается текущим правый предыдущий узел, иначе если бит равен 0 — устанавливается текущим левый предыдущий узел.
4) Повторяется операция считывания очередного бита кодированных данных и перемещения по дереву (см. шаг 3), пока номер текущего узла не станет меньше 256.
5) Записывается номер текущего узла в «*буфер для декодированных данных*».
6) Повторяются операции с шага 2), пока не декодируем весь «*буфер с закодированными данными*».

Схематически процесс декодирования байта данных можно представить как это показано на рис. 3.16.

Процесс декодирования продолжается, пока не будет декодирован весь «*буфер с закодированными данными*». Возможная реализация декодировки приведена в ПРИЛОЖЕНИИ.

ПРОЦЕСС ДЕКОДИРОВАНИЯ ДАННЫХ




В качестве примера использовано дерево Хаффмана для данных (2.1), см. рис. 2.4. Частотная информация для декодировки не нужна.

Рис. 3.16

### 3.8.2. Ускоренная табличная декодировка

Описанный выше процесс декодировки простой, но довольно медленный. Это связано с необходимостью многократно обходить дерево, декодируя последовательность битов. Предпочтительнее было бы декодировать код Хаффмана целиком или «одношагово», но реализация такого алгоритма затруднена тем, что коды имеют различную длину, к тому же, не кратную 8 битам. Однако предполагаемый выигрыш «одношаговой» декодировки настолько высок[7], что есть смысл попытаться ее реализовать, хотя бы частично.

Для реализации «почти одношаговой» декодировки можно вспомнить, что большинство кодов должны иметь длину менее 8 бит (иначе невозможно сжатие методом Хаффмана) и лишь некоторые коды длиннее 8 бит. Можно предложить одношагово декодировать 8 первых бит кода[8] и, если код длиннее,

[7] Вместо цикла декодирования 8 бит (8 шагов), можно получить декодирование за один шаг, что ускорит процесс в разы. Хотя очевидно, что ускорение будет не 8-и кратным, а меньшим, поскольку сложность шага декодировки бита и байта могут различаться — декодировка байта несколько сложнее.

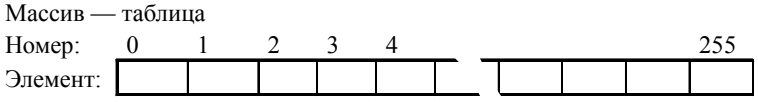
[8] Ограничившись первыми 8 битами мы получаем приемлемый размер таблицы — 256 элементов, если декодировать 16 бит, то нужна таблица в 65535 элементов. Использование таблицы с промежуточными длинами (более 8 и менее 16 бит) затруднительно из-за необходимости манипулировать данными не кратными байту.

то декодировать остаток проходом по дереву. В этом случае большая часть кодов будет декодироваться «одношагово», что существенно ускорит процесс декодирования.

Для реализации такого алгоритма, нужна дополнительная структура данных, назовем ее «*таблицей-ускорителем декодировки*». *Таблица-ускоритель* должна содержать ссылки на узлы, а индексом таблицы должны служить первые 8 бит кода Хаффмана. Структура данных таблицы может быть организована, например, так как это изображено на рис. 3.17.

Некоторая сложность возникает при декодировании таким образом коротких (менее 8 бит) кодов. Дело в том, что получив байт (8 бит) из «*буфера с закодированными данными*», невозможно до декодирования узнать какая часть из этих 8 бит относится к коду одного байта, а какая — к коду следующего. Решить проблему можно следующим образом. Для всех кодов с длиной менее 8 бит генерируются все возможные продолжения для битовой цепочки до длины в 8 бит. Для каждого из этих новых кодов (код Хаффмана + продолжение до 8 бит) ссылка в таблице устанавливается на соответствующий коду Хаффмана узел дерева. После декодировки по таблице значащие биты (проверяется по значению *Длина_кода*, см. рис. 3.17) считаются декодированными, а остаток битов должен принять участие в декодировке следующего байта.

ФОРМАТ ТАБЛИЦЫ-УСКОРИТЕЛЯ ДЕКОДИРОВКИ




Структура элемента таблицы:

| | |
|---|---|
| Номер узла в массиве узлов *Nodes* дерева Хаффмана (ссылка). Размер поля *word*. | Длина кода, если она меньше 8 бит или 8, если длина кода больше или равна 8 бит. Размер поля *byte*. |

Рис. 3.17

### 3.9. МАНИПУЛЯЦИИ С БИТАМИ

Технически сложной операцией в процессах кодирования-декодирования являются операции с битовыми последовательностями. Связано это с особенностью конструкции процессора ЭВМ — процессор оперирует с минимальной порцией в 1 *байт* или 8 *бит*. И, соответственно, в языках программирования обычно отсутствуют инструкции типа: «прочитать *n*-ый бит» и «записать *n*-ый бит». Нет в них и структуры данных «массив битов».

Чтобы оперировать с битами необходимо иметь представление о внутренней структуре хранения данных в ОЗУ и операциях доступа к битам.

Простейшая операция чтения *N*-го бита из массива байтов *B* выполняется так (нумерация битов и байтов идет с нуля):

1) Вычисляется номер нужного байта в массиве байтов: $N_b := N$ div 8. Здесь «div» — оператор целочисленного деления *N* на 8.
2) Считывается байт *b* содержащий нужный бит: $b := B[N_b]$.
3) Вычисляется номер нужного бита: $n_b := N$ mod 8. Здесь «mod» — оператор вычисления остатка целочисленного деления *N* на 8.
4) Считывается значение бита *bit* из байта: $bit := (b \text{ shr } n_b)$ and 1. Здесь «shl» — оператор сдвига битовой цепочки влево, «shr» — оператор сдвига битовой цепочки вправо.
5) В результате *bit*=1, если *N*-й бит из массива байтов *B* равен 1 и *bit*=0, если *N*-й бит из массива байтов *B* равен 0.

Простейшая операция записи бита *bit* в *N*-й бит в массив байтов *B* выполняется так (нумерация битов и байтов идет с нуля):

1) Вычисляется номер нужного байта в массиве байтов: $N_b := N$ div 8. Здесь «div» — оператор целочисленного деления *N* на 8.
2) Считывается байт *b* содержащий нужный бит: $b := B[N_b]$.
3) Вычисляется номер нужного бита: $n_b := N$ mod 8. Здесь «mod» — оператор вычисления остатка целочисленного деления *N* на 8.
4) Вычисляется новое значение для байта *b* так, чтобы изменить только нужный бит. Если *bit* =1, то $b := b$ or (1 shl $n_b$), иначе $b := b$ and (not (1 shl $n_b$)). Здесь «or» — оператор побитового «ИЛИ», «and» — оператор побитового «И» , «not» — оператор побитового «НЕ».
5) Записывается бит *b* обратно в массив битов: $B[N_b] := b$.

Все упомянутые битовые операции приведены в нотации Borland Pascal.

# 4. ЗАДАНИЕ ДЛЯ САМОСТОЯТЕЛЬНОГО ВЫПОЛНЕНИЯ

## 4.1. Материалы для выполнения задания

Вам предоставляется:

1) Исходный код простой программы, демонстрирующей сжатие буфера с данными методом Хаффмана, состоящей из модулей:
   - «Test.pas» — основная программа;
   - «Common\xStrings.pas» — вспомогательные модули.
   - «HufTypes.pas» — определения структур данных;

- «uMiscFunc.pas» — вспомогательные функции;
- «HuffBase.pas» — основные процедуры сжатия по методу Хаффмана, осуществляющие операции по сжатию;
- «Huffman0.pas» и

«HuffmanE.pas» — определения функций упаковки буфера для метода Хаффмана.

Эти модули позволяют получить программу TEST.exe, осуществляющую кодировку буфера с данными. Данные в программе генерируются генератором случайных чисел.

1) Исходный код простой программы, демонстрирующей сжатие файла методом Хаффмана, состоящей из модулей:
   - «FTest.pas» и
     «FCmdLine.pas» — основная программа;
   - см. пречень модулей пункта 1).
   - «HuffmanF.pas» — определения функций упаковки файла для метода Хаффмана.

Эти модули позволяют получить программу FTest.exe, осуществляющую упаковку/распаковку файла.

### 4.2. Варианты заданий

Задание лабораторной работы выполняется индивидуально. Варианты помеченные звездочкой имеют повышенную сложность и могут выполняться группой в 2 человека. Варианты помеченные звездочкой дают право на освобождение от экзамена (при полном выполнении) или на освобождение от одного вопроса на экзамене (при частичном выполнении). Уровень полное/частичное выполнение определяет преподаватель.

Вариант 1 (стандартный)

Состоит из трех частей. Выполнение первой части задания дает право гордо заявлять: «Я делал лабораторную работу». Выполнение первой и второй части задания дает право на освобождение от 1 (одного) вопроса на экзамене по выбору. Выполнение первой, второй и третей части задания дает право на освобождение от экзамена.

*Первая часть задания*

1) Научиться компилировать программу и запускать тестовую программу FTest.exe, см. ПРИЛОЖЕНИЕ.
2) Научиться упаковывать и распаковывать произвольный файл с помощью тестовой программы.

*Вторая часть задания*

1) Объяснить, почему файл сжатый тестовой программой не сжимается еще сильнее другими архиваторами (zip, rar и т.п.), либо сжимается очень незначительно. И, что особенно важно, файл сжатый тестовой программой сжимается хуже (имеет больший размер архива), чем если бы архиватор сжимал исходный файл.
2) Объяснить, почему файлы одинакового размера сжимаются в разной степени. При каких условиях файлы одинакового размера будут сжаты в одинаковой степени? При каких условиях файлы разного размера будут сжаты в одинаковой степени? Под степенью сжатия понимается отношение размера исходного файла к размеру архива.
3) Какой файл будет иметь максимальную степень сжатия тестовой программой (алгоритмом Хаффмана)?
4) Какой файл не будет сжиматься тестовой программой (алгоритмом Хаффмана) совсем?
5) Создать самый длинный файл с максимальным сжатием тестовой программой (алгоритмом Хаффмана).
6) Создать самый короткий файл с отсутствием сжатия тестовой программой (алгоритмом Хаффмана).
7) Какой вариант алгоритма Хаффмана: статический или динамический будет иметь наибольшую степень сжатия одних и тех же данных? Существуют ли данные с другим вариантом?

*Третья часть задания*

1) Модифицировать тестовую программу FTest.exe так, чтобы она либо работала быстрее при том же уровне сжатия, либо сжимала более эффективно. Критерии сравнения: ускорение/улучшение сжатия должно быть не менее 1% по сравнению с готовым компилированным вариантом тестовой программы FTest.exe, который предоставлен вам вместе с исходным кодом. Этот вариант откомпилирован с параметрами, обеспечивающими максимальное быстродействие.
2) Сравнение проводится при равных размерах буфера и одинаковых данных.
3) Сравнение должно быть проведено на различных типах файлов, не менее трех: *.doc; *.exe; *.zip. Размер каждого файла должен быть не менее 2 Мбайт.

**ВНИМАНИЕ!!!** Самодельные реализации принимаются только если: 1) вычисляют время сжатия (в миллисекундах), степень сжатия (как отношение размера упакованного файла к размеру исходного файла) и скорость сжатия/декомпрессии (в байтах в секунду); 2) скоростные характеристики или степень сжатия превосходит прилагаемую тестовую программу «FTest.exe» на величину не менее 1%.

Вариант 2*

1. Спроектировать данные, описать алгоритм и написать процедуры для метода *Running*. В качестве образца интерфейса использовать код программы, демонстрирующей сжатие данных методом Хаффмана, см. «Huffman0.pas» и «HuffmanE.pas».
2. Проверить работоспособность метода *Running*.
3. Сравнить быстродействие и эффективность метода *Running* с методом Хаффмана.

Вариант 3*

1. Спроектировать данные, описать алгоритм и написать процедуры для **адаптивного** (динамического) метода Хаффмана. В качестве образца интерфейса использовать код программы, демонстрирующей сжатие данных методом Хаффмана, см. «Huffman0.pas» и «HuffmanE.pas».
2. Проверить работоспособность **адаптивного** метода Хаффмана.
3. Сравнить быстродействие и эффективность **адаптивного** метода Хаффмана со статическим методом Хаффмана.

Вариант 4*

1. Подробно описать алгоритм для ЛЮБОГО метода сжатия без потерь (кроме *Running*, метода Хаффмана и метода Лемпеля-Зива-Уэлча /LZW/) с конкретным примером ручного вычисления простого примера. Оценить быстродействие алгоритма, степень сжатия и эффективность сжатия данных различной степени упорядоченности. В качестве образца описания использовать настоящее руководство.
2. Спроектировать данные, описать алгоритмы процедур и написать процедуры. В качестве образца интерфейса использовать код программы, демонстрирующей сжатие методом Хаффмана.
3. Попробовать проверить работоспособность метода.
4. Попробовать сравнить быстродействие и эффективность метода с методом Хаффмана.

Вариант 5*

1. Усовершенствовать сжатие методом Хаффмана. Например, создать процедуру автоматического подбора наилучшей длины буфера для сжатия или процедуру многократного сжатия буфера методом Хаффмана. Возможны ваши собственные варианты усовершенствований, но следует подтвердить их работоспособность простым и очевидным примером.

---

* Варианты помеченные звездочкой имеют повышенную сложность и выполняются группой в 2 человека.

2. Проверить работоспособность усовершенствований на примере файлового сжатия (см. «FTest.pas» и пункт 4.1, 2) ).
3. Сравнить быстродействие и эффективность усовершенствованого сжатия методом Хаффмана со сжатием при фиксированной длине буфера (см. «FTest.pas»).

Вариант 6*

1. Разработать проект структуры данных для хранения архива (нескольких сжатых файлов в одном файле-архиве).
2. Создать программу управления архивом. Программа должна иметь набор команд: «добавить файл(ы) в архив», «распаковать файл(ы) из архива», «удалить файл(ы) из архива» , «вывести список файлов в архиве».
3. Проверить работоспособность программы (см. «FTest.pas» и пункт 4.1, 2)).

**ВНИМАНИЕ!!!** В качестве метода упаковки можно применять только метод Хаффмана. Программа должна поддержывать ВСЕ указаные выше операции.

### 4.3. Оформление результатов работы

Вы должны представить письменный отчет (один на группу) по выполненной работе и работоспособный код программы. Объем отчета ~20 страниц, не считая листингов программы. Листинги рекомендуется не печатать. Отчет должен быть оформлен в соответствии со стандартом [1].

Отчет должен состоять из следующих частей:

1) титульный лист;
2) введение;
3) основная часть (может состоять из нескольких глав, с произвольными названиями);
4) заключение;
5) список использованных источников.

Отчет должен содержать:

1) краткий обзор математических алгоритмов сжатия информации (приветствуется описание алгоритмов не упомянутых в данной методичке (*Running*, Хаффман);
2) описание проблем, с которыми вы столкнулись при написании программы, и их решений;
3) подробное описание вашего кода и наиболее интересных решений, использованных в нем;
4) описание результатов сравнения эффективности работы вашего и предоставленного вам готового кода.

Работоспособный код вашей программы представляется в виде исходного файла (файлов) программы на дискете. Распечатывать полный листинг не нужно. Страницы с листингом не защитываются в общий объем отчета.

### 4.4. Прием зачета по результатам работы

Зачет принимается в форме обсуждения отчета и программы с членами группы, представившей отчет. При обсуждении каждый из членов группы должен продемонстрировать:

1) Знание основ теории сжатия информации: измерение информации и размер данных, причины и неизбежность избыточности информации, практическая необходимость сжатия данных, пределы сжимаемости и существуют ли несжимаемые данные, основные методы сжатия, алгоритм метода *Running* и метода Хаффмана.
2) Знание устройства и взаимодействия частей представленного кода программы.
3) Умение компилировать и запускать представленный код программы.
4) Умение модифицировать код программы и способность объяснить назначение (функции) отдельных частей кода программы.
5) Умение интерпретировать результаты сравнения работы своего и предоставленного вам готового кода.

## ЗАКЛЮЧЕНИЕ

В результате выполнения этой работы:

1) Вы сможете лучше понять что такое информация.
2) Ознакомитесь с методами ее хранения, обработки и сжатия.
3) Получите практический навык использования алгоритма Хаффмана.
4) Получите практические навыки разработки и кодирования алгоритмов.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. СТП УГТУ-УПИ 1-96. Общие требования и правила оформления дипломных и курсовых проектов (работ). 1996. 34 с. Группа Т51.

# ОБСУЖДЕНИЕ ОСНОВНЫХ ПРОЦЕДУР

**Содержание**



## П.1. Общие замечания

Для выполнения задания 1 необходимо понять как работают следующие процедуры:

1) «CountBytes» — подсчет частот байтов в буфере;
2) «BuildTree» — построение дерева по частотам;
3) «DefineCodesR» — вычисление кодов для байтов по дереву;
4) «Encode» — кодирование буфера по готовым кодам;
5) «Decode» — декодирование буфера по готовому дереву.

Определения (заголовки) этих процедур находятся в файле «Huffman0.pas», а определения типов данных в файле «HufTypes.pas».

Порядок выполнения следующий.

1) Прежде всего, следует ознакомиться с пунктом 2.2 и главой 3 настоящего руководства и понять алгоритм Хаффмана.
2) Потом, следует научиться запускать тестовую программу для проверки работоспособности метода Хаффмана.
3) Затем, следует ознакомиться с определениями и понять структуру данных используемых в файлах «HufTypes.pas» и «Huffman0.pas».
4) Следующий этап — понять логику работы основных функций.
5) После чего, можно приступать к кодированию собственных функций.

Ниже будут рассмотрены примеры кодирования для всех процедур, но степень подробности будет уменьшаться от процедуры к процедуре.

### П.2. Запуск тестовой программы TEST

В комплект задания входит работающий вариант тестовой программы TEST (см. файл «Test.pas»). Текст программы и модулей, кроме модуля «Huffman0.pas», изменять не надо.

Программа выполняет следующие действия:

1) заполняет буфер *b1* (область ОЗУ) случайными данными;
2) запускает компрессию буфера *b1* по методу Хаффмана и помещает результаты в другой буфер *b2*;
3) запускает десомпрессию буфера *b2* по методу Хаффмана и помещает результаты в другой буфер *b3*;
4) сравнивает буфер *b1* и буфер *b3*, и делает вывод о правильной или неправильной работе компрессии/декомпрессии.

Дополнительно вычисляются некоторые характеристики компрессии.

Пример результата выполнения приведен на рис. П.1. Ключевой признак правильной работы — слово «правильно», это означает, что упаковка и последующая распаковка привели к данным, совпадающим с исходными.

Пример результата выполнения программы *Test*

```
Программа HuffmanTest запущена (protected mode)...

 Генерация массива... сжатие (повтор, раз: 55)... завершено.
   Начальный размер данных, байт: 64000
   Размер после сжатия, байт: 46070
   В том числе:
     размер данных после сжатия, байт: 45728
     размер упаков. дерева, байт: 337
     размер заголовка, байт: 5
   Длина неупак. Дерева, узлов: 149, байт: 596
   Минимальная длина кода, бит: 5, максимальная длина кода, бит: 14
   Время на упаковку 1 буфера, мс: 12.4909
   Скорость упаковки, байт/с: 5333333
 Декодирование (повтор, раз: 55)... проверка... правильно.
   Время на распаковку 1 буфера, мс: 3.4909
   Скорость распаковки, байт/с: 21333333

программа HuffmanTest завершена.
```

Рис. П.1

Программа расчитана на использование Borland Pascal 7.0 или Borland Delphi 5.0. Ниже рассмотрен запуск программы для Borland Pascal 7.0.

Для запуска программы:

1) Скопируйте папку «1) Сжатие данных» в свою рабочую папку, например, «C:\users\Я\».
2) Запустите «Norton Commander» или «Far», или другой менеджер файлов для DOS.

3) Перейдите в папку «C:\users\Я\1) Сжатие данных\Программа» и запустите «runme.bat».
4) Перейдите в папку «P:\Программа\1) DOS - BP7» и запустите Borland Pascal 7.0:

   >путь\bp.exe

   Желательно чтобы путь к «bp.exe» был указан в переменной окружения «Path».
5) Установите в качестве основной программы «Test.pas»: меню Compile → Primary file… → «Test.pas».
6) Установите в качестве рабочей среды «Real mode»: меню Compile → Target … → Real Mode Application.
7) Включите все проверки на ошибки: меню Compile → Compiler … → Runtime errors [x] (все).
8) Нажмите клавиши: Ctrl+F9. На экране должно появиться сообщение вида рис. П.1.
9) Сохраните настройки: меню Options → Save As… «.\BP.TP».

Если что-нибудь не получается обратитесь к преподавателю за разъяснениями.

Теперь вы готовы запускать программу и отлаживать собственный вариант метода Хаффмана.

### П.3. Тестовая программа сжатия файла FTEST

В комплект задания входит работающий вариант тестовой программы FTEST (см. файл «FTest.pas»).

Программа может выполнять следующие операции:

1) Упаковывает любой файл методом Хаффмана.
2) Распаковывает упакованный методом Хаффмана файл.
3) Упаковывает методом Хаффмана, распаковывает и сравнивает результат распаковки и оригинальный файл (см. рис. П.2).

Дополнительно вычисляются некоторые характеристики компрессии.

Программа предназначена для более реалистичного (чем сжатие сгенерированного буфера) тестирования метода Хаффмана на реальных данных (файлах).

Программа дает представление о работе архиваторов и может быть использована для сравнения эффективности сжатия Хаффмана с коммерческими архиваторами.

Информация о запуске и работе с программой *FTEST* доступны при запуске программы без параметров в командной строке:

```
>FTEST.exe
```

ПРИМЕР РАБОТЫ *FTEST*

```
Программа HuffmanFileTest запущена (protected mode)...
 Разбор командной строки...
 Завершен разбор командной строки.
 Упаковка: "test.txt" -> "test._tx" (повтор, раз: 5)... завершено.
  Время на 1 упаковку, мс: 1463.4
  Отношение (Размер упакованного файла)/(Размер исходного): 0.8202
 Распаковка: "test._tx" -> "test.!tx" (повтор, раз: 5)... завершено.
  Время на 1 распаковку, мс: 1023.4
 Проверка. Сравнение "test.txt" и "test.!tx"... завершено.
  Правильно.
Программа HuffmanFileTest завершена.
```

Рис. П.2

### П.4. ПОДСЧЕТ БАЙТОВ В БУФЕРЕ

Рассмотрим подробно процесс кодирования на примере простейшей из функций — подсчет байтов в буфере.

В файле «Huffman0.pas» процедура «CountBytes» определена как

```
procedure tHuffman.CountBytes(const Buffer; Size:tBufferIndex);
```

Процедуре передаются два аргумента (*Buffer* и *Size*). Процедура определена как метод объекта, т.е. в ней доступны и глобальные данные объекта *tHuffman*. Определение объекта *tHuffman* можно найти в «Huffman0.pas»:

```
tHuffman=object
    public
      { доступные извне методы (процедуры) и данные}
      ...
    private
      { доступные только здесь методы (процедуры) и данные}
      HuffmanData:tHuffmanFullTreeData; {- глобальные данные объекта,
                                           используются при работе процедур}
      ...
      procedure CountBytes(const Buffer; Size:tBufferIndex);

      ...
end;
```

В определении объекта присутствуют методы (процедуры) и глобальные данные объекта (выше приведена только часть определения). Все методы имеют доступ к глобальным данным объекта. В нашем случае глобальными данными служит структура *HuffmanData:tHuffmanFullTreeData*. Структура *tHuffmanFullTreeData* определена в «HufTypes.pas».

Полный текст процедуры «CountBytes» в файле «Huffman0.pas» выглядит так

```
procedure tHuffman.CountBytes(const Buffer; Size:tBufferIndex);
begin
  HuffBase.CountBytes(Buffer, Size, HuffmanData.NodesData);
end;
```

Процедура обращается к внешней процедуре *HuffBase.CountBytes*, определенной в файле «HuffBase.pas». Следует уяснить что она получает в качестве входных данных и что она должна выдать в качестве результата.

ВХОД в процедуру:

Параметры процедуры:

1) *Buffer* → указывает на начало массива байтов (буфер);
2) *Size* = числу байтов в массиве *Buffer*.

Глобальные данные *HuffmanData.NodesData*:

1) Счетчики частот вхождений *HuffmanData.NodesData*[*i*].*Counter* = 0; *i* = [0..255].

ВЫХОД из процедуры:

Параметры процедуры:

1) *Buffer* - не изменяется;
2) *Size* - не изменяется.

Глобальные данные *HuffmanData*:

1) Счетчики частот вхождений *HuffmanData.NodesData*[*i*].*Counter* = числу байт со значением *i* в массиве *Buffer*; *i* = [0..255];

Процедура «CountBytes» должна просмотреть массив байтов

```
Buffer:array[0..Size-1] of tByte;
```

и сосчитать сколько раз в нем встречаются различные значения байта, т.е. сколько там единичек, двоек, троек и т.д.

Пример, обращение к *i*-тому байту массива *Buffer*:

```
b:=tBuffer(Buffer)[i];
```

эта конструкция присваивает переменной *b* значение *i*-того байта массива *Buffer*. Конструкция *tBuffer(Buffer)* преобразует нетипизованную переменную *Buffer* к типу *tBuffer*, определенному так:

```
tBufferIndex=0..cMaxBufferLength;
tBuffer=array[tBufferIndex] of tByte;
```

Просмотр буфера *Buffer* можно организовать в цикле:

```
for i:=0 to Size-1 do begin
  ...операции с tBuffer(Buffer)[i] и
     HuffmanData.NodesData[tBuffer(Buffer)[i]].Counter...
end;
```

Для хранения данных о количестве различных значений байтов в буфере используется структура *HuffmanData.NodesData*, определенная как массив

```
tNodesData=array[tNodeIndex] of tNodeData;
```

где

```
tNodeData=record
  Counter:tFriquency; {!!! счетчик числа вхождений байта}
  NextNode:tNodeReference; {ссылка на следущий узел дерева(здесь не используется)}
end;
```

где

```
tBufferIndex=0..cMaxBufferLength;
tFriquency=tBufferIndex;
```

Для хранения информации о количестве байт значения которых равны *i*, используется *HuffmanData.NodesData*[*i*].*Counter*. При входе в процедуру *tHuffman.CountBytes* все счетчики *Counter* массива *HuffmanData.NodesData* обнулены.

При просмотре массива *Buffer* мы должны увеличивать соответствующий счетчик. Т.е., например, обнаружив в массиве *Buffer* байт равный 2, мы должны увеличить на единицу *HuffmanData.NodesData*[*2*].*Counter*:

```
HuffmanData.NodesData[2].Counter:=
                        HuffmanData.NodesData[2].Counter+1;
```

такую же операцию мы должны провести для других и для других значений. Например, *tHuffman.CountBytes* может выглядеть так:

```
procedure tHuffman.CountBytes(const Buffer; Size:tBufferIndex);
  var j:tBufferIndex; b:tByte;
begin
  for j:=0 to Size-1 do begin
    b:=tBuffer(Buffer)[j];
    HuffmanData.NodesData[b].Counter:=
                                    HuffmanData.NodesData[b].Counter+1;
  end;
end;
```

Хотя можно написать и более эффективный код. Оптимизацию процедуры «CountBytes» вы можете провести самостоятельно.

В результате выполнения процедуры *HuffmanData.NodesData*[*i*].*Counter* для всех $0 \le i \le 255$ будут иметь значения количества раз, которое байт с значением *i* встретился в буфере *Buffer*.

### П.5. Построение двоичного дерева Хаффмана

В файле «Huffman0.pas» процедура «BuildTree» определена как

```
procedure tHuffman.BuildTree;
```

Процедура не имеет аргументов и работает только с глобальными данными объекта *tHuffman*. Определение объекта *tHuffman* можно найти в «Huffman0.pas», см также пункт П.4.

Следует уяснить что она получает в качестве входных данных и что она должна выдать в качестве результата.

ВХОД в процедуру:

Глобальные данные *HuffmanData*:

1) *HuffmanData.NodesData* (определение см. пункт П.4).

   В МОМЕНТ ВЫЗОВА содержит:

1) Счетчики частот вхождений: *HuffmanData.NodesData*[*i*].*Counter* = числу байт со значением *i* в массиве *Buffer*; *i* = [0..255];
2) Ссылка на следующий узел в дереве Хаффмана: *HuffmanData.NodesData*[*i*].*NextNode* = -1; *i* = [0..511].

2) *HuffmanData.BaseTree.Nodes* — массив узлов для построения двоичного дерева, определен так:

```
tNodes=array[tNodeIndex] of tNode;
```

где

```
tNodeIndex=0..(cMaxNodeCount-1); { 0..511 }
```

```
tNode=record
  UpperNodes:tUpNodesReference;
end;
```

```
tUpNodesReference=record
  case byte of
    0:(Left, {левый предыдущий узел, если =-1, то нет предыдущего}
       Right:tNodeReference); {правый предыдущий узел, если =-1, то нет предыду-
щего}
    1:(Refs:array[0..1] of tNodeReference);
end;
```

В МОМЕНТ ВЫЗОВА содержит:

1) *HuffmanData.BaseTree.Nodes*[*i*].*Left* = -1; *i* = [0..511];
2) *HuffmanData.BaseTree.Nodes*[*i*].*Right* = -1; *i* = [0..511].

3) *HuffmanData.BaseTree.TreeIndexes* — данные по основным узлам дерева, определен так:

```
tTreeIndexes=record
   Root:tIndex; { номер корневого узла дерева }
   MinCode:tBaseNodeIndex; {номер узла с кодом минимальной длины }
   MaxCode:tBaseNodeIndex; {номер узла с кодом максимальной длины }
end;
```

где

```
tNodeIndex=0..(cMaxNodeCount-1); { 0..511 }
tBaseNodeIndex=0..(cBaseNodeCount-1); { 0..255 }
```

В МОМЕНТ ВЫЗОВА содержимое не определено.

ВЫХОД из процедуры:

Глобальные данные *HuffmanData*:

1) *HuffmanData.NodesData* — изменяется:
*HuffmanData.NodesData*[*i*].*NextNode* = номер следующего узла, для узлов включенных в дерево.
2) *HuffmanData.BaseTree.Nodes* — изменяется:
*HuffmanData.BaseTree.Nodes*[*i*].*Left* = номер предыдущего левого узла, *i* = [256 .. *Root*];
*HuffmanData.BaseTree.Nodes*[*i*].*Right* = номер предыдущего правого узла, *i* = [256 .. *Root*], для узлов включенных в дерево.

3) *HuffmanData.BaseTree.TreeIndexes* — изменяется:

*HuffmanData.BaseTree.TreeIndexes.Root* = номер корневого узла дерева;

*HuffmanData.BaseTree.TreeIndexes.MinCode* = номер узла с кодом минимальной длины;

*HuffmanData.BaseTree.TreeIndexes.MaxCode* = номер узла с кодом максимальной длины.

Процедура «BuildTree» должна построить дерево Хаффмана по известным частотам *HuffmanData.NodesData*[*i*].*Counter*, *i* = [0..255] так, как это описано в пунктах 2.2 и 3.3.

Алгоритм процедуры «BuildTree» можно описать так:

1) Сформировать упорядоченный по возрастанию *HuffmanData.NodesData*[*i*].*Counter* список узлов исходной таблицы *i* = [0..255].
2) Исключить из списка узлы с *HuffmanData.NodesData*[*i*].*Counter* = 0.
3) Взять из списка два левых узла ($n_1$ и $n_2$). Эти узлы имеют наименьшие частоты.
4) Исключить эти два узла ($n_1$ и $n_2$) из списка.
5) Создать на основе двух исключенных узлов ($n_1$ и $n_2$) новый узел ($n_n$). Для $n_n$ левым предыдущим узлом будет $n_1$ и правым предыдущим — $n_2$. Для $n_1$ и $n_2$ следующим узлом будет $n_n$.
6) Добавить новый узел в упорядоченный список.
7) Повторить шаги с 3) по 6) пока в списке более чем 1 узел.

Например, код процедуры *BuildTree* может быть следующим (см. файл «HuffBase.pas»), все данные переданы процедуре явно:

```
procedure BuildTreeSimple(
          var   Nodes:tNodes;
          var   NodesData:tNodesData;
          var   TreeIndexes:tTreeIndexes
);
```

{ Размещаем вспомогательный массив индексов узлов - ссылок на узлы в массиве *HuffmanData.Nodes*. Этот массив можно будет отсортировать и получить упорядоченный список узлов по возрастанию частот вхождений. }

```
  var Indexes:tIndexes;
```

{ Определяем вспомогательную функцию — она ищет место для нового узла с частотой *Counter* в диапазоне индексов [*il*..*ir*] массива *Indexes*, методом деления отрезка пополам и возвращает номер узла, перед которым должен находиться новый узел. Диапазон индексов [*il*..*ir*] массива *Indexes* предполагается упорядоченным по возрастанию. }

```
function FindPlace(il,ir:tIndex; Counter:tFriquency):tIndex;
      var im:tIndex;
   begin
     while (il<ir) do begin
       im:=(il+ir) shr 1; { im:=(il+ir) div 2 }
       if NodesData[Indexes[im]].Counter>Counter then
```

```
      ir:=im
      else
      il:=Succ(im);
    end;
    FindPlace:=ir;
  end;
```

{ Определяем вспомогательные переменные }

```
  var
    UpLeftNode,UpRightNode:tNodeIndex;
    NewCounter:tFriquency;
    iFirst,iLast,NodePlace:tIndex;

begin
```

{1) формируем упорядоченный по возрастанию *HuffmanData.NodesData*[*i*].-*Counter* список узлов исходной таблицы *i* = [0..255].

1.1) Заполняем вспомогательный массив индексов начальными значениями *Indexes*[0]:=0; *Indexes*[1]:=1;...}

```
  for i:=0 to 511 do begin
      Indexes[i]:=i;
  end;
```

{ 1.2) Строим упорядоченный индексный массив для исходной таблицы. Для этого необходимо отсортировать массив *Indexes*[*i*] для *i* = [0..255] так, чтобы *HuffmanData.NodesData*[*Indexes*[*i*]].*Counter* ≤ *HuffmanData.NodesData*-[*Indexes*[*j*]].*Counter* для *i* < *j*.

Для сортировки можно применить любой метод. Но использование «продвинутых» методов сортировки, типа QuickSort, здесь неразумно — массив для сортировки мал и выигрыша от применения «продвинутых» методов практически не будет, возможен даже проигрыш в быстродействии.

Можно упорядочивать массив *Indexes* добавляя в него новые элементы в нужное место (сортировка типа сортировки Шелла).

Возмем таблицу состоящую из одного элемента *i*=0 и будем добавлять в нее новые элементы.}

```
for i:=1 to 255 do begin
    { i - номер нового элемента в массиве узлов и массиве Indexes,
      здесь это еще совпадает i = Indexes[i].
      NodePlace - позиция нового элемента в Indexes. }
   NodePlace:=FindPlace(0, i, NodesData[i].Counter);
   { если место нового элемента не совпадает с i, сдвигаем
      остальные элементы }
   if NodePlace<i then begin
     for j:=i downto NodePlace+1 do begin
       Indexes[j]:=Indexes[j-1];
     end;
     {и вставляем его на нужное место. }
     Indexes[NodePlace]:=i;
     { Теперь позиция в массиве узлов и в массиве Indexes различаются}
   end;
end;
```

{ Теперь *Indexes*[0..255] содержит упорядоченный список узлов от 0 до 255. }

{2) Формируем исходную таблицу.

Последний узел таблицы = 255.}

```
iLast:=255;
```

{ Первый узел таблицы — первый узел с ненулевым счетчиком. Ищем в исходной таблице индексов первый узел с ненулевым счетчиком }

```
iFirst:=FindPlace(0, 255, 0);
```

{ узел с минимальной длиной кода Хаффмана}

```
TreeIndexes.MinCode:=Indexes[iLast];
```

{ узел с максимальной длиной кода Хаффмана }

```
TreeIndexes.MaxCode:=Indexes[iFirst];
```

{ Если весь буфер состоит из одного и того же байта, то дерево не будет построено и далее возникнет ошибка. Для ликвидации этого добавляем фальшивый узел, это не приведет к изменению длины кодированного буфера. }

```
if iLast=iFirst then begin
  Dec(iFirst);
end;
```

{ Строим дерево, последовательно добавляя узлы, пока таблица содержит более одного узла: *Indexes*[*iFirst*..*iLast*] → текущая таблица частот. }

```
while iFirst<iLast do begin   { добавить узел }
```

{3) Выбираем первый и второй узлы текущей таблицы для объединения в новый узел, т.к. они имеют минимальные значения счетчиков (таблица отсортирована по возрастанию)}

```
  UpLeftNode:=Indexes[iFirst];
  UpRightNode:=Indexes[Succ(iFirst)];
```

{ исключаем эти узлы (первый и второй) из текущей таблицы }

```
  Inc(iFirst,2);
```

{ добавляем в таблицу новый узел }

```
  Inc(iLast);
```

{ вычисляем сумму счетчиков частоты вхождений}

```
   NewCounter:=NodesData[UpLeftNode].Counter+
                  NodesData[UpRightNode].Counter;
```

{ запоминаем левый предыдущий узел дерева }

```
  Nodes[iLast].UpperNodes.Left:=UpLeftNode;
```

{ запоминаем правый предыдущий узел дерева}

```
  Nodes[iLast].UpperNodes.Right:=UpRightNode;
```

{ Запоминаем сумму счетчиков частоты вхождений в новом узле. }

```
  NodesData[iLast].Counter:=NewCounter;
```

{ Устанавливаем для предыдущих узлов ссылки на следующий узел дерева. }

```
  NodesData[UpLeftNode].NextNode:=iLast;
  NodesData[UpRightNode].NextNode:=iLast;
```

{ Вставляем новый узел на нужное место текущей таблицы индексов для сохранения упорядоченности таблицы по возрастанию:

```
      1) поиск места вставки методом деления отрезка пополам; }
      NodePlace:=FindPlace(iFirst, iLast, NewCounter);
     { 2) вставка индекса узла на место. }
      if NodePlace<iLast then begin
        for j:=i downto NodePlace+1 do begin
           Indexes[j]:=Indexes[j-1];
        end;
        Indexes[NodePlace]:=iLast;
      end;
  end;
{ Запоминаем корневой узел дерева Хаффмана. }
  TreeIndexes.Root:=iLast;
end;
```

Теперь глобальные данные *HuffmanData.BaseTree* содержат бинарное дерево Хаффмана и можно вычислить для него коды символов.

## П.6. Рекурсивное[9] вычисление кодов Хаффмана

В файле «Huffman0.pas» процедура «DefineCodesR» определена как

```
procedure tHuffman.DefineCodesR;
```

Процедура не имеет аргументов и работает только с глобальными данными объекта *tHuffman*. Определение объекта *tHuffman* можно найти в «Huffman0.pas», см также пункт П.4.

Для написания процедуры следует уяснить что она получает в качестве входных данных и что она должна выдать в качестве результата.

ВХОД в процедуру:

Глобальные данные *HuffmanData*:

1) *HuffmanData.BaseTree.Nodes* содержит (определение см. П.5):
   *HuffmanData.BaseTree.Nodes*[*i*].*Left* = номер предыдущего левого узла;
   *HuffmanData.BaseTree.Nodes*[*i*].*Right* = номер предыдущего правого узла, для узлов включенных в дерево.
2) *HuffmanData.BaseTree.TreeIndexes* содержит (определение см. П.5):
   *HuffmanData.BaseTree.TreeIndexes.Root* = номер корневого узла дерева;
3) *HuffmanData.Codes* — массив для хранения кодов Хаффмана, определен так:

```
tCodes=packed array[tBaseNodeIndex] of tCode;
```

где

```
tBaseNodeIndex=0..255;
tCode=packed record
  Length:tCodeLength; { длина битового кода в битах}
  Code:tCodeData;     { место для хранения битового кода }
end;
```

[9] Рекурсией называют обращение процедуры самой к себе. Непосредственное обращение называют прямой рекурсией, а обращение из другой процедуры (которая, в свою очередь, была вызвана в данной) — косвенной рекурсией.

```
tCodeLength=0..255;
tCodeData=record
  case byte of
    0:(Words:tCodeWords); {последовательность слов кода}
    1:(Bytes:tCodeBytes); {последовательность байт кода}
end;
tCodeWords=array[0..Pred(cMaxCodeLengthInWords)] of word;
tCodeBytes=array[tCodeByteIndex] of byte;
tCodeByteIndex=0..Pred(cMaxCodeLengthInBytes);
cMaxCodeLengthInWords=(cMaxCodeLengthInBits+15) div 16;
cMaxCodeLengthInBytes=(cMaxCodeLengthInBits+7) div 8;
```

В МОМЕНТ ВЫЗОВА содержит:

1) *HuffmanData.Codes*[*i*].*Length* = 0; *i* = [0..255];
2) *HuffmanData.Codes*[*i*].*Code.Bytes*[*j*] = 0; *i* = [0..255], *j* = [0..32].

ВЫХОД из процедуры:

Глобальные данные *HuffmanData*:

1) *HuffmanData.BaseTree.Nodes* — не изменяется.
2) *HuffmanData.BaseTree.TreeIndexes* — не изменяется.
3) *HuffmanData.Codes* содержит:

*HuffmanData.Codes*[*i*].*Length* = длина кода для байта *i*; *i* = [0..255];

*HuffmanData.Codes*[*i*].*Code.Bytes*[*j*] = битовый код для байта *i*; *i* = [0..255], *j* = [0..32].

Вычисление кода символа заключается в записи последовательностью битов пути от корневого узла дерева к символу в исходной таблице. Поворот налево кодируется 0, а направо — 1 (данный выбор кодировки произволен, можно кодировать и наоборот).

Вычисление кодов можно провести двумя путями

1) Используя *HuffmanData.NodesData*[*i*].*NextNode* — прямой проход по дереву. В этом случае для каждого ненулевого *HuffmanData.NodesData*[*i*].*Counter* и *i* = [0..255] процедура должна пройти от узла исходной таблицы до корня дерева, записывая повороты в виде последовательности битов.
2) Используя ссылки *HuffmanData.BaseTree.Nodes*[*i*].*Left* и *.Right* двоичного дерева Хаффмана. В этом случае процедура должна пройти от корневого узла дерева по всем возможным путям, записывая повороты и дойдя до узла исходной таблицы (*i* = [0..255]) записать для него код в виде последовательности битов.

Каждый из указанных способов имеет свои достоинства и недостатки. Например, недостатки прямого прохода по дереву:

1) требует хранения дополнительной ссылки *HuffmanData.NodesData*[*i*].*NextNode* — избыточные данные;
2) При вычислении кода получается обратная последовательность битов от узла к корню (а надо от корня к узлу), что приводит к дополнительным операциям и ухудшает быстродействие.

В свою очередь, рекурсивный проход по дереву имеет следующие недостатки

1) сложнее понять и реализовать,
2) активно использует стек программы, что может привести к переполнению стека малого размера.

Вы можете выбрать и реализовать любой из указанных алгоритмов.

Ниже рассмотрим рекурсивный способ вычисления кодов в виде отдельной процедуры. Все необходимые данные переданы процедуре явно.

{ Вычисляет коды для байтов рекурсивным обходом дерева (от корня к исходным узлам) }

```
procedure DefineCodesR(
          const Nodes:tNodes;
          var   Codes:tCodes;
                Root:tNodeIndex
);
```

{Определим вспомогательную переменную, в ней находится код узла, с которым в данный момент работает программа. Напомним, что формально код Хаффмана можно приписать любому узлу дерева, кроме корневого. }

```
  var TmpCode:tCode;
```

{Определим вспомогательную функцию вычисления кода узлов (*Left* и *Right*) предшествующих узлу *NodeIndex*, с которым в данный момент работает программа. Предполагается, что в *TmpCode* при вызове присутствует вычисленный код для узла *NodeIndex* }

```
  procedure CodesCalculate(NodeIndex:tNodeIndex);
  begin
```

{ на входе в процедуру имеем код для узла с номером *NodeIndex* в переменной *TmpCode* }

```
    if NodeIndex<cBaseNodeCount then begin
      { дошли до одного из узлов исходной таблицы - запомнить код }
      ...операцию напишите сами...
    end else begin
      { иначе, вычислить коды для левого и правого предшествующих узлов }

      { Примечание:
      номер байта кода, с битом которого оперируем }
      (TmpCode.Length div 8)
      {  номер бита в байте которым оперируем }
      (TmpCode.Length mod 8)

      { увеличить длину кода на 1 - входим в вышележащие узлы }
      ...операцию напишите сами...


      { вычисление кода для предыдущего правого узла }
            { установить очередной бит в 1 - правый узел }
             ...операцию напишите сами...
            { рекурсивный вызов - продолжаем для правого узла }
            CodesCalculate(Nodes[NodeIndex].UpperNodes.Right);
```

```
      { вычисление кода для предыдущего левого узла }
            { убрать бит }
             ...операцию напишите сами...
            { рекурсивный вызов - продолжаем для левого узла }
            CodesCalculate(Nodes[NodeIndex].UpperNodes.Left);

      { уменьшить длину кода на 1 - выходим в нижележащие узлы  }
      ...операцию напишите сами...
    end;
  end;
{ Конец вспомогательной функции }

{ Начало основной процедуры DefineCodesR }
begin
  { Обнуление всех кодов }
  FillMemByByte(Codes,SizeOf(Codes), 0);
  { Код для корневого узла: длина = 0; код = 0.}
  TmpCode:= Codes[0];
  { Вызов вспомогательной функции. Вычисление кодов для дерева, начать с корня }
  CodesCalculate(Root);
end;
```

В результате выполнения процедуры в массиве *HuffmanData.Codes* будут находиться коды Хаффмана для бинарного дерева.

### П.7. Кодирование буфера

В файле «Huffman0.pas» процедура «Encode» определена как

```
procedure tHuffman.Encode(const InitialBuf;
                          InitialBufSize:tBufferIndex;
                  var     CompressedBuf;
                          CompressedBufMaxSize:tBufferIndex;
                  var     CompressedSize:tBufferIndex);
```

Для написания процедуры следует уяснить что она получает в качестве входных данных и что она должна выдать в качестве результата.

Процедура имеет следующие аргументы:

1) *InitialBuf* — указатель на начало буфера с исходными данными;
2) *InitialBufSize* — размер данных (байт) в буфер *InitialBuf*;
3) *CompressedBuf* — указатель на начало буфера для сжатых данных;
4) *CompressedBufMaxSize* — максимальный размер данных (байт) в буфере *CompressedBuf*;
5) *CompressedSize* — реальный размер сжатых данных (байт) в буфере *CompressedBuf*.

ВХОД в процедуру:

Процедура имеет следующие аргументы:

1) *InitialBuf* — данные для упаковки;
2) *InitialBufSize* — размер данных для упаковки;
3) *CompressedBuf* — содержимое не определено;
4) *CompressedBufMaxSize* — максимальный возможный размер упакованных данных;
5) *CompressedSize* — не определено.

Глобальные данные *HuffmanData*:

1) *HuffmanData.Codes* — массив кодов Хаффмана (опр. см. пункт П.6):
   В МОМЕНТ ВЫЗОВА содержит:
   *HuffmanData.Codes*[*i*].*Length* = длина кода для байта *i*; *i* = [0..255];
   *HuffmanData.Codes*[*i*].*Code.Bytes*[*j*] = битовый код Хаффмана для байта *i*; *i* = [0..255], *j* = [0..32].

ВЫХОД из процедуры:

Глобальные данные *HuffmanData*:

1) *HuffmanData.Codes* — не изменяется.
2) *CompressedBuf* — заполнен упакованными данными;
3) *CompressedSize* = реальный размер сжатых данных (байт) в буфере *CompressedBuf*.

Ниже рассмотрим простейший способ кодирования, в виде отдельной процедуры, которой явно переданы все, указанные выше, данные.

```
procedure Encode(
          const InBuf;
                Size:tBufferIndex;
          var   OutBuf;
                OutBufMaxSize:tBufferIndex;
          var   OutSize:tBufferIndex;
          var   Codes:tCodes
);
  type
    tPDByte=^tDByte;
  var
    i,j:tBufferIndex;
    BitsShift:byte;
    CodeLength:tCodeByteIndex;
    n:-1..High(CodeLength);
begin
```

{ Очистить выходной буфер (в общем случае это не обязательно, можно чистить по мере необходимости при кодировании)}

```
  ... код напишите сами ...
```

{ Присвоить начальные значения вспомогательным переменным: }

```
  OutSize:=0;      { - размер упакованных данных }
  j:=0;            { - индекс буфера упакованных данных OutBuf }
  BitsShift:=0;    { - сдвиг битов от начала очередного байта в OutBuf
                       или количество занятых битов. При копировании очередного
                     кода он практически всегда оканчивается не на границе байта, а
                     где-то внутри байта, BitsShift — указывает число занятых бит}
```

{ Цикл по всему входному буферу *InBuf* от 0 до (*Size* -1)}

```
for i:=Low(i) to Pred(Size) do begin
```

{ Для очередного символа из входного буфера *tBuffer*(*InBuf*)[*i*] скопировать код *Codes*[ *tBuffer*(*InBuf*)[*i*] ].*Code* длиной *Codes*[ *tBuffer*(*InBuf*)[*i*] ].*Length* бит в выходной буфер *OutBuf*.}

{ Копирование можно провести в два приема:

1) копировать полные байты кода (число таких байт (*Codes*[ *tBuffer*(*InBuf*)[*i*] ].*Length* div 8)), здесь следует помнить, что предыдущий код может оканчиваться не на границе байта:}

```
CodeLength:=Codes[ tBuffer(InBuf)[i] ].Length div 8;
for n:=0 to CodeLength-1 do begin
  Inc( tPDByte(@tBuffer(OutBuf)[j])^,
   (Codes[ tBuffer(InBuf)[i] ].Code.Bytes[n] shl BitsShift) );
  inc(j);
end;
```

{2) копировать остаток битов

(число таких бит (*Codes*[ *tBuffer*(*InBuf*)[*i*] ].*Length* mod 8).}

```
n:=Codes[ tBuffer(InBuf)[i] ].Length mod 8;
if n>0 then begin
  inc(n,BitsShift);
  if n>=cSizeOftByteInBits then begin
    Inc( tPDByte(@tBuffer(OutBuf)[j])^,
        (Codes[ tBuffer(InBuf)[i] ].Code.Bytes[CodeLength] shl BitsShift) );
    inc(j);
    dec(n,cSizeOftByteInBits);
  end else begin
    Inc( tBuffer(OutBuf)[j],
        (Codes[ tBuffer(InBuf)[i] ].Code.Bytes[CodeLength] shl BitsShift) );
  end;
  BitsShift:=n;
end;
  end;
```

{ Конец цикла упаковки. Возвращаем размер упакованных данных }

```
  OutSize:=j;
end;
```

В результате выполнения процедуры в буфере *OutBuf* будут находиться коды Хаффмана для данных из буфера *InBuf* и *OutSize* будет равно размеру упакованных данных в байтах.

### П.8. Декодировка буфера

В файле «Huffman0.pas» процедура «Encode» определена как

```
procedure tHuffman.Decode(const CompressedBuf;
                          var   DecompressedBuf;
                                DecompressedSize:tBufferIndex);
```

Следует уяснить что она получает в качестве входных данных и что она должна выдать в качестве результата. Процедура имеет следующие аргументы:

1) *CompressedBuf* — указатель на начало буфера со сжатыми данными;
2) *DecompressedBuf* — указатель на начало буфера для декодированных данных;
3) *DecompressedSize* — размер НЕСЖАТЫХ данных (байт).

ВХОД в процедуру:

Процедура имеет следующие аргументы:

1) *CompressedBuf* — данные для распаковки;
2) *DecompressedBuf* — содержимое не определено;
3) *DecompressedSize* — размер распакованных данных.

Глобальные данные *HuffmanData.BaseTree* (остальные глобальные данные не определены и не нужны).

В МОМЕНТ ВЫЗОВА *HuffmanData.BaseTree* содержит:

1) *HuffmanData.BaseTree.Nodes*[*i*].*Left* = номер предыдущего левого узла;
   *HuffmanData.BaseTree.Nodes*[*i*].*Right* = номер предыдущего правого узла, для узлов включенных в дерево.
2) *HuffmanData.BaseTree.TreeIndexes.Root* = номер корневого узла дерева.

ВЫХОД из процедуры:

Глобальные данные *HuffmanData*:

1) *HuffmanData.BaseTree* — не изменяется.
2) *CompressedBuf* — не изменяется.
3) *DecompressedBuf* — заполнен распакованными данными.
4) *DecompressedSize* — не изменяется.

Ниже рассмотрим простейший способ кодирования, в виде отдельной процедуры, все данные переданы процедуре явно.

```
procedure DecodeSimple(const InBuf;
                       var   OutBuf;
                             DecodedSize:tBufferIndex;
                       const Nodes:tNodes;
                             Root:tNodeIndex
);
  var
    j,i:tBufferIndex;
    Node:tNodeIndex;
    ByteDecoded:boolean;
    lc:0..8;
    b:byte;

begin
  i:=0;  { - индекс для кодированного буфера }
  lc:=0; { - число недекодированных бит в буферной переменной b}
```

{ Цикл декодирования выходного буфера. }

```
  for j:=0 to Pred(DecodedSize) do begin {j - индекс для декодированного буфера}
```

{ Начинаем проход по дереву с корневого узла. }

```
    Node:=Root;

    repeat
      if lc=0 then begin
```

{ Считываем очередной байт данных из кодированного (входного) буфера в буферную переменную *b*. }

```
end;
```

{ Декодировка проходом по дереву. }

```
      repeat
         Node:=...следующий узел;
        b:=b shr 1; { удаляем декодированный бит из буферной переменной b }
        Dec(lc); { уменьшаем счетчик недекодированных битов
                   в буферной переменной b }
```

```
        { проверяем условие окончания декодировки очередного байта }
          ByteDecoded:=(Node<cBaseNodeCount);
       until (lc=0) or ByteDecoded;

     until ByteDecoded;

   { Заносим декодированный байт (Node) в буфер. }
     tBuffer(OutBuf)[j]:=Node;
  end;
end;
```

В результате выполнения процедуры в буфере *OutBuf* будут находиться декодированные данные для кодированного буфере *InBuf*.

### Заключение

Выше описаны самые простые варианты реализации для основных процедур алгоритма Хаффмана. При выполнении лабораторной работы Вы можете улучшить их (см., например, пункт 3.8.2).

Любые улучшения реализации алгоритма будут учитываться как дополнительная заслуга при сдаче зачета.