МИНИСТЕРСТВО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

УРАЛЬСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

кафедра молекулярной физики

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНОЙ РАБОТЕ

КОМПРЕССИЯ ДАННЫХ ИЛИ ИЗМЕРЕНИЕ И ИЗБЫТОЧНОСТЬ ИНФОРМАЦИИ. ДИНАМИЧЕСКИЙ МЕТОД ХАФФМАНА

ПО КУРСУ «ТЕОРИЯ ИНФОРМАЦИОННЫХ СИСТЕМ»

ДЛЯ СТУДЕНТОВ ДНЕВНОЙ ФОРМЫ ОБУЧЕНИЯ СПЕЦИАЛЬНОСТЕЙ:

07.19.00 - ИНФОРМАЦИОННЫЕ СИСТЕМЫ В ТЕХНИЧЕСКОЙ ФИЗИКЕ

Екатеринбург 2006

УДК 774:002:006.354

Составители: О. Е. Александров.

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 Екатеринбург: УГТУ-УПИ, 2000. 36 с.

Изложена краткая теория динамических методов сжатия информации. Описан динамический вариант 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».

Библиогр. 0 назв. Рис. 6. Табл. 0. Прил. 1.

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

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

1. СТАТИЧЕСКИЕ И ДИНАМИЧЕСКИЕ МЕТОДЫ

Методы сжатия данных без потерь можно разделить на два основных типа:

1) статическое (блочное) сжатие;
2) динамическое (адаптивное или потоковое) сжатие.

Примером первого типа является классический алгоритм Хаффмана [1]. Примером второго типа — алгоритм Лемпеля-Зива [2].

Основное отличие статический и динамических алгоритмов состоит в следующем:

- Динамические алгоритмы обрабатывают поток данных за один проход, т.е., грубо говоря, упаковывают данные по-байтно и результат упаковки очередного байта данных, поступившего на вход алгоритма, сразу же доступены на выходе, не зависимо от наличия/отсутствия следующего байта.

- Статические алгоритмы обрабатывают данные порциями (блоками), причем обработка блока требует, как минимум, двух проходов. На первом проходе данные анализируются, а на втором преобразуются в упакованный вид.

Второе отличие:

- Динамический алгоритм не требует записи специальной вспомогательной информации для распаковки и распаковка также осуществляется потоком, т.е. по-байтно.
- Статический алгоритм требует записи специальной вспомогательной информации для каждого упакованного блока данных и упаковка/распаковка осуществляется по-блочно (порциями).

В результате статические методы неприменимы там, где поток данных невозможно прервать для обработки порции данных, например, при передаче данных по медленной линии (модему). В этом случае задержка на упаковку блока вызывает простой линии, кроме того требуется располагать достаточно большим буфером на обоих концах линии для хранения блока данных.

Однако деление на статические/динамические методы не абсолютно. Практически любой статический метод можно преобразовать в динамический по следующему алгоритму:

1) Вспомогательная информация для упаковки/распаковки инициализируется наперед заданным образом.
2) Очередной байт данных преобразуется в соответствии с текущим состоянием вспомогательной информации.
3) Вспомогательная информация для упаковки/распаковки изменяется в соответствии с обработанным байтом.
4) Повторяется шаг 2) пока данные не закончились.

Так можно преобразовать практически любой динамический алгоритм в статический. Почему же вводят деление? Причина проста — при изменении способа обработки данных алгоритм может терять либо в скорости обработки, либо в степени упаковки. Если оптимальное соотношение скорость обработки/степень упаковки достигается в статическом варианте, то такой алгоритм считают статическим, иначе — динамическим.

Далее будет рассмотрен динамический вариант алгоритма Хаффмана и проанализированы его недостатки.

2. АДАПТИВНЫЙ МЕТОД ХАФФМАНА

2.2. Проблемы реализации динамического метода

Данные для электронной обработки представляют собой некоторую последовательность чисел. Каждое число в этой последовательности обычно представлено битовой цепочкой фиксированной длины. Например, если рассматривать данные как последовательность байтов (8-и битных кусочков), то битовая цепочка[1] 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[2], 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.

Главная и, в сущности единственная, проблема при реализации метода Хаффмана в динамическом варианте — большие затраты времени на построение кодов. Вычисление кода в методе Хаффмана реализовано через построение дерева (см. [1]).

В статическом варианте коды вычисляются один раз на блок данных, в динамическом варианте — вычисление кода происходит каждый раз при поступлении очередного байта данных. При прямой реализации динамического варианта (без изменения алгоритма построения дерева, см. [1]) это ведет к существенному проигрышу в скорости обработки — скорость падает в десятки, сотни или тысячи раз — конкретное число фактически равно размеру буфера статического метода. В этом случае необходим более эффективный алгоритм вычисления кода.

Можно конечно решить эту задачу «просто» — вычислять код не каждый раз при поступлении очередного байта данных, а только раз на N байт, где N сравнимо с размером буфера статического метода. Недостаток такого подхода — кодирование информации текущего буфера происходит на основе предыдущих

---

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

[2] 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.

статистических данных, которые могут сильно отличаться по частотным характеристикам и, следовательно, неизбежно ухудшение степени сжатия. Если уменьшать *N*, то вновь возникнет проблема снижения скорости обработки.

2.4. Алгоритм быстрого перестроения дерева Хаффмана

Алгоритм динамического метода Хаффмана был предложен независимо Фоллером [1973] и Галлагером [1978]. В 1985 Дональд Кнут разработал усовершенствованный вариант алгоритма, получивший название FGK алгоритм (Faller, Gallager, Knuth).

Основная иделя быстрого алгоритма перестроения дерева содержится в самом слове «перестроение» — при изменении счетчика одного байта в дереве на единицу дерево не должно претерпеть существенных изменений, т.е. если не строить дерево каждый раз заново, а лишь перестраивать уже имеющееся дерево, то можно рассчитывать на увеличение скорости.

Алгоритм перестроения базируется на возможности представить дерево Хаффмана (см. [1]) в виде упорядоченного по возрастанию частоты массива узлов. На рис. 2.1 представлено такое дерево для 6 базовых узлов 0, 1, … 5. Обычно используют байт (0..255 или 256 базовых узлов), но для примера годится и 6 узлов. Далее будем обозначать число базовых узлов как *N*, а значение для базового узла как *b*. Полный размер массива узлов для дерева Хаффмана равен $2N - 1$. При этом все пары узлов, объединяемые в новый узел при построении дерева будут находиться в таком массиве рядом. Левый узел пары будет иметь четный номер, а правый — нечетный, если нумерация узлов в массиве начинается с нуля.

Такое дерево можно представить тремя таблицами как это показано на рис. 2.2. Таблица а) обеспечивает связность от корня дерева к листьям. Таблица б) обеспечивает связность от листьев к корню. Таблица в) обеспечивает доступ к листьям — базовым узлам.

Табличное представление можно усовершенствовать. Прежде всего видно, что в таблице б) рис. 2.2 для четного и нечетного номера ссылки повторяются. Далее в таблице а) рис. 2.2 не используется ссылка на предыдущий узел для базовых узлов таблицы.

Поэтому можно сократить число таблиц до двух, объединив таблицы б) и в) в одну таблицу ссылок на «следующий узел», используя первые $N - 1$ ячеек как таблицу б), а следующие *N* ячеек как таблицу в). Ссылка на «следующий узел» для узлов *i* и $i+1$ хранится в одной ячейке $[i/2]$, где *i* - четное. Ссылку на базовый узел для значения *b* можно определить как значение в ячейке $b + N - 1$. Исключить из таблицы а) строку «Знач. *b*» и заменить пустующие ссылки на

«предыдущий узел» для базовых узлов на значение $b + N - 1$. Значения ссылок p на «предыдущий узел» для прочих узлов — заменим на $[p/2]$.

ДЕРЕВО УЗЛОВ В МЕТОДЕ ХАФФМАНА

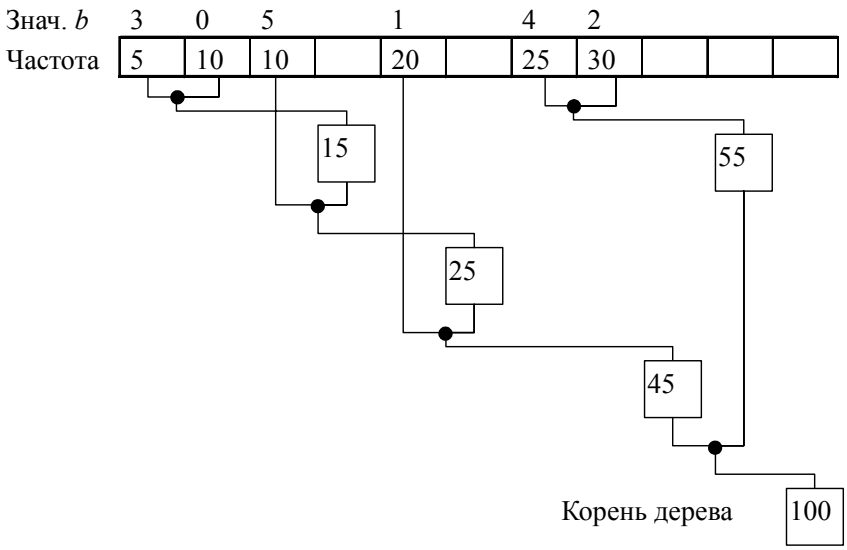


Рис. 2.2

ТАБЛИЧНОЕ ПРЕДСТАВЛЕНИЕ ДЕРЕВА ХАФФМАНА

а) Массив узлов дерева

Номер	0	1	2	3	4	5	6	7	8	9	10
Знач. *b*	3	0	5		1		4	2			
Частота	5	10	10	15	20	25	25	30	45	55	100
Ссылка на пред.				0		2			4	6	8

б) Ссылки на следующий узел

Номер	0	1	2	3	4	5	6	7	8	9	10
Ссылка на след.	3	3	5	5	8	8	9	9	10	10	

в) Ссылки на базовые узлы

Число	0	1	2	3	4	5
Ссылка	1	4	7	0	6	2

Табличное представление соответствует дереву на рис. 2.2. Все ссылки = номеру узла в таблице а).

Рис. 2.4

Усовершенствованное табличное представление дерева Хаффмана

1) Массив узлов дерева

Номер	0	1	2	3	4	5	6	7	8	9	10
Частота	5	10	10	15	20	25	25	30	45	55	100
Ссылка на пред.	5+3	5+0	5+5	0 = [0/2]	5+1	1 = [2/2]	5+4	5+2	2 = [4/2]	3 = [6/2]	4 = [8/2]

2) Ссылки на следующий узел

Номер	0	1	2	3	4	5	6	7	8	9	10
	Табл. б) — ссылки на следующий узел для узла [номер/2]					Табл. в) — ссылки на базовые узлы					
Ссылка	3	5	8	9	10	1	4	7	0	6	2

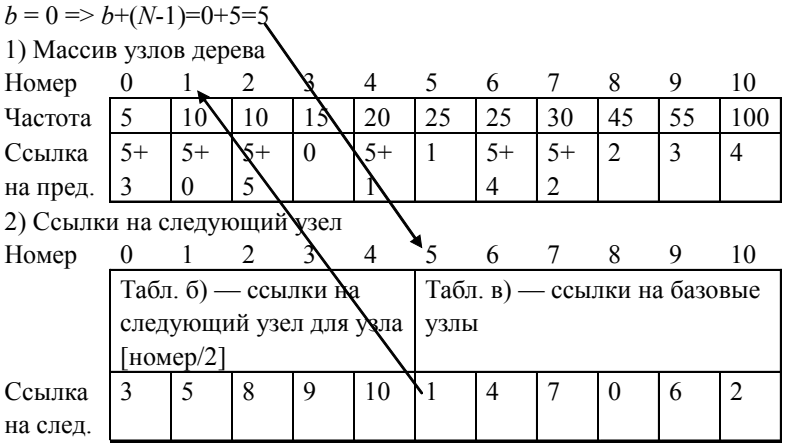
Табличное представление соответствует дереву на рис. 2.2. В табл. 1) «Ссылки на пред.» соответствуют номеру ячейки в табл. 2), а «ссылка» в табл. 2) номеру ячейки в табл. 1).

Рис. 2.6

Преимущества такого представления дерева будут ясны после рассмотрения алгоритма перестроения.

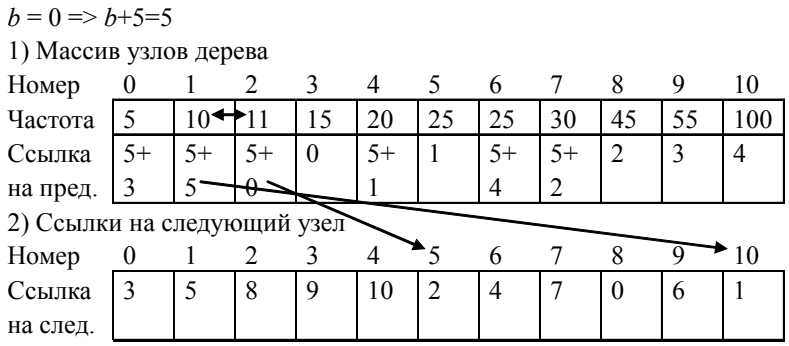
Рассмотрим теперь процесс перестроения дерева, изображенного на рис. 2.2 и соответствующего ему табличного представления рис. 2.6. Пусть на вход поступает очередное значение $b = 0$.

1. По таблице 2) находим в таблице 1) узел соответствующий $b = 0$ — это №1

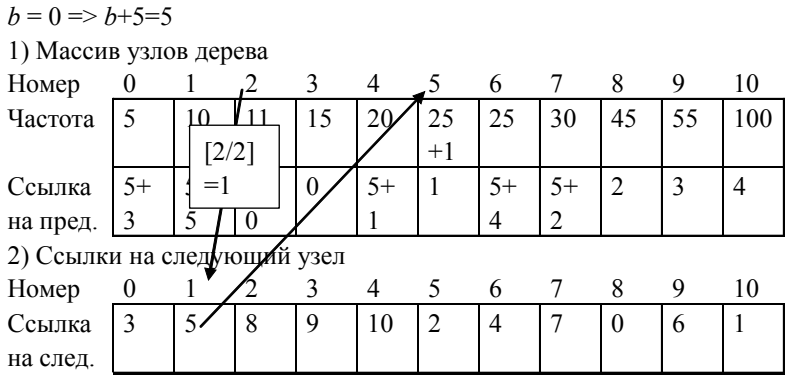


2. Увеличиваем частоту узла №1 на 1

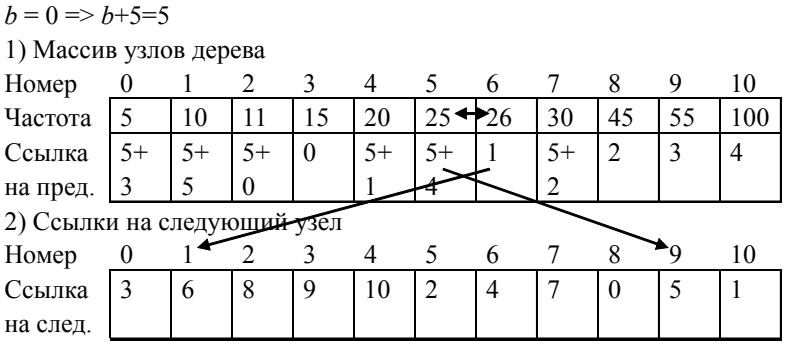
3. Сравниваем частоту в узле №1 (11) с частотой в следующем узле №2 (10) — 11>10 — упорядоченность нарушена. Ищем ближайший следующий узел для которого частота ≥11 — это узел №3.
4. Обмениваем узлы №1 и №2 чтобы восстановить упорядоченность массива узлов и поправляем ссылки в таблице 2) используя «ссылку на пред.».



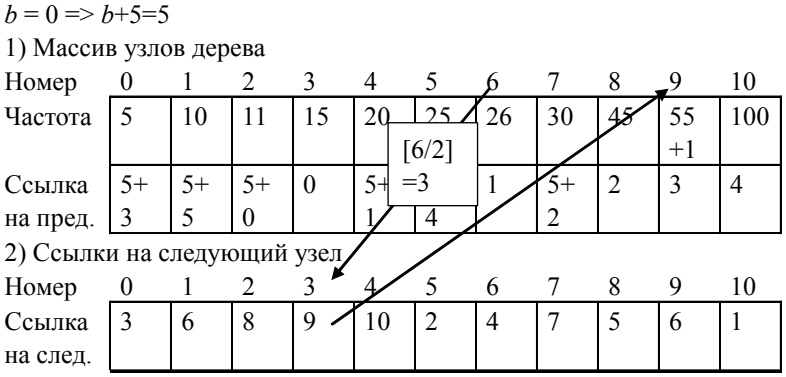
5. Для узла №2 (бывший №1) находим следующий узел — [2/2]=1 — по таблице 2) находим №5 и увеличиваем его частоту на 1



6. Сравниваем частоту в узле №5 (26) с частотой в следующем узле (25) — 26>25 — упорядоченность нарушена.
7. Обмениваем узлы №5 и №6 чтобы восстановить упорядоченность массива узлов и поправляем ссылки в таблице 2) используя «ссылку на пред.».



8. Для узла №6 находим следующий узел — [6/2]=3 — по таблице 2) находим №9 и увеличиваем его частоту на 1



9. Сравниваем частоту в узле №9 (56) с частотой в следующем узле (100) — 56≤100 — упорядоченность не нарушена.
10. Для узла №9 находим следующий узел — [9/2]=4 — по таблице 2) находим №10 и увеличиваем его частоту на 1

11. Узел №10 — это корень дерева, поэтому заканчиваем перестроение.

Легко заметить, что шаги со 2 по 10 можно объединить в цикл. Результат перестроения в виде дерева показан на рис. 2.4.

Перестроенное дерево Хаффмана

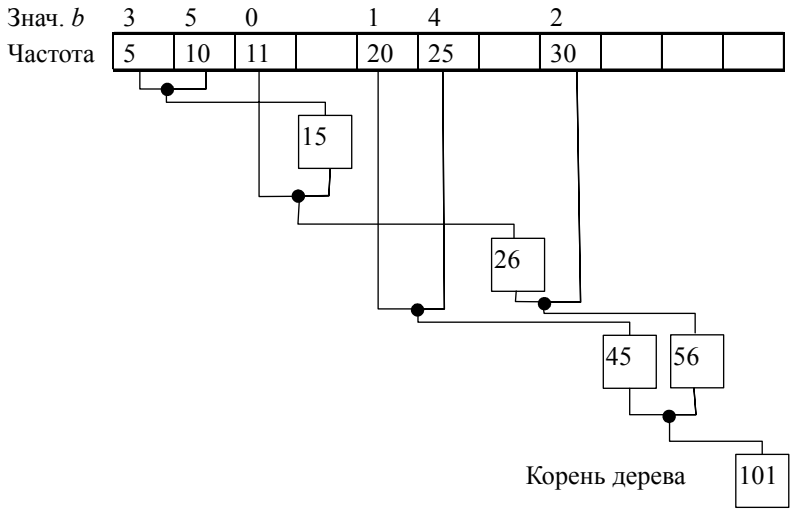


Рис. 2.8

Описанный выше алгоритм перестроения дерева достаточно компактен и обеспечивает максимальное (из известных) быстродействие.

Существуют несколько вариаций этого алгоритма, связанные с началом построения дерева. Выше рассмотрено уже готовое дерево, включающее в себя ВСЕ возможные значения b. Для того чтобы построить такое дерево нужно, как минимум, каждому возможному значению b приписать ненулевую частоту. Т.е. даже отсутствующим в последовательности байтам приходится приписывать частоту 1 — это, теоретически, ухудшает степень сжатия, т.к. увеличивает длину кода.

В реальности может иметь место случай, когда не все возможные значения b присутствуют в кодируемой последовательности. Более того в начальном участке кодируемой последовательности длиной до b_{max} обязательно присутствуют не все возможные значения b. Существует вариант построения динамического дерева Хаффмана «на ходу» с динамическим изменением количества базовых узлов от 0 до b_{max}. В этом варианте добавляют специальный узел b_0 с частотой 0 и первоначально строят дерево из одного узла b_0. При поступлении на вход значения b отсутствующего в дереве в выходную последовательность выводится код для b_0 и полное значение (8 бит) нового b. После чего дерево перестраивается, в него включается новый узел b с частотой 1. При поступлении на вход присутствующего в дереве значения b, дерево перестраивается по описанному выше алгоритму.

Далее используется первый способ построения дерева, когда всем возможным значениям b присваивается частота 1, строится начальное дерево и да-

лее это дерево перестраивается по мере поступления очередных значений *b* из кодируемой последовательности.

2.6. Кодирование длинных последовательностей

При кодировании длинной последовательности возникает проблема: частотные характеристики на основе которых построено дерево характеризуют данные «с начала» последовательности, т.е. могут не соответствовать текущим частотным характеристикам данных — это может ухудшать характеристики сжатия. Требуется ограничивать «память» дерева, отсекая старые данные и забывая их частотные характеристики.

Другая проблема: переполнение счетчиков, при кодировании сверхдлинных последовательностей — в любом случае, рано или поздно, счетчики будут заполнены и потребуется их очистка.

Решить эти две проблемы можно одновременно. Для этого периодически, например, по достижении некоторого значения полного числа обработанных байтов дерево реинициализируется. Но полностью реинициализировать дерево неразумно — сведения о частотных характеристиках будут утрачены совсем. Разумнее периодически уменьшать, например, в два раза все частоты в дереве. Так как некоторые частоты могут быть равны 1, то уменьшение их в два раза даст 0, поэтому при уменьшении частот в два раза, одновременно, все частоты увеличиваются на 1. Последнее гарантирует, что частота для любого *b* будет больше или равна 1.

2.8. Вычисление кода по дереву

Кроме перестроения дерева, для очередного байта надо вычислить код. Код в динамическом алгоритме Хаффмана вычисляется ДО перестроения дерева. Вычисление кода по дереву Хаффмана можно посмотреть в [1]. Однако в отличие от статического алгоритма Хаффмана, где код по дереву можно вычислить однократно, а затем многократно использовать при кодировании буфера, в динамическом варианте код байта приходится вычислять каждый раз, ибо код байта изменяется при перестроении дерева.

Табличное представление дерева (см. рис. 2.6) позволяет достаточно быстро вычислять код Хаффмана для байта. Рассмотрим этот алгоритм на примере вычисления кода для $b = 0$ по таблице, соответствующей дереву рис. 2.2

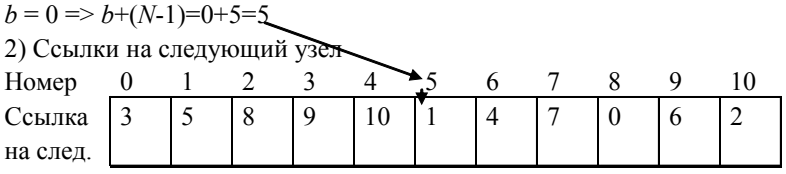
1) Массив узлов дерева

Номер	0	1	2	3	4	5	6	7	8	9	10
Частота	5	10	10	15	20	25	25	30	45	55	100
Ссылка на пред.	5+3	5+0	5+5	0	5+1	1	5+4	5+2	2	3	4

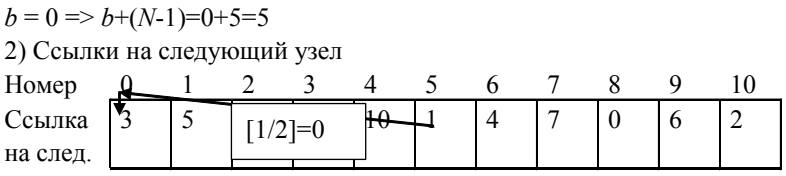
2) Ссылки на следующий узел

Номер	0	1	2	3	4	5	6	7	8	9	10
	Табл. б) — ссылки на следующий узел для узла [номер/2]					Табл. в) — ссылки на базовые узлы					
Ссылка на след.	3	5	8	9	10	1	4	7	0	6	2

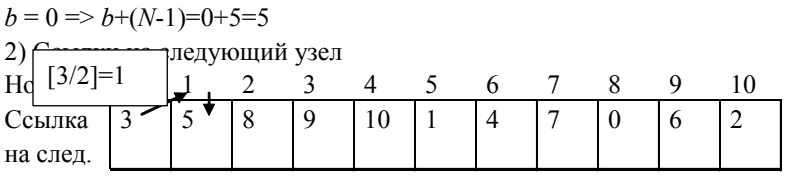
1. На вход поступает новый байт.
2. По таблице 2) находим узел соответствующий $b = 0$ — это №1.



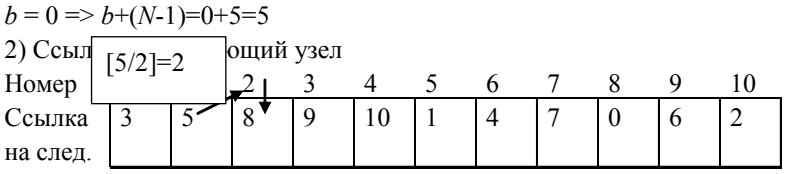
3. №1 — нечетный номер, следовательно последний бит кода =1, записываем в код c = 1b.
4. По таблице 2) находим следующий узел для №1 — это №3.



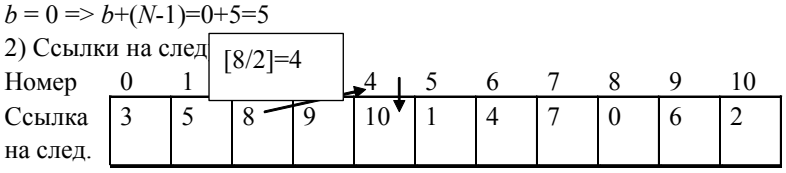
5. №3 — нечетный номер, следовательно последний бит кода 1, записываем (слева) в c = 11b.
6. По таблице 2) находим следующий узел для №3 — это №5.



7. №5 — нечетный номер, следовательно последний бит кода 1, записываем (слева) в c = 111b.
8. По таблице 2) находим следующий узел для №5 — это №8.



9. №8 — четный номер, следовательно последний бит кода =0, записываем (слева) в c = 1110b.
10. По таблице 2) находим следующий узел для №8 — это №10.



11. №10 — корень дерева — заканчиваем вычисление кода. Окончательно код для b = 0: c = 1110b. Код можно проверить по дереву рис. 2.2.
12. Перестраиваем дерево для b = 0.
13. Если на входе еще есть байты — переходми к шагу 1.

Очевидно, что шаги 3-4; 5-6; 7-8 можно объединить в цикл.

2.10. Декодирование кода по дереву

Код в динамическом алгоритме Хаффмана декодируется ДО перестроения дерева, а затем дерево перестраивается для полученного при декодировании байта. Вычисление кода по дереву Хаффмана можно посмотреть в [1]. Здесь мы рассмотрим это процесс применительно к новой структуре храения дерева.

Табличное представление дерева (см. рис. 2.6) позволяет достаточно быстро декодировать код Хаффмана в соответствующий байт. Рассмотрим этот алгоритм на примере декодирования кода c = 1110b по таблице, соответствующей дереву рис. 2.2:

1) Массив узлов дерева

Номер	0	1	2	3	4	5	6	7	8	9	10
Частота	5	10	10	15	20	25	25	30	45	55	100
Ссылка на пред.	5+3	5+0	5+5	0	5+1	1	5+4	5+2	2	3	4

2) Ссылки на следующий узел

Номер	0	1	2	3	4	5	6	7	8	9	10
	Табл. б) — ссылки на следующий узел для узла [номер/2]					Табл. в) — ссылки на базовые узлы					
Ссылка на след.	3	5	8	9	10	1	4	7	0	6	2

1. Устанавливаем начальный узел №10 (корень дерева).
2. Биты кода считываются от младших к старшим. На вход поступает бит 0.

3. По таблице 1) находим предыдущую пару узлов для №10 — это (№8, №9). Поскольку бит 0, выбираем из пары четный узел — №8.

1) Массив узлов дерева

Номер	0	1	2	3	4	5	6	7	8	9	10
Частота	5	10	10	15	20	25	25	30			100
Ссылка	5+	5+	5+	0	5+	1	5+	5+			4
на пред.	3	0	5		1		4	2			

[4x2]=8

4. На вход поступает бит 1.
5. По таблице 1) находим предыдущую пару узлов для №8 — это (№4, №5). Поскольку бит 1, выбираем из пары нечетный узел — №5.

1) Массив узлов дерева

Номер	0	1	2	3	4	5	6	7	8	9	10
Частота	5	10	10	15	20			30	45	55	100
Ссылка	5+	5+	5+	0	5+			5+	2	3	4
на пред.	3	0	5		1		4	2			

[2x2]=4

6. На вход поступает бит 1.
7. По таблице 1) находим предыдущую пару узлов для №5 — это (№2, №3). Поскольку бит 1, выбираем из пары нечетный узел — №3.

1) Массив узлов дерева

Номер	0	1	2	3	4	5	6	7	8	9	10
Частота	5	10			20	25	25	30	45	55	100
Ссылка	5+	5+			5+	1	5+	5+	2	3	4
на пред.	3	0	5		1		4	2			

[1x2]=2

8. На вход поступает бит 1.
9. По таблице 1) находим предыдущую пару узлов для №3 — это (№0, №1). Поскольку бит 1, выбираем из пары нечетный узел — №1.

1) Массив узлов дерева

Номер	0	1	2	3	4	5	6	7	8	9	10
Частота	5	10	10	15	20	25	25	30	45	55	100
Ссылка	5+	5+	5+	0	5+	1	5+	5+	2	3	4
на пред.	3	0	5		1		4	2			

[0x2]=0

10. По таблице 1) находим предыдущую пару узлов для №1. Поскольку значение ссылки на предыдущую пару больше или равно 5, следовательно, это ссылка на исходный байт. Находим $b = 5 - 5 = 0$.
11. Перестраиваем дерево для $b = 0$.
12. Если на входе еще есть биты — переходим к шагу 1.

3. ВОЗМОЖНАЯ РЕАЛИЗАЦИЯ ДИНАМИЧЕСКОГО МЕТОДА ХАФФМАНА

3.2. 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 три процедуры:

1) *ЗакодироватьБайт*.
2) *ДекодироватьКод.*
3) *ПерестроитьДерево.*

Первая процедура должна осуществлять кодирование одного байта, вторая — декодирование код Хаффмана и третья — перестроение дерева. Кодирование (декодирование) последовательности осуществляется последовательным применением процедур *ЗакодироватьБайт, ПерестроитьДерево* (*ДекодироватьКод*, *ПерестроитьДерево*) ко всем байтам последовательности по-очереди.

Потом, следует определиться с аргументами (данными) над которыми будут проводить операции эти процедуры:

1) *ЗакодироватьБайт*(*Байт*; var *Код*; *ДанныеДеревьяХаффмана*).
2) *ДекодироватьКод*(var *Код*; var *Байт*; *ДанныеДеревьяХаффмана*).
3) *ПерестроитьДерево*(*Байт*; *ДанныеДеревьяХаффмана*).

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

3.4. Структуры данных для динамического 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 динамического метода Хаффмана. Разумно воспользоваться представлением дерева в виде рис. 2.6 — это обеспечит максимальное быстродействие. Основные данные лучше размес-

тить в виде двух массивов. Поскольку размер данных невелик — можно ограничиться статическим выделением памяти под данные.

Полный набор данных для динамического дерева Хаффмана будет выглядеть так (определения типов и данных см. в файле «DHufType.pas» и «HuffType.pas»):

```
{ Полный набор данных динамического дерева Хаффмана }
tDynHuffmanFullTreeData=record
      { данные узлов }
      Nodes:tNodesData;
      { ссылки узлов }
      Refs:tReferencies;
      { общая сумма счетчиков для контроля сброса }
      MaxTotalCounter:tFriquency;
      { Информационные данные }
      { число сбросов счетчиков }
      TotalHalfCounterCount:tVeryLongCounter;
      { число байтов в исходном потоке (файле)}
      TotalByteCount:tVeryLongCounter;
      { число битов в упакованном потоке (файле )}
      TotalBitCount:tVeryLongBitCounter;
end;
```

1) *Nodes*:*tNodesData* — соответствует таблице 1) рис. 2.6;
2) *Refs*:*tReferencies* — соответствует таблице 2) рис. 2.6.
3) *MaxTotalCounter*:*tFriquency* — предельное значение суммы счетчиков, при достижении которого происходит уменьшение всех счетчиков в два раза.
4) Остальное — вспомогательные данные для подсчета статистических данных.

Массивы *Nodes* и *Refs* организованы так:

```
tNodesData=packed array[tIndex] of tNodeData;
```

где

```
{ Данные узла динамического дерева Хаффмана }
tNodeData=packed record
      { счетчик узла }
      Counter:tFriquency;
      { ссылка на предыдущие узлы }
      UpperNode:tHalfIndexEx;
end;

tHalfIndexEx=Low(tHalfIndex)..High(tHalfIndex)+cBaseNodeCount;

tHalfIndex=Low(tIndex)..(High(tIndex) div 2);

tIndex=0..(cMaxNodeCount-1);

cBaseNodeCount=High(tByte)+1;

tByte=byte;
```

и

```
{ ссылочные данные дерева }
tReferencies=packed record
      case byte of
      { Объединенный массив ссылок }
      0:(   NextNode:tNextNodesEx);

      1:(   { ссылка на узел, который образуется из узла i и i+1 }
            NextNode0:tNextNodes;
            { ссылка на базовые узлы [0..255] }
            NodeRef: tNodeRef
      );
```

```
end;
```

где

```
tNextNodesEx=packed array[tHalfIndexEx] of tIndex;
tNextNodes=packed array[tHalfIndex] of tIndex;
tNodeRef=packed array[tBaseNodeIndex] of tIndex;
tBaseNodeIndex=0..(cBaseNodeCount-1);
```

Прочие типы данных см. в файле «DHufType.pas» и «HuffType.pas».

3.6. Реализация алгоритмов для динамического метода Хаффмана

Поскольку основные процедуры динамического метода Хаффмана очень и очень невелики, рассмотрим их подробнее. Ниже приведена реализация алгоритмов, описанных в пунктах 2.4÷2.10 для данных, описанных в пункте 3.4.

3.6.2. Инициализация дерева

Дерево заполняется исходными данными однократно при начале кодирования. Предполагается, что все байты равновероятны, т.е. частота для каждого значения байта равна единице, см. пункт 2.4.

Аргументы:

- *aTreeData* — неинициализированные данные дерева Хаффмана;
- *aMaxTotalCounter* — значение для порога уменьшения счетчиков.

Возвращает:

- *aTreeData* — инициализированное дерево.

```
procedure dhInitTree(
      aMaxTotalCounter:tFriquency; { значение для порога уменьшения счетчиков }
var   aTreeData:tDynHuffmanFullTreeData
);
var b:tByte; i,ii:tIndex;
begin { построение исходного дерева }
      with aTreeData do begin
            if aMaxTotalCounter=0 then aMaxTotalCounter:=cMaxTotalCounter;
            { полное число сосчитанных байт после которого проводится
            ополовинивание счетчиков }
            MaxTotalCounter:=aMaxTotalCounter;

            { заполняем базовую часть дерева }
            for b:=Low(b) to High(b) do begin
                  Refs.NodeRef[b]:=b;
                  with Nodes[b] do begin
                        Counter:=1;
                        UpperNode:=b+cBaseNodeCount;
                  end;
            end;

            { строим остаток дерева }
            ii:=Low(b);
            for i:=Succ(High(b)) to High(i) do begin
                  with Nodes[i] do begin
                        UpperNode:=ii shr 1;
                        Counter:=Nodes[ii].Counter+Nodes[ii+1].Counter;
                  end;
                  { устанавливаем признак следующий узел }
                  Refs.NextNode[ii shr 1]:=i;
```

```
                Inc(ii,2);
            end;

            { обнуляем счетчики }
            ZeroingVLC(TotalHalfCounterCount);
            ZeroingVLC(TotalByteCount);
            with TotalBitCount do begin
                BitCount:=0;
                ZeroingVLC(ByteCount);
            end;
        end;
end;
```

3.6.4. Перестроение дерева

Перестраивает дерево в соответствии с изменением счетчика байта *aByte* на единицу, см. пункт 2.4.

Аргументы:

- *aByte* — значение очередного байта;
- *aTreeData* - данные дерева Хаффмана.

Возвращает:

- *aTreeData* — перестроенное дерево.

```
procedure dhReBuildTree(
            aByte:tByte;
        var   aTreeData:tDynHuffmanFullTreeData
);
        var
        c     :tFriquency;
        i,i0  :tIndex;
        node  :tNodeData;

begin
        with aTreeData do begin
            { если достигнут предел для значения счетчиков }
            if Nodes[cRootNode].Counter>=MaxTotalCounter then begin
                { ополовиниваем счетчики }
                dhHalfCounters(aTreeData);
            end;

            { увеличиваем полное число байт }
            Inc(Nodes[cRootNode].Counter);
            { получаем ссылку на позицию байта aByte
            в упорядоченном списке-дереве }
            i0:=Refs.NodeRef[aByte];

            repeat { пока не уткнемся в корень }
                { увеличиваем счетчик узла }
                Inc(Nodes[i0].Counter);
                { запоминаем счетчик }
                c:=Nodes[i0].Counter;
                { сравниваем со следующим в упорядоченном списке }
                if c>Nodes[Succ(i0)].Counter then begin
                    { если упорядоченность нарушена —
                    ищем ближайший следующий счетчик,
                    больший или равный данному }
                    i:=i0;
                    repeat
                        Inc(i0);
                    until c<=Nodes[Succ(i0)].Counter;

                    { обмениваем узлы }
                    node:=Nodes[i0];
                    Nodes[i0]:=Nodes[i];
```

```
                        Nodes[i]:=node;

                        { обновляем ссылки для узлов-предков }
                        Refs.NextNode[node.UpperNode]:=i;

                        { обновляем ссылки для узлов-предков }
                        Refs.NextNode[Nodes[i0].UpperNode]:=i0;
                  end;
                  i0:=Refs.NextNode[i0 shr 1];
            until i0=cRootNode; { пока не уткнемся в корень }

            { увеличиваем счетчик байтов пока не уткнемся в корень }
            IncVLC(TotalByteCount,1);

      end;
end;
```

3.6.6. Кодирование байта

Вычисляет код для байта *aByte* (см. пункт 2.8), перестраивает дерево (см. пункт 3.6.4) в соответствии с изменением счетчика для байта *aByte.*

Аргументы:

- *aByte* - значение очередного байта для кодирования;
- *aCode* - переменная для возврата кода Хаффмана, в переменной может находиться остаток бит (неполный байт), длина остатка в *aCode.Length* не более 7 бит.
- *aTreeData* - данные дерева Хаффмана.

Возвращает:

- *aCode* - остаток кода+код Хаффмана для *aByte*.
- *aTreeData* - перестроенное дерево.

```
procedure dhEncodeChar(
            aByte:tByte;
      var   aCode:tCodeEx;
      var   aTreeData:tDynHuffmanFullTreeData
);
var
      bc,bc1:tCodeByteIndexEx;
      n:tIndex;
      c:0..7;
      ol,nl:tCodeLength;
      pBytes:^tCodeBytesEx;
type
      pW=^Word;

begin
 { Вычисление кода для байта aByte }
 with aTreeData do begin
      pBytes:=@aCode.Code.Bytes;
      { получаем ссылку на узел для aByte }
      n:=Refs.NodeRef[aByte];

      { Вычисляем код (но биты заносятся начиная со старшего бита) }
      bc:=High(bc);
      repeat
            for c:=7 downto 0 do begin
                  { Проверяем является ли данный узел ПРАВЫМ предыдущим
                  узлом для следующего, если ДА, то устанавливаем в битовой
                  цепочке 1.}
                  pBytes^[bc]:=(pBytes^[bc] shl 1) or (n and 1);
                  { получаем ссылку на следующий узел в дереве. }
```

```
                  n:=Refs.NextNode[n shr 1];
                  { повторяем, пока не дойдем до корневого узла дерева. }
                  if (n=cRootNode) then begin
                        pBytes^[bc]:=pBytes^[bc] shl c;
                        break;
                  end;
            end;
            Dec(bc);
            { повторяем, пока не дойдем до корневого узла дерева. }
      until (n=cRootNode);

      { Сдвигаем вычисленный код }
      ol:=aCode.Length; {длина остатка кода в aCode.Code}
      bc1:=(High(bc)-bc); {число полных и неполных байтов нового кода}
      nl:=bc1*8-c; {число бит нового кода}
      IncVLBC(TotalBitCount, nl); { увеличиваем счетчик бит }
      Inc(aCode.Length,nl); {полная длина кода в битах Остаток+Новый}
      Inc(bc);
      if c=ol then begin
            if ol=0 then
                  Move(pBytes^[bc],pBytes^[Low(bc1)], bc1+1)
            else begin
                  Inc(pBytes^[Low(bc1)], pBytes^[bc]);
                  Move(pBytes^[bc+1], pBytes^[Low(bc1)+1], bc1);
            end;
      end else begin
            if c>ol then begin
                  bc1:=Low(bc1);
                  if ol>0 then begin
                        c:=c-ol;
                        Inc(pBytes^[Low(bc1)],Lo((pW(@pBytes^[bc])^ shr c)));
                        Inc(bc); Inc(bc1);
                  end;
            end else begin
                  c:=ol-c;
                  Inc(pBytes^[Low(bc1)],(pBytes^[bc] shl c));
                  c:=8-c;
                  bc1:=Low(bc1)+1;
            end;
            for bc:=bc to High(bc) do begin
                  pBytes^[bc1]:=Lo(pW(@pBytes^[bc])^ shr c);
                  Inc(bc1);
            end;
      end;

      { Перестраиваем дерево }
      dhReBuildTree(aByte, aTreeData);
 end;
end;
```

3.6.8. Декодирование кода

Простая декодировка проходом по дереву. Декодирование *aCodeBitCount* битов из *aCodeBits* в номер узла *aNode*, начиная с *aNode* (см. пункт 2.10). После декодировки обновляется дерево (см. пункт 3.6.4).

Аргументы:

- *aCodeBits* - биты для декодирования;
- *aCodeBitCount* - число бит в *aCodeBits*;
- *aNode* - индекс узла для начала декодирования;
- *aTreeData* - данные дерева Хаффмана.

Возвращает:

- *TRUE* - декодировано в байт, иначе *FALSE* - декодировано до промежуточного узла, так как закончились биты в *aCodeBits*.
- *aCodeBits* - остаток недекодированных бит сдвинутый в младшие биты;
- *aCodeBitCount* - число недекодированных бит в *aCodeBits*;
- *aNode* - индекс узла (если *TRUE* = декодированный байт);
- *aTreeData* - перестроенное дерево.

```
function dhDecodeBits(
      var aCodeBits:tCodeMaxPortion;
      var aCodeBitCount:tCodeMaxPortionLength;
      var aNode:tNodeIndex;
      var aTreeData:tDynHuffmanFullTreeData
):boolean;

var
   Node:tNodeIndex;
   CodeBits:tCodeMaxPortion;
   CodeBitCount:tCodeMaxPortionLength;
begin
 dhDecodeBits:=FALSE;

      { инициируем локальные переменные }
      CodeBitCount:=aCodeBitCount;
      if CodeBitCount=0 then begin
            Exit;
      end;
      CodeBits:=aCodeBits;
      Node:=aNode;

      { Декодировка проходом по дереву. }
      with aTreeData do begin
            repeat
                  { ссылка на предыдущий узел }
                  Node:=Nodes[Node].UpperNode;
                  { если байт декодирован }
                  if (Node>=cBaseNodeCount) then begin
                        dhDecodeBits:=TRUE;
                        { вычисляем код байта и возвращаем его }
                        Dec(Node,cBaseNodeCount);
                        aNode:=Node;
                        { обновляем дерево }
                        dhReBuildTree(Node, aTreeData);
                        { возвращаем остаток бит }
                        aCodeBits:=CodeBits;
                        aCodeBitCount:=CodeBitCount;
                        { выходим из процедуры }
                        Exit;
                  end;
                  { предыдущий узел }
                  Node:=(Node shl 1) + (CodeBits and 1);
                  { уменьшаем счетчик недекодированных бит }
                  Dec(CodeBitCount);
                  { удаляем декодированный бит }
                  CodeBits:=CodeBits shr 1;
            until (CodeBitCount=0);
      end;

      { Запоминаем декодированную часть. }
      aNode:=Node;
      { все биты декодированы }
      aCodeBitCount:=0;
end;
```

3.6.10. Сброс счетчиков при переполнении

Уменьшает счетчики для БАЗОВЫХ узлов дерева в два раза (см. пункт 2.6) и перестраивает дерево. Это самая длинная процедура алгоритма из рассмотренных здесь. Демонстрирует алгоритм быстрого полного построения дерева для данных в формате рис. 2.6 (см. также пункт 3.4).

Аргументы:

- *aTreeData* — данные дерева Хаффмана.

Возвращает:

- *aTreeData* — данные дерева Хаффмана со счетчиками равными половине исходного заначения и перестроенное дерево.

```
procedure dhHalfCounters(
      var aTreeData:tDynHuffmanFullTreeData
);
 var b:tBaseNodeIndex; i,f,l,j:tIndex;
     Index:tBaseNodesDataEx; EOP:boolean;
begin
      with aTreeData do begin
            { ополовиниваем все счетчики для базовых байтов
            и строим индекс для базовых байтов }

            { первые два узла можно пропустить - они уже на нужном месте}
            Nodes[0].Counter:=Succ(Nodes[0].Counter shr 1);
            Nodes[1].Counter:=Succ(Nodes[1].Counter shr 1);
            b:=Low(b)+1;
            i:=Low(i)+2;
            repeat
                  if (Nodes[i].UpperNode>=cBaseNodeCount) then begin
                        Inc(b);
                        { счетчик ополовиниваем }
                        with Index[b] do begin
                              UpperNode:=Nodes[i].UpperNode;
                              Counter:=Succ(Nodes[i].Counter shr 1);
                        end;
                  end;
                  Inc(i);
            until b=High(b);
            { по окончании этого цикла в массиве Index находится
            упорядоченная по возрастанию частоты (Counter) последовательность
            "базовых узлов", кроме двух первых }

            { НАЧИНАЕМ СТРОИТЬ ДЕРЕВО }

            { Инициализируем две последовательности:
            1 - в Index;
            2 - в Nodes (начиная с cBaseNodeCount)}
            i:=Low(i)+2; { i - далее номер первого необработанного узла
                           из Index}
            f:=Low(f)+2; { f - далее номер первого свободного узла в Nodes }

            j:=cBaseNodeCount; { далее j = номер первого необработанного
                                 узла из Nodes (вторая последовательность) }
            l:=cBaseNodeCount; { новый узел }

            { добавляем в последовательность 2 новый узел }
            with Nodes[cBaseNodeCount] do begin
                  { запоминаем сумму счетчиков частоты вхождений }
                  Counter:=Nodes[0].Counter+Nodes[1].Counter;
                  { запоминаем ссылку на предыдущие узлы }
                  UpperNode:=0;
```

```
end;

{ пока не пуста 1 последовательность }
while i<cBaseNodeCount do begin
      { добавляем в таблицу новый узел }
      Inc(l);

      { Выбираем минимальные узлы для объединения в новый узел. }

      { значение 1-го счетчика 2-й последовательности
      сравниваем со значеним 1-го счетчика 1-й последовательности }
      if Nodes[j].Counter<Index[i].Counter then begin
            { если значение счетчика 2-й последовательности
            меньше, то выбираем его как первый (левый) узел
            для слияния }
            Nodes[f]:=Nodes[j];
            Refs.NextNode[Nodes[f].UpperNode]:=f;

            { и исключаем этот узел из 2-й последовательности }
            Inc(j);
            Inc(f);

            { проверяем окончание последовательности 2}
            EOP:=(j=l);
            if EOP then begin
                  { если послед. 2 закончилась,
                  то выбираем 1-й узел 1-й последовательности
                  как второй (правый) узел для слияния }
                  Nodes[f]:=Index[i];
                  Refs.NextNode[Index[i].UpperNode]:=f;
                  { и исключаем этот узел из 1-й послед. }
                  Inc(i);
            end;
      end else begin
            { если значение счетчика 1-й последовательности
            меньше, то выбираем его как первый (левый)
            узел для слияния }
            Nodes[f]:=Index[i];
            Refs.NextNode[Index[i].UpperNode]:=f;

            { и исключаем этот узел из 1-й последовательности }
            Inc(i);
            Inc(f);

            { проверяем окончание последовательности 1}
            EOP:=(i=cBaseNodeCount);
            if EOP then begin
                  { если послед. 1 закончилась, то выбираем
                  1-й узел 2-й последовательности как второй
                  (правый) узел для слияния }
                  Nodes[f]:=Nodes[j];
                  Refs.NextNode[Nodes[f].UpperNode]:=f;
                  { и исключаем этот узел из 2-й послед. }
                  Inc(j);
            end;
      end;
      if not EOP then begin
            { если в последовательностях есть узлы, то
            значение 1-го счетчика 2-й последовательности
            сравниваем со значеним 1-го счетчика 1-й
            последовательности }
            if Nodes[j].Counter<Index[i].Counter then begin
                  { если значение счетчика
                  2-й последовательности меньше,
                  то выбираем его как как второй (правый)
                  узел для слияния }
                  Nodes[f]:=Nodes[j];
```

```
                              Refs.NextNode[Nodes[f].UpperNode]:=f;

                              { и исключаем этот узел из 2-й послед. }
                              Inc(j);
                        end else begin
                              { иначе выбираем 1-й узел
                              1-й последовательности как второй (правый)
                              узел для слияния }
                              Nodes[f]:=Index[i];
                              Refs.NextNode[Index[i].UpperNode]:=f;

                              { и исключаем этот узел из 1-й послед. }
                              Inc(i);
                        end;
                  end;

                  with Nodes[l] do begin
                        { запоминаем сумму счетчиков частоты вхождений }
                        Counter:=Nodes[Pred(f)].Counter+Nodes[f].Counter;
                        { запоминаем ссылку на предыдущие узлы }
                        UpperNode:=f shr 1;
                  end;
                  Inc(f);
            end; { while }

            { пока не пуста 2 посл. }
            while (f<l) do begin
                  { добавляем в таблицу новый узел }
                  Inc(l);
                  { объединяем минимальные узлы в новый узел }
                  Refs.NextNode[Nodes[f].UpperNode]:=f;
                  Inc(f);
                  Refs.NextNode[Nodes[f].UpperNode]:=f;
                  with Nodes[l] do begin
                        { запоминаем сумму счетчиков частоты вхождений }
                        Counter:=Nodes[Pred(f)].Counter+Nodes[f].Counter;
                        { запоминаем ссылку на предыдущие узлы }
                        UpperNode:=f shr 1;
                  end;
                  Inc(f);
            end;

            IncVLC(TotalHalfCounterCount,1);
      end;
end;
```

3.8. Размер и формат записи упакованных данных

В отличие от статического метода Хаффмана, в динамическом невозможно заранее сказать каков будет размер упакованных данных, более того нельзя гарантировать НЕУВЕЛИЧЕНИЕ размера данных. Все это связано с использованием «предыдущих» данных, как оценки статистики частот для «последующих» данных. Если корреляция такого сорта отсутствует в данных — результат обработки данных может быть длиннее, чем исходные данные. Пример такого увеличения размера кодированных данных легко получить попытавшись упаковать динамическим методом Хаффмана архив .zip, .rar или провторно применив динамический метод Хаффмана к упакованному им же файлу.

При записи в файл результаты динамического метода Хаффмана не требуют записи никакой дополнительной информации — в кодированный файл вы-

водятся только коды Хаффмана для байтов в той последовательности, в которой следовали байты исходного файла.

3.10. 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

4.2. Краткое описание тестовой программы для динамического метода Хаффмана

Для выполнения лабораторной работы вам предоставлен работоспособный программный код, реализующий динамический метод Хаффмана, и программа, использующая этот код для компрессии/декомпрессии произвольного файла. Программный код был написан с учебными целями, поэтому в некоторых местах эффективность принесена в жертву простоте и понятности кода.

Для удобства модули программы сгруппированы в различные папки. Размещение модулей программы в папках следующее:

- «Сжатие данных. Динамический метод Хаффмана» — основная папка, здесь находятся:
  - «Компрессия данных или измерение и избыточность информации. Динамический метод Хаффмана.doc» (или .pdf) — методические указания к лабораторной работе.
  - «Программа» — папка с исходным кодом программы.
    - «1) DOS - BP7» — вариант программы для Borland Pascal 7.0.
    - «2) Win - Delphi5» — вариант программы для Borland Delphi 5.0.
    - «DynHuff» — модули алгоритма динамического метода Хаффмана, пригодные для обеих платформ (Borland Pascal 7.0 и Delphi 5.0).
    - «Common» — вспомогательные модули.
    - «RunMe.bat» — вспомогательный файл для удобства компиляции и запуска в Borland Pascal 7.0. Файл «RunMe.bat» при запуске создает отображение папки, в которой он находится, в диск «P:». Файл «RunMe.bat» работает только под ОС Windows NT 4.0/2000/XP.

Программа состоит из следующих модулей.

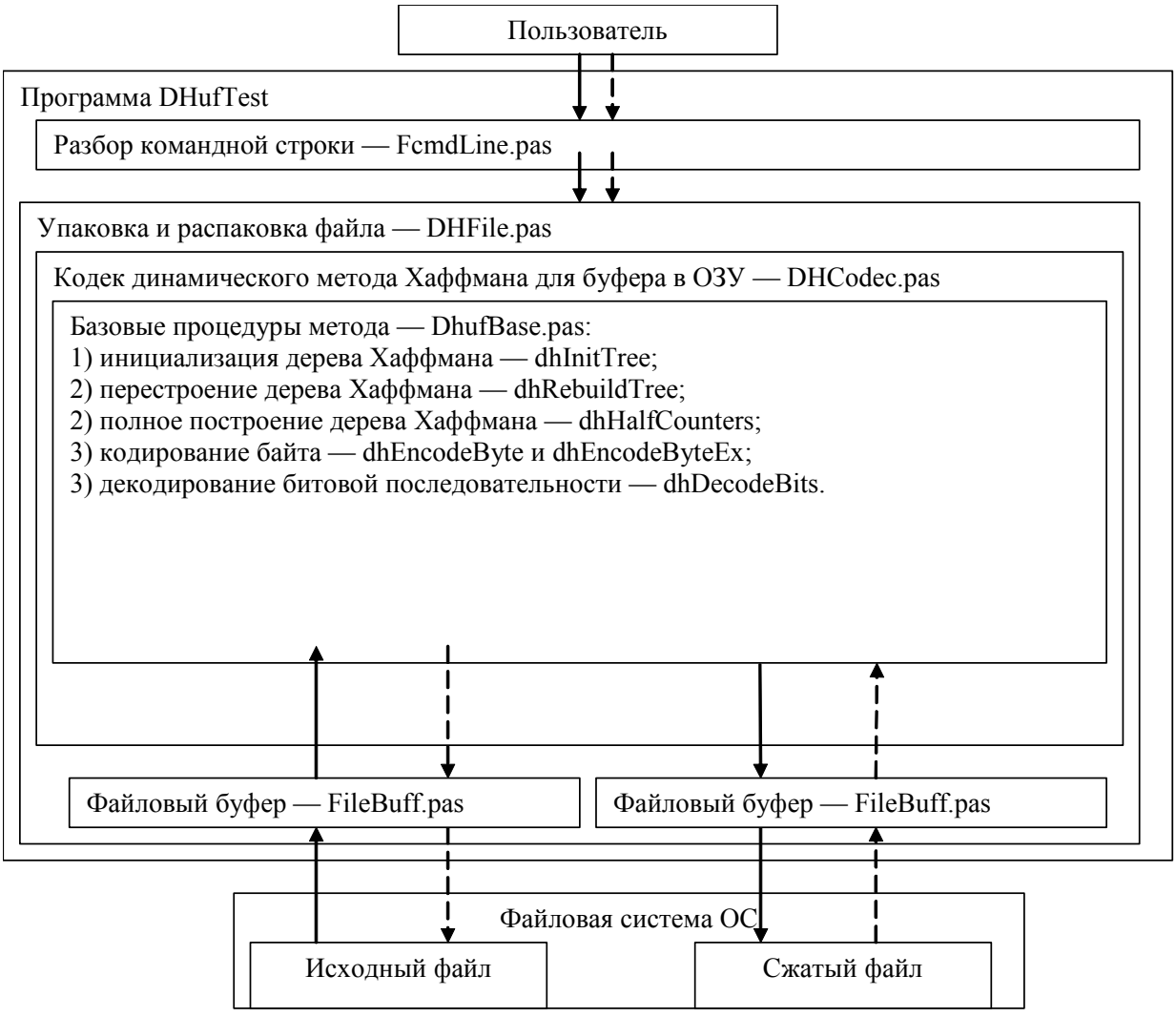
1) Программных модулей, реализующих динамический метод Хаффмана, см. папку DynHuff:
   - DHufType.PAS, HuffType.PAS — общие определения типов,
   - DHUFBase.PAS — основные операции по инициализации дерева, перестроению дерева, кодированию байта и декодированию битовой последовательности, см. пункты 3.6.2-3.6.8.
2) Программных модулей, реализующих сжатие буфера/файла, см. папку DynHuff:
   - DHCodec.PAS — кодирование и декодирование буфера,
   - DHFile.PAS — кодирование и декодирование файла.
3) Программы сжатия файла, см. папку «DOS - BP7» или «Win - Delphi5»:
   - DHFTEST.PAS/.DPR — главная программа упаковки/распаковки файла,

- FCMDLINE.PAS — разбор командной строки программы,
- PRGFUNCS.PAS — вспомогательные функции для программы.

4) Вспомогательных программных модулей, см. папку Common:
   - FileBuf.PAS — вспомогательный буфер для чтения/записи файла,
   - xStrings.pas — вспомогательные функции работы со строками.

Схема взаимодействия модулей программы показана на рис. 4.1. На рисунке приведены только основные модули.

СХЕМА ВЗАИМОДЕЙСТВИЯ МОДУЛЕЙ В ТЕСТОВОЙ ПРОГРАММЕ



— поток данных при упаковке; - - - поток данных при распаковке файла.

Рис. 4.2

4.4. КОМПИЛЯЦИЯ ТЕСТОВОЙ ПРОГРАММЫ В BP 7.0

Для получения возможности модифицировать код программы в BP 7.0:

1) Скопируйте папку «Сжатие данных. Динамический метод Хаффмана» на локальный диск в любое место.
2) Запустите «Far» или «Norton Commander» и переместитесь в папку «Сжатие данных. Динамический метод Хаффмана \Программа».
3) Запустите файл «RunMe.bat» — в системе появится новый диск «P:»[*] .
4) Переместитесь на диск P: — в действительности это папка «Сжатие данных. Динамический метод Хаффмана \Программа».
5) Переместитесь в папку «P:\ 1) DOS - BP7», запустите Borland Pascal 7.0 — наберите BP.exe и нажмите клавишу ENTER.

Вы готовы откомпилировать модули и получить исполняемый файл программы:

1) Нажмите клавишу Ctrl+F9 — должна запуститься программа как это изображено на рис. 4.2.
2) Исполняемый файл «DHFTest.exe» компилятор поместит в папку «C:\TMP».
3) Если программа не запустилась и появились сообщения об ошибках — обратитесь к преподавателю.

Программа может выполнять следующие операции:

1) Упаковывать любой файл динамическим методом Хаффмана (по умолчанию упаковывается файл «TEST.txt» в файл «TEST._tx»).
2) Распаковывать упакованный динамическим методом Хаффмана файл (по умолчанию распаковывается файл «TEST._tx» в файл «TEST.!tx»).
3) Упаковывать динамическим методом Хаффмана, сразу распаковывать и сравнить результат распаковки и оригинальный файл (см. рис. 4.2).

Дополнительно вычисляются некоторые характеристики компрессии.

Программа предназначена для тестирования динамического метода Хаффмана на реальных данных (файлах).

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

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

>DHFTEST.exe

4.6. Компиляция тестовой программы в Delphi 5.0

Для получения возможности модифицировать код программы в Delphi 5.0:

---

[*] На Windows 9x/Me не работает отображение папки в диск, которое выполняет файл «RunMe.bat». Следует переместить файлы программы в любую папку с более коротким путем доступа (не более 127 символов) и изменить параметр меню *Compile* → *Primary file*, указав на DHFTEST.pas.

1) Скопируйте папку «Сжатие данных. Динамический метод Хаффмана» на локальный диск в любое место.
2) Переместитесь в папку «Сжатие данных. Динамический метод Хаффмана \Программа\Win - Delphi5».
3) Щелкните (дважды) мышкой на файле «DHFTest.dpr» — запустится Delphi 5.0.

Вы готовы откомпилировать модули и получить исполняемый файл программы:

1) Нажмите клавишу F9 — должна запуститься программа как это изображено на рис. 4.2.
2) Исполняемый файл «DHFTest.exe» компилятор поместит в папку «C:\TEMP».
3) Если программа не запустилась и появились сообщения об ошибках — обратитесь к преподавателю.

Программа может выполнять следующие операции:

1) Упаковывать любой файл динамическим методом Хаффмана(по умолчанию упаковывается файл «TEST.txt» в файл «TEST._tx», расположенные в той же папке, где и «DHFTest.exe»).
2) Распаковывать упакованный динамическим методом Хаффмана файл (по умолчанию распаковывается файл «TEST._tx» в файл «TEST.!tx»).
3) Упаковывать динамическим методом Хаффмана, сразу распаковывать и сравнить результат распаковки и оригинальный файл (см. рис. 4.2).

Дополнительно вычисляются некоторые характеристики компрессии.

Программа предназначена для тестирования динамического метода Хаффмана на реальных данных (файлах).

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

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

```
>DHFTEST.exe
```

Пример результата выполнения приведен на рис. 4.2. Ключевой признак правильной работы — слово «правильно», это означает, что упаковка и последующая распаковка привели к данным, совпадающим с исходными.

ВНИМАНИЕ! Собственный вариант код следует проверить на нескольких разных (по размеру и содержимому) файлах.

ПРИМЕР РАБОТЫ *DHFTEST*

```
Программа DHFileTest запущена (real mode)...
Разбор командной строки...
Завершен разбор командной строки.
 Ополовинивание счетчиков через, симв.: 3000
Упаковка: "test.txt" -> "test._tx" (повтор, раз: 1)... завершено.
 Время на 1 упаковку, мс: 11660.0; Скорость, байт/с: 330600.
 Отношение (Размер упакованного файла)/(Размер исходного): 0.8018
 Степень сжатия данных (упакованные)/(исходные): 0.8017
 Ополовиниваний счетчиков:       2960
Распаковка: "test._tx" -> "test.!tx" (повтор, раз: 1)... завершено.
 Время на 1 распаковку, мс: 9845.0; Скорость, байт/с: 391550.
Проверка. Сравнение "test.txt" и "test.!tx"... завершено.
 Правильно.
программа DHFileTest завершена.
```

Рис. 4.4

4.8. ВАРИАНТЫ ЗАДАНИЙ

Задание лабораторной работы выполняется индивидуально. Варианты помеченные звездочкой имеют повышенную сложность и могут выполняться группой до 2-х человек. 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

1. Научиться компилировать программу и запускать тестовую программу, см. пункты 4.4 и 4.6.
2. Научиться упаковывать и распаковывать произвольный файл с помощью тестовой программы.

Вторая часть задания 1

1. Объяснить, почему файл сжатый тестовой программой не сжимается еще сильнее другими архиваторами (zip, rar и т.п.), либо сжимается незначительно. И, что особенно важно, файл сжатый тестовой программой сжимается хуже, чем если бы архиватор сжимал исходный файл.
2. Объяснить, почему файлы сжимаются в разной степени.
3. Какой файл будет иметь максимальное сжатие тестовой программой.

4. Какой файл не будет сжиматься тестовой программой совсем.
5. Создать самый длинный файл с максимальным сжатием тестовой программой.
6. Создать самый короткий файл с отсутствием сжатия тестовой программой.

Третья часть задания 1

1. Модифицировать тестовую программу 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 компилированным вариантом тестовой программы DHFTest.exe, который предоставлен вам вместе с исходным кодом. Этот вариант откомпилирован с параметрами, обеспечивающими максимальное быстродействие. Сравнение проводится при одинаковых параметрах. Сравнение должно быть проведено на различных типах файлов, не менее трех: *.doc; *.exe; *.zip. Размер файлов должен быть не менее 2 Мбайт.

Вариант 2*

1. Если вы считаете, что предложенная реализация динамического метода Хаффмана не лучшим образом реализует алгоритм и вы можете самостоятельно написать лучший вариант, то Вам предоставляется это право (на любом языке).
2. Единственное требование — создать аналог «DHFTest.pas» для сравнения с примером программы.

Вариант 3*

1. Реализовать иной способ построения динамического дерева. Например, использовать вариант с заполнением дерева «на ходу», см. пункт 2.4. Сравнить быстродействие и степень сжатия для вашего варианта с предоставленной программой. Сделать выводы о предпочтительной конструкции дерева.

Вариант 4*

1. Спроектировать данные, описать алгоритм и написать процедуры для динамического варианта арифметического метода. В качестве образца использовать код предоставленной программы.
2. Проверить работоспособность метода.
3. Сравнить быстродействие и эффективность[3] метода с динамическим методом Хаффмана.

---

[*] Варианты помеченные звездочкой имеют повышенную сложность и могут выполняться группой в 2 человека.

[3] Под эффективностью понимается, прежде всего, степень сжатия исходных данных.

Вариант 5*

1. Подробно описать алгоритм для любого метода сжатия без потерь (**кроме метода Хаффмана и LZ78**) с конкретным примером ручного вычисления для небольшой порции данных 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 предоставленной программы.
3. Попробовать проверить работоспособность метода.
4. Попробовать сравнить быстродействие и эффективность метода с динамическим методом Хаффмана.

Вариант 6*

1. Создать комбинированный архиватор, например, LZ78+дожатие методом Хаффмана или иным.
2. Проверить работоспособность усовершенствований на примере файлового сжатия.
3. Сравнить быстродействие и эффективность усовершенствованого сжатия с исходными методами.

Вариант 7*

1. Разработать структуру 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.

4.10. Оформление результатов работы

Вы должны представить письменный отчет (один на группу) по выполненной работе (10÷20 страниц, не считая листингов программы — листинги рекомендуется не печатать) и работоспособный код программы. Отчет должен быть оформлен в соответствии со стандартом [3].

Отчет должен состоять из следующих частей:

1) титульный лист;
2) введение;
3) основная часть (может состоять из нескольких глав);
4) заключение;

5) список использованных источников.

Отчет должен содержать:

1) краткий обзор математических алгоритмов сжатия информации, приветствуется описание алгоритмов не упомянутых в данных методических указаниях;
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.

4.12. Прием зачета по результатам работы

Зачет принимается в форме обсуждения отчета о выполнении лабораторной работы и программы с членами группы, представившей отчет. При обсуждении отчета 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 Лемпеля-Зива и его модификации.
2) Знание устройства и взаимодействия частей представленного и/или своего кода программы.
3) Умение компилировать код и запускать программу.
4) Умение модифицировать свой 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 динамического алгоритма Хаффмана.

4) Получите практические навыки разработки и кодирования алгоритмов.

Любые улучшения алгоритма будут учитываться как дополнительная заслуга при сдаче зачета. Улучшения должны быть работающие, голые идеи не в счет.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. КОМПРЕССИЯ ДАННЫХ ИЛИ ИЗМЕРЕНИЕ И ИЗБЫТОЧНОСТЬ ИНФОРМАЦИИ. Метод Хаффмана: Методические указания к лабораторной работе/ О. Е. Александров, Попков В.И. Екатеринбург: УГТУ, 2000. 49 с.
2. КОМПРЕССИЯ ДАННЫХ ИЛИ ИЗМЕРЕНИЕ И ИЗБЫТОЧНОСТЬ ИНФОРМАЦИИ. Метод ЛЕМПЕЛЯ-ЗИВА: Методические указания к лабораторной работе / О. Е. Александров. Екатеринбург: УГТУ, 2001. 54 с.
3. СТП УГТУ-УПИ 1-96. Общие требования и правила оформления дипломных и курсовых проектов (работ). 1996. 34 с. Группа Т51.