МИНИСТЕРСТВО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

УРАЛЬСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

кафедра молекулярной физики

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНОЙ РАБОТЕ

КОМПРЕССИЯ ДАННЫХ ИЛИ ИЗМЕРЕНИЕ И ИЗБЫТОЧНОСТЬ ИНФОРМАЦИИ. МЕТОД ЛЕМПЕЛЯ-ЗИВА

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

Составитель: О. Е. Александров.

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

КОМПРЕССИЯ ДАННЫХ ИЛИ ИЗМЕРЕНИЕ И ИЗБЫТОЧНОСТЬ ИНФОРМАЦИИ. Метод ЛЕМПЕЛЯ-ЗИВА: Методические указания к лабораторной работе / О. Е. Александров. Екатеринбург: УГТУ, 2001. 54 с.

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

Приведено подробное описание алгоритма Лемпеля-Зива и его модификаций.

Материалы предназначены для студентов кафедры «Молекулярная физика».

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

Библиогр. 3 назв. Рис. 15. Табл. 3. 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 — шестнадцатеричное 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

Одна из задач любой информационной системы — обеспечивать хранение и передачу информации. Причем хранение и передача информации занимают определяющее место в функционировании информационной системы.

Нынешний век называют информационным веком — информация играет все более и более важную роль в современной жизни. Ее объемы постоянно возрастают и требуются все большие хранилища, 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. Просто уменьшать объем информации нежелательно и приходится искать другие способы уменьшения. Необходимо уменьшить объем информации, не изменяя ее содержание. Такой процесс называется архивацией, компрессией или сжатием данных.

На практике возможно сжатие практически любой, т.н. «обычной» информации. 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 звуковой речи, так как в ней обязательно есть частоты, не слышимые человеком, или несущественные для восприятия. Избыточно представление информации в электронном виде, обязательно есть какие-то повторяющиеся символы, цепочки символов. Удалив избыточность мы можем уменьшить потребности в емкостях и пропускных способностях линий передачи, необходимых для хранения и передачи информации и при этом не уменьшив содержательную сторону информации, т.е.

сохранив возможность восстановления ее к исходному виду (такое сжатие называется обратимым).

В данной лабораторной работе вы ознакомитесь с одним из алгоритмов сжатия данных — алгоритмом Лемпеля-Зива.

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 несущественная 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 — удаления несущественной части данных — может быть достигнута более высокая степень сжатия. В этом случае потери при сжатии означают незначительное искажение изображения (звука) которые не препятствуют нормальному восприятию (незаметны), но при детальном сличении оригинала и восстановленной после сжатия копии могут быть обнаружены.

Об алгоритмах сжатия с потерями можно прочитать в методических указаниях к лабораторной работе «Компрессия данных или измерение и избыточность информации. Метод Хаффмана» [1].

2. АЛГОРИТМЫ СЖАТИЯ БЕЗ ПОТЕРЬ

2.1. Обзор алгоритмов

В настоящее время существует несколько алгоритмов сжатия без потерь, частично это открытые[1] алгоритмы, частично коммерческие[2] 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, ACE и др.) достаточно известны и с ними можно познакомиться самостоятельно.

Алгоритмы обратимого сжатия данных можно разделить на две группы:

1) Алгоритмы частотного анализа — подсчет частоты различных символов в данных и преобразование кодов символов с соответствии с их частотой.
2) Алгоритмы корреляционного анализа — поиск корреляций (в простейшем случае точных повторов) между различными участками данных и замена коррелирующих данных на код(ы), позволяющая восстановить данные на основе предшествующих данных. В простейшем случае точных повторов, кодом является ссылка на начало предыдущего вхождения этой последовательности символов в данных и длина последовательности.

Можно отметить следующие алгоритмы обратимого сжатия данных из первой группы:

1) Метод Хаффмана — замена кода равной длины для символов на коды неравной длины в соответствии с частотой появления символов в данных, коды минимальной длины присваиваются наиболее часто встречающимся символам. Если частоты появления символов являются степенью двойки (2^n), то этот метод достигает теоретической энтропийной границы эффективности сжатия для методов такого типа.
2) Метод Шеннона-Фано — сходен с методом Хаффмана, но использует другой алгоритм генерации кодов и не всегда дает оптимальные коды (оптимальный код — код дающий наибольшее сжатие данных из всех возможных для данного типа преобразования).
3) Арифметическое кодирование — усовершенствование метода Хаффмана, позволяющее получать оптимальные коды для данных, где частоты появления символов не являются степенью двойки (2^n). Этот метод достигает тео-

[1] Открытые алгоритмы обычно рассматривают только саму идею, не останавливаясь на проблемах ее реализации в виде программы или реализованы в виде «демонстрационной» (не очень эффективной) программы.

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

ретической энтропийной границы эффективности сжатия этого типа для любого источника.

Для второй группы можно назвать следующие алгоритмы:

1) Метод Running (или RLE) — заменяет цепочки повторяющихся символов на код символа и число повторов. Это пример элементарного и очень понятного метода сжатия, но, к сожалению, он не обладает достаточной эффективностью.
2) Методы Лемпеля-Зива — это группа алгоритмов сжатия объединенная общей идеей: поиск повторов фрагментов текста в данных и замена повторов ссылкой (кодом) на первое (или предыдущее) вхождение этого фрагмента в данные. Отличаются друг от друга методом поиска фрагментов и методом генерации ссылок (кодов).

О некоторых алгоритмах сжатия без потерь можно прочитать в методических указаниях к лабораторной работе «Компрессия данных или измерение и избыточность информации. Метод Хаффмана» [1].

В настоящее время, в различных модификациях и сочетаниях, два алгоритма — метод Хаффмана (или арифметическое кодирование) и метод Лемпеля-Зива — составляют основу всех коммерческих алгоритмов и программ.

Ниже подробно будет рассмотрен алгоритм сжатия Лемпеля-Зива.

2.2. Идея алгоритма Лемпеля-Зива. Замещающие компрессоры

Основная идея алгоритма Лемпеля-Зива состоит в замене появления фрагмента в данных (группы байт) ссылкой на предыдущее появление этого фрагмента. Существуют два основных класса методов, названных в честь Якоба Зива (Jakob Ziv) и Абрахама Лемпеля (Abraham Lempel), впервые предложивших их в 1977 (алгоритм LZ77) и 1978 годах (алгоритм LZ78).

2.3. Алгоритм LZ78

Этот алгоритм генерирует на основе входных данных словарь фрагментов, внося туда фрагменты данных (последовательности байт) по определенным правилам (см. ниже) и присваивает фрагментам коды (номера). При сжатии данных (поступлении на вход программы очередной порции) программа на основе LZ78 пытается найти в словаре фрагмент максимальной длины, совпадающий с данными, заменяет найденную в словаре порцию данных кодом фрагмента и дополняет словарь новым фрагментом. При заполнении словаря (размер словаря ограничен по-определению) программа очищает словарь и начинает процесс заполнения словаря снова. Реализации этого метода различают-

ся конструкцией словаря, алгоритмами его заполнения и очистки при переполнении.

Новые фрагменты обычно добавляются в словарь по следующему алгоритму: к найденному в словаре фрагменту добавляется следующий за фрагментом символ из входных данных, образуя новый фрагмент. Например, если в словаре имелся фрагмент «abc», а на вход поступила последовательность «abcx», то LZ78 найдет ее начало «abc» в словаре и пополнит словарь фрагментом «abcx». В общем случае, можно заполнять словарь быстрее, наращивая фрагменты не одним символом, а двумя или более символами, но это может привести к быстрому заполнению словаря, и не обязательно улучшит степень сжатия.

Оригинальный LZ78 выводит в упакованные данные коды, состоящие из пары значений: (код_фрагмента_в_словаре, следующий_за_фрагментом_символ_в_исходных_данных). Это гарантирует возможность вывода данных, даже если в словаре не найдено подходящего фрагмента, тогда в качестве кода_фрагмента_в_словаре выводится ноль — нет фрагмента.

В 1984 году Терри Уэлч (Terry Welch) предложил вариант LZ78, в котором в упакованные данные выводятся только коды_фрагмента_в_словаре. А для того, чтобы в словаре всегда существовал подходящий фрагмент, Терри Уэлч предложил предварительно заполнить словарь всеми односимвольными (элементарными) фрагментами — всеми возможными значениями байта от 0 до 255. Это гарантирует, что при поступлении на вход очередной порции данных будет найден в словаре хотя бы однобайтовый фрагмент. Алгоритм известен под названием LZW. Далее будет рассматриваться именно LZW-вариант алгоритма.

Алгоритм LZ78 резервирует специальный код, назовем его «Reset», который вставляется в упакованные данные, отмечая момент сброса словаря. Значение кода Reset обычно принимают равным 256.

Таким образом при начале кодирования минимальная длина кода составляет 9 бит. Максимальная длина кода зависит от объема словаря — количества различных фрагментов, которые туда можно поместить. Если объем словаря измерять в байтах (символах), то очевидно, что максимальное количество фрагментов равно числу символов, а, следовательно, максимальная длина кода равна $\log_2$(объем_словаря_в_байтах).

Рассмотрим алгоритм заполнения словаря на примере кодирования фрагмента. Для примера используем данные, состоящие только из символов (букв), кавычки «» не входят в данные:

«abcdabce·abcd·abceabcabcabcd·····abcaabcdabce·abceca»,

где символ «·» обозначает пробел. Пример кодировки приведен в табл. 2.1.

Таблица 2.1

ПРИМЕР РАБОТЫ КОДИРОВКИ LZ78

Данные на входе — байты	Текущий фрагмент	Данные на выходе — коды	Комментарий	Добавление фрагмента в словарь — «фрагмент»:код
1.		256	сброс словаря при начале кодирования данных	… «·»:32; … «a»:97; «b»:98; «c»:99; «d»:100; «e»:101 …
2. a			фрагмент «a» есть в словаре	
3. b	a	97	фрагмента «ab» нет в словаре	«ab»: 257
4. c	b	98	фрагмента «bc» нет в словаре	«bc»:258
5. d	c	99	-/-	«cd»:259
6. a	d	100	-/-	«da»:260
7. b	a		фрагмент «ab» есть в словаре	
8. c	ab	257	фрагмента «abc» нет в словаре	«abc»:261
9. e	c	99	-/-	«ce»:262
10. ·	e	101		«e·»:263
11. a	·	32		«·a»:264
12. b	a			
13. c	ab			
14. d	abc	261		«abcd»:265
15. ·	d	100		«d·»:266
16. a	·a			
17. b	·a	264		«·ab»:267
18. c	b			
19. e	bc	258		«bce»:268
20. a	e	101		«ea»:269
21. b	a			
22. c	ab			
23. a	abc	261		«abca»:270
24. b	a			
25. c	ab			
26. a	abc			

Данные на входе — байты	Текущий фрагмент	Данные на выходе — коды	Комментарий	Добавление фрагмента в словарь — «фрагмент»:код
27. b	abca	270		«abcab»:271
28. c	b			
29. d	bc	258		«bcd»:272
30. ·	d			
31. ·	d·	266		«d··»:273
32. ·	·	32		«··»:274
33. ·	·			
34. ·	··	274		«···»:275
35. a	·			
36. b	·a			
37. c	·ab	267		«·abc»:276
38. a	c	99		«ca»:277
39. b	a			
40. c	ab			
41. d	abc			
42. a	abcd	265		«abcda»:278
43. b	a			
44. c	ab			
45. e	abc	261		«abce»:279
46. ·	e·			
47. a	e·	263		«e·a»:280
48. b	a			
49. c	ab			
50. e	abc			
51. c	abce	279		«abcec»:281
52. a	c		фрагмент «ca» есть в словаре	
53.	ca	277	закончились данные — завершен процесс кодирования — надо вывести код последнего фрагмента	
если бы данные были длиннее, то процесс продолжался бы до заполнения словаря…				
		256	словарь заполнен — сброс	… «·»:32; … «a»:97; «b»:98; «c»:99; «d»:100; «e»:101 …
			процесс продолжается сно-	

Данные на входе — байты	Текущий фрагмент	Данные на выходе — коды	Комментарий	Добавление фрагмента в словарь — «фрагмент»:код
ва…				

В исходных данных 52 *байта* или 52×8=416 *бит*. В кодированных данных 27 кодов по 9 *бит* или 27×9=243 *бита*. Уменьшение размера данных составило 1,71 раза или кодированные данные составляют 0,58 исходного размера.

Дополнительных данных для декодировки не требуется (сравните с методом Хаффмана, см. [1]). Восстановить исходные данные можно, располагая только кодированной последовательностью[3]. Действительно, если считать, что в качестве входных данных поступает кодированная последовательность, то можно восстанавливать словарь так, что к моменту поступления нового кода на вход, в словаре уже будет соответствующая последовательность символов (табл. 2.2).

Особый случай имеет место на шаге 16 и шаге 20 (см. табл. 2.2). В этом случае на вход попадает код, который отсутствует в словаре. Но очевидно, что фрагмент для этого кода может быть построен в момент поступления кода. Значение фрагмента для кода равно предыдущему фрагменту плюс первый символ (байт) предыдущего фрагмента.

Таблица 2.2

ПРИМЕР РАБОТЫ ДЕКОДИРОВКИ LZ78

Данные на входе — коды	Текущий фрагмент	Данные на выходе — байты	Комментарий	Добавление фрагмента в словарь, «фрагмент»:код
1. 256			сброс словаря при начале декодирования данных	… «·»:32; … «a»:97; «b»:98; «c»:99; «d»:100; «e»:101 …
2. 97	a	a	код 97 есть в словаре	
3. 98	b	b	фрагмента «ab» нет в словаре	«ab»: 257
4. 99	c	c	фрагмента «bc» нет в словаре	«bc»:258
5. 100	d	d	-/-	«cd»:259
6. 257	ab	ab	-/-	«da»:260
7. 99	c	c		«abc»:261
8. 101	e	e		«ce»:262
9. 32	.	.		«e·»:263

[3] Такие алгоритмы еще называют адаптивными, в отличие от алгоритмов типа Хаффмана, которые называют статическими.

Данные на входе — коды	Текущий фрагмент	Данные на выходе — байты	Комментарий	Добавление фрагмента в словарь, «фрагмент»:код
10. 261	abc	abc		«.a»:264
11. 100	d	d		«abcd»:265
12. 264	·a	·a		«d·»:266
13. 258	bc	bc		«·ab»:267
14. 101	e	e		«bce»:268
15. 261	abc	abc		«ea»:269
16. 270	abca	abca	кода 270 нет в словаре!!! Конструируем фрагмент для него из текущего фрагмента+первый байт текущего фрагмента.	«abca»:270
17. 258	bc	bc		«abcab»:271
18. 266	d·	d·		«bcd»:272
19. 32	·	·		«d··»:273
20. 274	··	··	кода 274 нет в словаре!!!	«··»:274
21. 267	·ab	·ab		«···»:275
22. 99	c	c		«·abc»:276
23. 265	adcd	adcd		«ca»:277
24. 261	abc	abc		«abcda»:278
25. 263	e·	e·		«abce»:279
26. 279	авсе	авсе		«e·a»:280
27. 277	ca	ca		«abcec»:281

Таким образом, алгоритм упаковки и распаковки методом LZ78 достаточно прост. Основную проблему, при реализации этого метода, представляет устройство словаря.

Очевидно, что чем больше словарь, тем большую степень сжатия можно достичь[4] при равных прочих условиях.

С другой стороны, важным практическим моментом является скорость упаковки, этот параметр тоже зависит от устройства словаря. Основные операции при упаковке: 1) поиск в словаре фрагмента; 2) вставка в словарь новых фрагментов. Необходимо, чтобы эти две операции были максимально быстрыми.

Вопрос о конструкции словаря для LZ78 будет рассмотрен ниже. А пока рассмотрим альтернативный алгоритм LZ77.

[4] Это утверждение, возможно, и справедливо при стремлении словаря к бесконечному размеру, но при сжатии конкретного файла может наблюдаться обратный эффект, когда размер сжатого файла увеличивается с ростом словаря.

2.4. Алгоритм LZ77

Алгоритм LZ77 предлагает оригинальное решение именно для устройства словаря. Вместо построения словаря для хранения фрагментов в виде отдельной структуры данных, LZ77 предлагает использовать предшествующий фрагмент исходных данных конечной длины, как готовый словарь.

Метод LZ77 оперирует двумя элементами данных: 1) буфер предыстории или словарь — уже обработанная порция исходных данных фиксированной длины; 2) буфер предпросмотра — еще не обработанная порция исходных данных фиксированной длины, следующая за буфером предыстории.

Метод LZ77 пытается найти в буфере предыстории фрагмент текста максимальной длины, совпадающий с началом буфера предпросмотра и когда это удается, то на выход выдается пара значений: позиция_фрагмента_в_буфере_-предыстории и длина_фрагмента, иначе выводится первый символ буфера предпросмотра и повторяется попытка поиска для следующей пары: буфер предыстории, буфер предпросмотра. В упрощенном виде алгоритм можно записать, как это показано на рис. 2.1.

Упрощенный алгоритм LZ77

```
пока (буфер_предпросмотра не пуст )
      найти наиболее длинное соответствие (позиция_начала, длина) в
      буфере_предыстории и в буфере_предпросмотра;
      если (длина>минимальной_длины), то
            вывести в кодированные данные пару (позиция_начала, длина);
      иначе
            вывести в кодированные данные первый символ буфера_предпросмотра;
      изменить указатель на начало буфера_предпросмотра;
      сдвинуть буфер предыстории;
продолжить.
```

Рис. 2.1

LZ77 передвигает окно фиксированного размера (буфер предыстории) по данным. Очевидно, что наилучшего результата можно было бы достичь, если бы буфер предыстории был бы по размеру равен данным, но на практике используют ограниченный буфер (обычно 16 или 32 *кБайт*). Это обусловлено, как очевидными ограничениями по размерам ОЗУ, так и необходимостью ограничить время поиска фрагмента в буфере.

Поскольку метод не требует построения словаря, а для поиска фрагмента в буфере существуют очень эффективные алгоритмы (о быстром поиске строки см. [2]), то LZ77 широко применяется на практике. Большинство коммерческих архиваторов (ace, arj, lha, zip, zoo) являются комбинацией LZ77 и метода Хаффмана или арифметического кодирования. Наиболее используемые алгоритмы происходят от метода LZSS, описанного Джеймсом Сторером (James Storer) и Томасом Шиманski (Thomas Szymanski) в 1982. Основная идея LZSS: пара

(позиция_фрагмента_в_буфере_предыстории, длина_фрагмента) выводится в упакованные данные только если длина фрагмента больше заданной минимальной длины[5], иначе в упакованные данные выводится просто байт исходных данных и осуществляется снова поиск фрагмента. Для различения байтов и кодов LZSS резервирует 1 бит: если первый бит равен, например, 0, то следующие биты — пара (позиция_фрагмента_в_буфере_предыстории, длина_фрагмента), если первый бит равен 1, то следующие 8 бит — байт исходных данных.

2.5. Модификации алгоритма Лемпеля-Зива

Позднее был предложен алгоритм, комбинирующий идеи LZ77 и LZ78, и называемый LZFG. LZFG использует стандартный «буфер предыстории», но хранит данные в модифицированной древовидной структуре данных и выдает в качестве кода позицию текста в дереве. Т.к. LZFG вставляет в словарь целые фразы, он должен работать быстрее LZ78.

Еще одна известная модификация LZ78 связана с усовершенствованием алгоритма сброса (очистки) словаря при заполнении. В LZ78 словарь сбрасывается всякий раз, когда он заполняется. Сброс словаря резко ухудшает степень сжатия данных, до момента пока словарь снова не будет частично заполнен. В 1984 году Терри Уэлч (Terry Welch) предложил адаптивный сброс словаря. В этом случае при заполнении словаря сброс словаря не производится и новые фрагменты в словарь не добавляются, пока степень сжатия данных остается «достаточно» высокой. Степень «достаточности» определяется субъективно, но простейший вариант «достаточности»: пока степень сжатия не хуже, чем на момент заполнения словаря. В этом случае резко возрастает скорость компрессии, поскольку операция вставки новых фрагментов в словарь не производится.

Расширенным вариантом адаптивного сброс словаря является частичный сброс — удаление из словаря только части фрагментов, например тех, повторов которых в данных не обнаружено к моменту заполнения словаря. Этот вариант адаптивного сброса требует хранения дополнительной информации в словаре и усложняет конструкцию словаря, но приводит к выигрышу в степени сжатия.

Существуют и другие модификации метода.

---

[5] Минимальная длина обычно принимается равной размеру необходимому для записи кода (позиция_фрагмента_в_буфере_предыстории, длина_фрагмента).

3. ВОЗМОЖНАЯ РЕАЛИЗАЦИЯ МЕТОДА Лемпеля-Зива (LZ78)

3.1. Описание используемой модификации LZ78

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

Поскольку реализовывать базовый алгоритм LZ78 скучно, то ниже реализован немного модифицированный вариант. Введем два усовершенствования:

1) переменная длина кода;
2) начало с пустого словаря.

Первая модификация LZ78 (переменная длина кода) широко используется на практике. Дело в том, что количество бит, необходимых для записи кодов в упакованные данные монотонно возрастает с ростом словаря от минимального (9 *бит*) до максимального числа бит, определяемого размером словаря. Поэтому очевидно, что после сброса словаря коды можно записывать более короткими и только по мере заполнения словаря, можно увеличивать длину кода. Это усовершенствование улучшает компрессию для участка данных сразу после сброса словаря.

Второе усовершенствование (начало с пустого словаря) предложено еще Терри Уэлчем — автором алгоритма LZW. Суть усовершенствования заключается в отказе от заполнения словаря при сбросе всеми значениями байта (всеми однобайтовыми фрагментами). Можно заполнять словарь однобайтовыми фрагментами по мере поступления на вход новых значений байта, в этом случае (в комбинации с первым усовершенствованием) можно начинать запись кодов не с 9-и битовой длины, а с меньшей — 2-х битовой, что даст дополнительный выигрыш на участке данных сразу после сброса словаря.

Для реализации этих двух усовершенствований, в алгоритм придется ввести два дополнительных спецкода:

1) Увеличение длины кода, обозначим его «*IncCodeLength*», за этим кодом будет следовать 5-и битное число указывающее величину изменения длины. Максимальное приращение длины $2^5 = 32$ *бита*, что достаточно для словаря из 2^{32} элементов. Для 16-битных приложений можно было бы использовать 4-х битное число, но выигрыш невелик.
2) Новый байт, обозначим его «*NewByte*», за кодом будет следовать значение байта (8 *бит*), которого еще нет в словаре.

Таким образом всего будем использовать три спецкода:

1) *Reset*, присвоим ему значение 0;
2) *IncCodeLength* = 1;
3) *NewByte* = 2.

Начальная длина кода составит 2 бита.

Полное исследование и доказательство эффективности этих модификаций затруднено, ограничимся поэтому демонстрацией примера сжатия фрагмента, рассмотренного в пункте 2.3. Алгоритм описанный здесь, для тех же данных: «abcdabce·abcd·abceabcabcabcd·····abcaabcdabce·abceca», даст на выходе последовательность кодов: «r, 97(a), n, 98(b), n, 99(c), n, 100(d), i, 1, 4(ab), 7(c), n, 101(e), n, 32(·), i, 1, 11(abc), 9(d), i, 1, 16(·a), 6(bc), 13(e), 11(abc), 22(abca), 6(bc), 18(d·), 15(·), 26(··), 19(·ab), 7(c), 3(a), 17(abcd), 11(abc), 14(e·), i, 1, 32(abce), 29(ca)», где «r» обозначает код *Reset*, «n» - *NewByte* и «i» - *IncCodeLength*, в скобках «(…)» приведен фрагмент, соответствующий цифровому коду перед скобками, сами коды разделены запятыми.

Вычисление длины сжатых данных даст в этом случае 197 *бит* и сжатие в 2,11 раза, что лучше, чем получалось в пункте 2.3. Разумеется, для больших размеров данных выигрыш от указанных усовершенствований менее значителен.

3.2. Устройство словаря для LZ78

Словарь содержит известные фрагменты текста для замены их кодами. Как уже подчеркивалось, устройство словаря для метода Лемпеля-Зива наиболее критично. В сущности, весь процесс компрессии/декомпрессии представляет собой непрерывный поиск в словаре и заполнение словаря, и чем быстрее эти две операции осуществляются, тем быстрее будет работать программа.

Другой, не менее важный аспект — размер словаря, чем он больше, тем больше фрагментов в нем поместится, тем более длинные фрагменты можно в нем хранить и тем более эффективным будет сжатие.

Можно хранить фрагменты в виде строк:

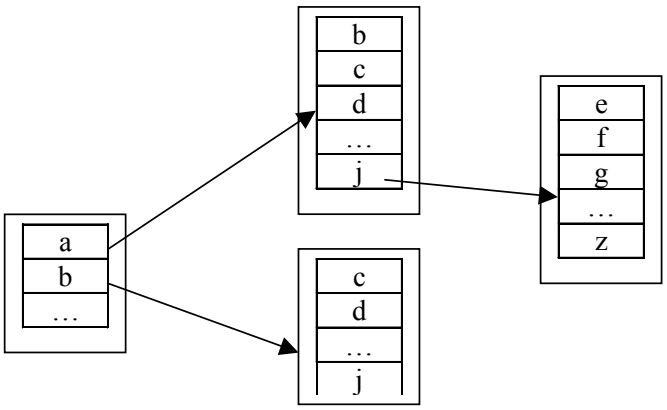
abc
abcd
abd
...

Назовем это «простым» словарем. Но «простой» словарь слишком расточителен в использовании памяти, т.к. повторы начала фрагментов неизбежны и не слишком хорош с точки зрения быстроты поиска — необходимо сравнение с множеством почти одинаковых фрагментов (хотя последнее преодолимо, для упорядоченного списка фрагментов).

Экономнее хранить фрагменты посимвольно в виде многосвязного дерева, например так, как это показано на рис. 3.1.

Начало фрагмента (первый символ) содержит ссылку на таблицу-продолжение, которая содержит список символов-продолжений фрагмента (все существующие вторые символы). Каждый следующий символ тоже может

ХРАНЕНИЕ ФРАГМЕНТОВ В ВИДЕ ДЕРЕВА



Приведенное дерево определяет 13 фрагментов (ячейки с «...» не учитывались): a, b, ab, ac, ad, aj, aje, ajf, ajg, ajz, bc, bd, bj.
Для хранения их в виде «простого словаря» необходимо 28 байт, при хранении в дереве — 13 байт. При увеличении числа ветвлений дерева, разница еще более возрастает.

Рис. 3.1

иметь таблицу-продолжение или не иметь, если продолжения фрагмента нет. И так далее до бесконечности.

Таблицу-продолжение можно тоже построить «просто» — в виде строки из 256 символов (байтов), но такое устройство также будет слишком расточительным с точки зрения расхода памяти. Следовательно, таблица-продолжение должна быть динамической (расширяемой на ходу) — для наращивания словаря фрагментов и упорядоченной — для ускорения поиска продолжения фрагмента в ней.

Указанным требованиям удовлетворяет таблица-продолжение в виде идеально сбалансированного двоичного дерева. Пример для таблицы-продолжения изображен на рис. 3.2. Поиск символа в ней займет не более $\log_2$(число_узлов_в_дереве) шагов, для максимального числа 256 узлов — это всего 8 шагов. Узлами дерева называют составляющие его элементы. Для сравнения, поиск в линейной таблице максимального размера в среднем требует 128 шагов[6].

Поиск в упорядоченном двоичном дереве осуществляется очень быстро. Двоичным дерево называется потому, что каждый вышележащий узел ссылается на два нижележащих. Идеально сбалансированным дерево называют потому, что слева и справа от любого узла находится одинаковое число нижележащих узлов. При четном общем числе узлов неизбежен дисбаланс на 1 узел, т.е. пра-

[6] Для таблицы в виде упорядоченного массива, применив для поиска метод деления отрезка пополам, тоже можно получить максимум 8 шагов, но реализовать таблицу-продолжение в виде массива практически невозможно.

вильнее сказать — число узлов слева и справа ниже любого узла различается не более чем на единицу.

Дерево считаем упорядоченным согласно правилу: справа (*Right*) от узла с байтом *b* находятся байты **большие** *b*, как это изображено на рис. 3.2.

Надо признать, что динамическая перебалансировка дерева при вставке в дерево новых узлов весьма трудоемкая операция, и выигрывая на скорости поиска, можно сильно проиграть на скорости вставки нового продолжения.

Таким образом словарь состоит из узлов (элементарных структур). Каждый узел должен содержать:

1) байт (символ);
2) ссылку на предшествующий байт — для восстановления фрагмента по коду;
3) ссылку на таблицу-продолжение — для записи продолжения фрагмента;
4) ссылку на правый следующий узел — для создания двоичного дерева;
5) ссылку на левый следующий узел — для создания двоичного дерева;
6) данные для балансировки дерева — доп. информация по балансировке двоичного дерева.

Кодом фрагмента является номер (*Index*) последнего узла, принадлежащего фрагменту, в массиве узлов.

ПРИМЕР ДВОИЧНОГО ИДЕАЛЬНО СБАЛАНСИРОВАННОГО ДЕРЕВА

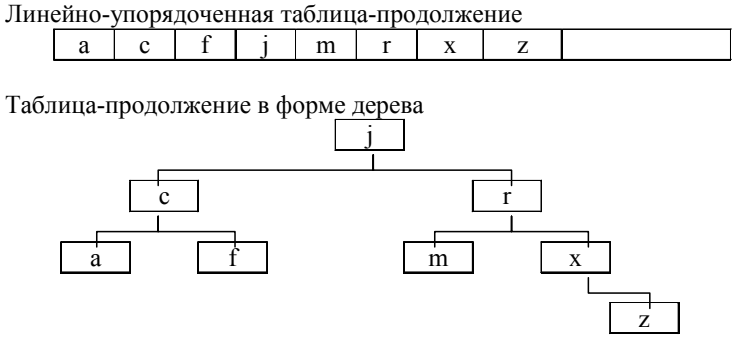


Рис. 3.2

Поскольку в сумме эти данные составят 9 *байт* для ссылки размером в слово (16 *бит*), то максимальный размер словаря для 16-и битной программы получится незначительным, около 65535/9 ≈ 7282 узлов, а может и еще меньше, т.к. могут потребоваться дополнительные данные. Для 32-битной программы размер словаря может быть существенно больше.

Для максимизации словаря в 16-битной программе можно разместить данные в виде отдельных массивов, хотя это усложнит операции с данными и ухудшит быстродействие. Ниже применено именно такое решение.

Программа размещает пять массивов (рис. 3.3):

1) *Bytes* — байт-код символа;

2) *Nodes* — ссылка на правый и ссылка на левый следующий узел (структура двоичного дерева);
3) *Flags* — флаги балансировки (данные для поддержания сбалансированности дерева);
4) *Successors* — ссылка на таблицу-продолжение (для записи продолжения фрагмента);
5) *Predecessors* — ссылка на предшествующий байт (для восстановления фрагмента по коду);

Узел идентифицируется по номеру (*Index*) в массиве узлов. Элементы вышеперечисленных массивов, имеющие одинаковый номер (*Index*), принадлежат одному узлу дерева. Определения типов *tBytesArray, tNodesArray* и др. можно найти в файле «LZTypes.pas».

Признаком отсутствия ссылки выберем значение равное НУЛЮ (*cNilIndex*), если в любую ячейку *Nodes*, *Successors* или *Predecessors* записан *cNilIndex*, то значит эта ячейка является конечной, и ни на что более не ссылается.

Кроме этого программа размещает описание размеров словаря и текущий размер словаря (*Descriptors*):

1) *Max* — размер словаря в узлах;
2) *FirstFree* - первый свободный узел словаря.

Данные описания размеров словаря также показаны на рис. 3.3.

В этом случае размер словаря для 16-битной адресации чуть более 16000 узлов. Лимитирует размер элемента массива *Nodes* (4 *байта*). Можно конечно разбить этот массив на 2 отдельных массива (тогда максимально возможный размер 32000 узлов), но в нашем случае (учебные задачи) — это излишнее усложнение.

Словарь до ~4 000 000 символов доступен в Delphi-варианте программы. В Delphi-варианте программы используются 32-битные индексы, вместо 16-и битных.

Уточним терминологию. Далее будут различаться термины: «Корень дерева» (*Root*) и «корневой узел дерева» (*RootNode*).

- «Корень дерева» (*Root*) — означает узел, для которого дерево является продолжением, он не принадлежит дереву, а является предшествующим узлом, для всех узлов дерева. Корень дерева неизменен для данного дерева.
- «Корневой узел дерева» (*RootNode*) — означает узел, являющийся корневым узлом дерева, он принадлежит дереву. Корневой узел дерева может изменяться при добавлении к дереву новых узлов.
- «Корень дерева (ссылка)» (*RootRef*) — означает ячейку памяти, в которой находится ссылка на «корневой узел дерева».

Соотношение между понятиями «Корень дерева» (*Root*) и «Корневой узел дерева» (*RootNode*):

RootNodeIndex = *Successors*[*RootIndex*];

RootIndex = *Predecessors*[*RootNodeIndex*];

RootIndex = *Predecessors*[любой *Index* для принадлежащего дереву *RootNodeIndex* узла].

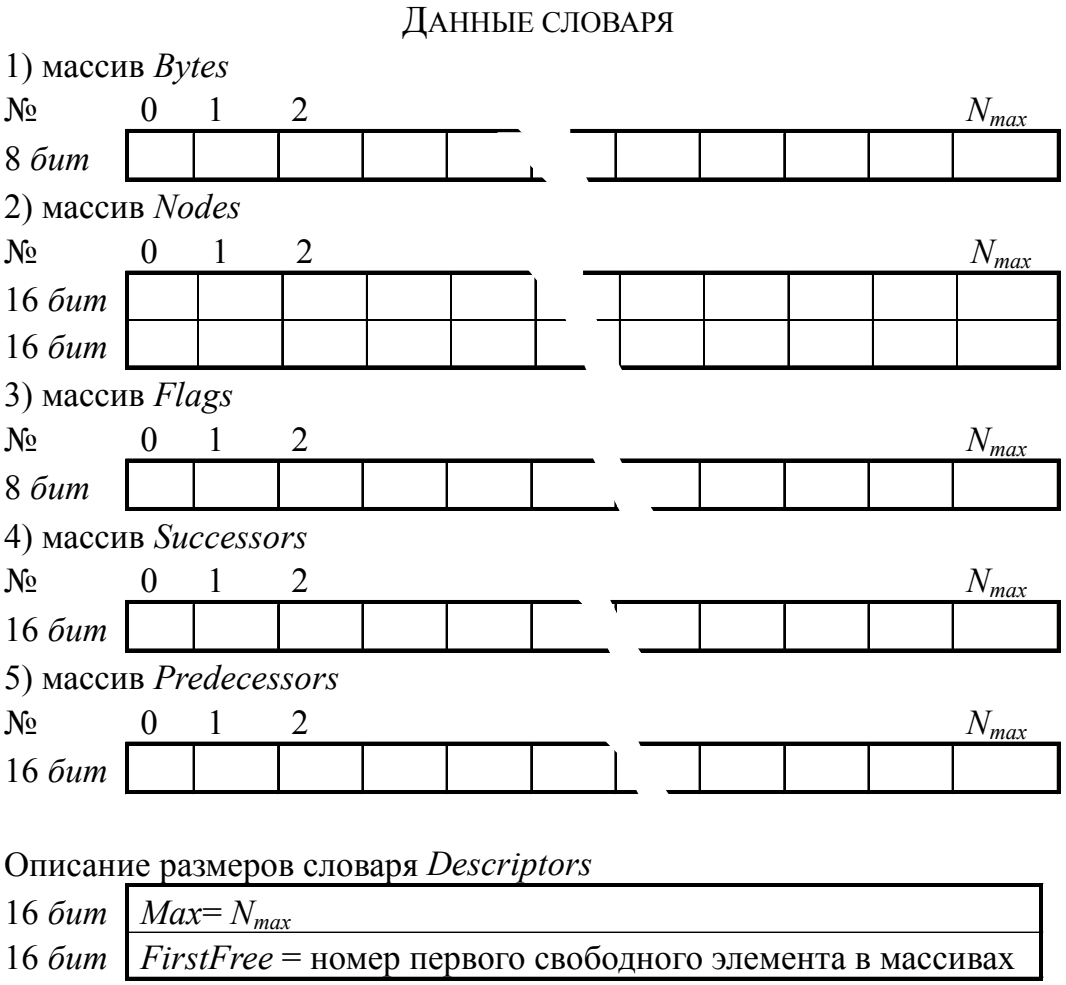


Рис. 3.3

Примечание. Массив *Successors* используется только при кодировании, а массив *Predecessors* — только при декодировании. Поэтому, имеется возможность использовать только один массив, но в этом случае при кодировании невозможно декодировать фрагмент, а при декодировании невозможен поиск в словаре. Выигрыш от такого решения — меньше необходимый размер памяти и быстрее работа декодировщика (в 2÷3 раза), т.к. при декодировании не нужно восстанавливать двоичное дерево.

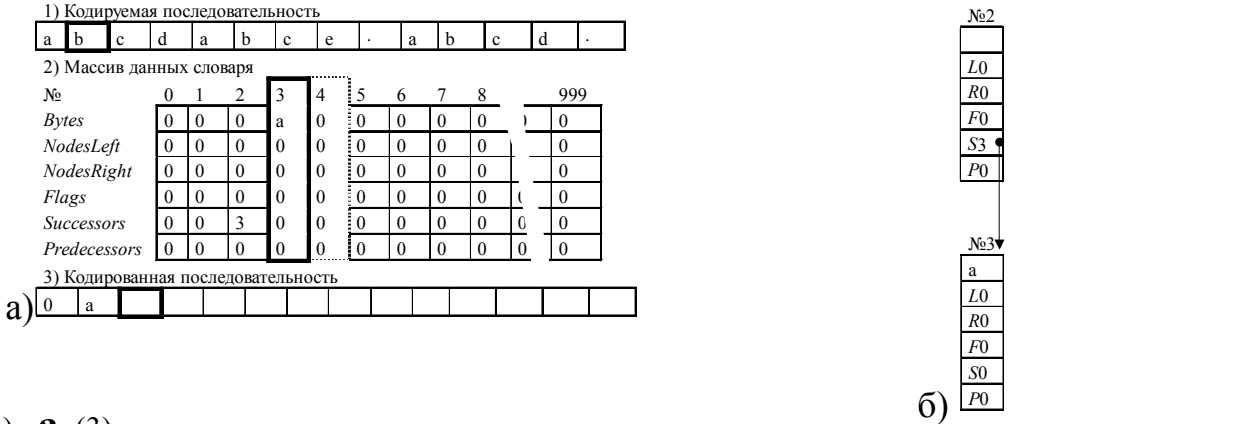
3.3. Пример заполнения словаря при кодировании

Рассмотрим работу словаря при кодировании первых семи символов примера из пункта 2.3: «abcdabce·abcd·abceabcabcabcd·····abcaabcdabce·abceca». Исходное состояние словаря изображено на рис. 3.4. Последующие состояния

словаря изображены на рис. 3.5÷рис. 3.11. Рисунки иллюстрируют работу словаря, так как она реализована в прилагаемой к методическим указаниям программе.

а) состояние массива данных; б) состояние словаря до кодирования первого символа и обозначения.

Рис. 3.4



в) «**a**»(3)

а) состояние массива данных словаря **после** кодирования символа;

б) граф состояния словаря после кодирования, стрелки графа обозначают ссылки ячейки откуда исходит ссылка на элемент массива, на который она указывает.

в) «» - фрагменты словаря (жирным шрифтом выделен текущее положение, курсивом – выведенный в упакованные данные код) в порядке добавления к словарю; (3) - код фрагмента.

Рис. 3.5

Кодирование второго символа

1) Кодируемая последовательность

a	b	c	d	a	b	c	e	·	a	b	c	d	·

2) Массив данных словаря

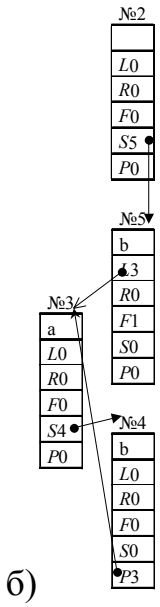
№	0	1	2	3	4	5	6	7	8		999
Bytes	0	0	0	a	b	b	0	0	0		0
NodesLeft	0	0	0	0	0	3	0	0	0		0
NodesRight	0	0	0	0	0	0	0	0	0		0
Flags	0	0	0	0	0	1	0	0	0		0
Successors	0	0	5	4	0	0	0	0	0		0
Predecessors	0	0	0	0	3	0	0	0	0		0

а)

3) Кодированная последовательность

0	a	2	b										

в) «a»(3), «ab»(4), «**b**»(5)



б)

Рис. 3.6

Кодирование третьего символа

1) Кодируемая последовательность

a	b	c	d	a	b	c	e	·	a	b	c	d	·

2) Массив данных словаря

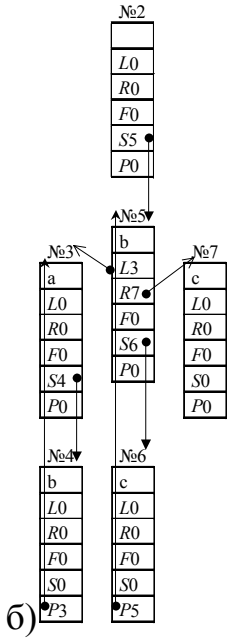
№	0	1	2	3	4	5	6	7	8		999
Bytes	0	0	0	a	b	b	c	c	0		0
NodesLeft	0	0	0	0	0	3	0	0	0		0
NodesRight	0	0	0	0	0	7	0	0	0		0
Flags	0	0	0	0	0	0	0	0	0		0
Successors	0	0	5	4	0	6	0	0	0		0
Predecessors	0	0	0	0	3	0	5	0	0		0

а)

3) Кодированная последовательность

0	a	2	b	2	c								

в) «a»(3), «ab»(4), «b»(5), «bc»(6), «**c**»(7)



б)

Рис. 3.7

Кодирование четвертого символа

1) Кодируемая последовательность

a	b	c	d	a	b	c	e	·	a	b	c	d	·

2) Массив данных словаря

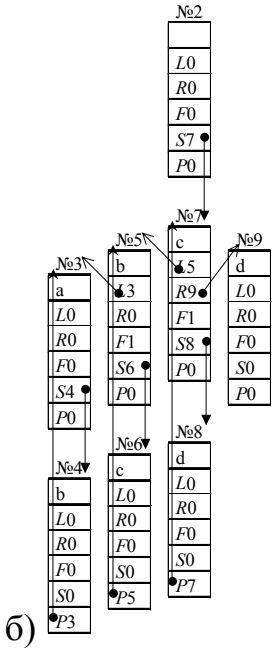
№	0	1	2	3	4	5	6	7	8	9	10
Bytes	0	0	0	a	b	b	c	c	d	d	0
NodesLeft	0	0	0	0	0	3	0	5	0	0	0
NodesRight	0	0	0	0	0	0	0	9	0	0	0
Flags	0	0	0	0	0	1	0	1	0	0	0
Successors	0	0	7	4	0	6	0	8	0	0	0
Predecessors	0	0	0	0	3	0	5	0	7	0	0

а)

3) Кодированная последовательность

0	a	2	b	2	c	2	d						

в) «a»(3), «ab»(4), «b»(5), «bc»(6), «c»(7), «cd»(8), «**d**»(9)



б)

Рис. 3.8

Кодирование пятого символа

1) Кодируемая последовательность

a	b	c	d	a	b	c	e	·	a	b	c	d	·

2) Массив данных словаря

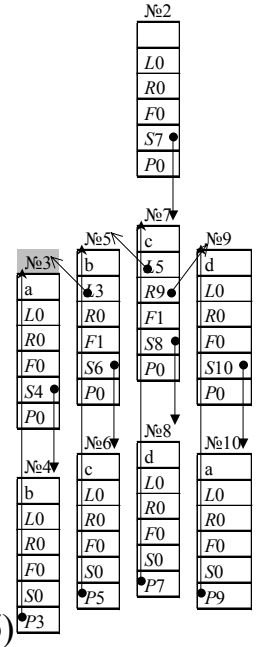
№	0	1	2	3	4	5	6	7	8	9	10	11
Bytes	0	0	0	a	b	b	c	c	d	d	a	0
NodesLeft	0	0	0	0	0	3	0	5	0	0	0	0
NodesRight	0	0	0	0	7	0	0	9	0	0	0	0
Flags	0	0	0	0	0	1	0	1	0	0	0	0
Successors	0	0	7	4	0	6	0	8	0	10	0	0
Predecessors	0	0	0	0	3	0	5	0	7	0	9	0

а)

3) Кодированная последовательность

0	a	2	b	2	c	2	d						

в) «**a**»(3), «ab»(4), «b»(5), «bc»(6), «c»(7), «cd»(8), «d»(9), «da»(10)



б)

Рис. 3.9

Кодирование шестого символа

1) Кодируемая последовательность

a	b	c	d	a	b	c	e	·	a	b	c	d	·

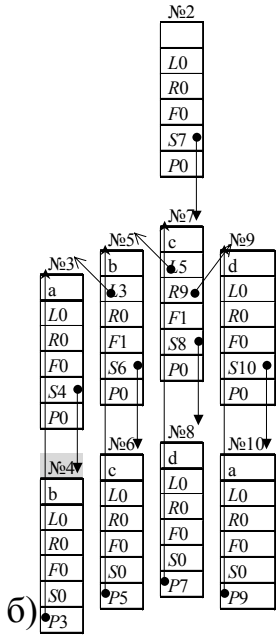
2) Массив данных словаря

№	0	1	2	3	4	5	6	7	8	9	10	11	12
Bytes	0	0	0	a	b	b	c	c	d	d	a	0	0
NodesLeft	0	0	0	0	0	3	0	5	0	0	0	0	0
NodesRight	0	0	0	0	0	0	0	9	0	0	0	0	0
Flags	0	0	0	0	0	1	0	1	0	0	0	0	0
Successors	0	0	7	4	0	6	0	8	0	10	0	0	0
Predecessors	0	0	0	0	3	0	5	0	7	0	9	0	0

а)

3) Кодированная последовательность

0	a	2	b	2	c	2	d						



б)

в) «a»(3), «**ab**»(4), «b»(5), «bc»(6), «c»(7), «cd»(8), «d»(9), «da»(10)

Рис. 3.10

Кодирование седьмого символа

1) Кодируемая последовательность

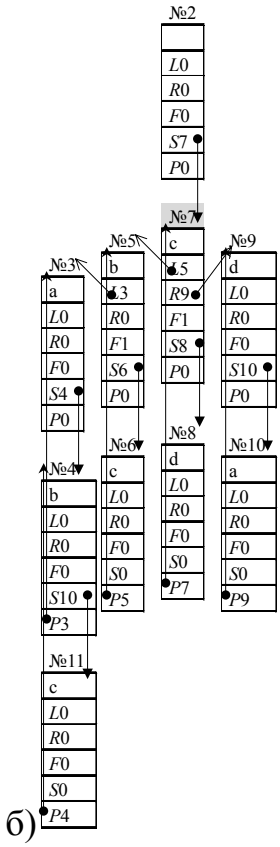
a	b	c	d	a	b	c	e	·	a	b	c	d	·

2) Массив данных словаря

№	0	1	2	3	4	5	6	7	8	9	10	11	12
Bytes	0	0	0	a	b	b	c	c	d	d	a	c	0
NodesLeft	0	0	0	0	0	3	0	5	0	0	0	0	0
NodesRight	0	0	0	0	0	0	0	9	0	0	0	0	0
Flags	0	0	0	0	0	1	0	1	0	0	0	0	0
Successors	0	0	7	4	11	6	0	8	0	10	0	0	0
Predecessors	0	0	0	0	0	0	5	0	7	0	9	4	0

3) Кодированная последовательность

0	a	2	b	2	c	2	d	1	1	4		



б)

в) «a»(3), «ab»(*4*), «b»(5), «bc»(6), «**c**»(7), «cd»(8), «d»(9), «da»(10), «abc»(11)

Рис. 3.11

3.4. Основные функции для работы со словарем

Исходный код функций работы со словарем доступен в файле «LZBase.pas», а описание типов данных словаря — в «LZTypes.pas».

Необходимы следующие основные функции:

1) *FindNodeInBinaryTree* — поиск узла в упорядоченном бинарном дереве.
2) *AddNodeToBinaryTree* — добавление узла в упорядоченное идеально сбалансированное бинарное дерево с сохранением балансировки.
3) *DecodeNode* — декодирование номера (кода) узла словаря в последовательность байт.
4) *AddNodeForDecode* — добавление узла в словарь для декодировки — не восстанавливает полную структуру словаря, заполняет только значение *Predecessors*. Функция очень проста и далее не рассматривается.

Рассмотрим первые три функции подробнее. Начнем с самой простой из них *FindNodeInBinaryTree.*

3.4.1. Поиск в двоичном упорядоченном дереве

Для поиска в дереве определена (см. «LZBase.pas») функция *FindNodeInBinaryTree*. Небольшие размеры этой функции делают ее работу вполне наглядной:

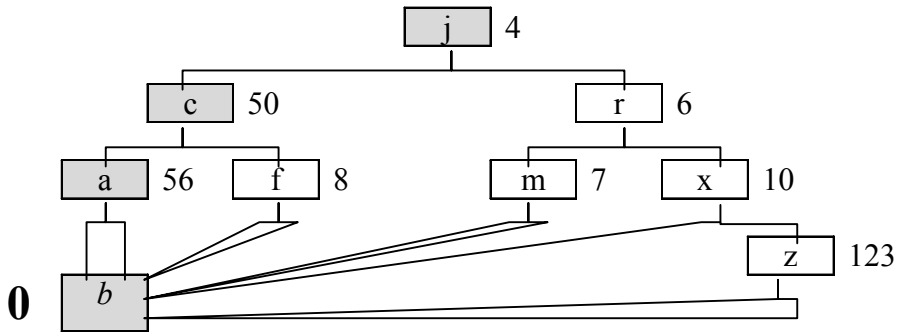
```
function FindNodeInBinaryTree({ принимает в качестве аргументов: }
           aByte:tByte;           { байт для поиска }
           aRootNodeIndex:tIndex; { номер корневого узла для дерева }
       var Dic:tDictionary        { ссылка на данные словаря }
):tIndex;
{ ВОЗВРАЩАЕТ:
  номер узла с байтом aByte для дерева с корнем aRootNodeIndex,
  иначе cNilIndex (ноль), если байт не найден. }

  var PNodes:tPNodesArray; b:tByte; PBytes:tPBytesArray;
begin
  { инициализация вспомогательных переменных - для ускорения работы }
  PNodes:=Dic.Nodes;
  PBytes:=Dic.Bytes;
  { установка барьера }
  PBytes^[cNilIndex]:=aByte;
  { значение байта в корневом узле дерева }
  b:=PBytes^[aRootNodeIndex];
  while (b<>aByte) do begin
    { корневой узел субдерева, где должен быть aByte }
    aRootNodeIndex:=PNodes^[aRootNodeIndex].LeftRight[aByte>b];
    { значение байта в корневом узле субдерева, где должен быть aByte }
    b:=PBytes^[aRootNodeIndex];
  end;
  { возвращаем значение индекса для найденного элемента }
  FindNodeInBinaryTree:=aRootNodeIndex;
end;
```

Поиск с барьером. Отслеживается одно событие — байт найден:

Dic.Bytes^[*aIndex*]=*aByte*;

тогда вернуть *aIndex*.

ИЛЛЮСТРАЦИЯ РАБОТЫ БАРЬЕРА ПРИ ПОИСКЕ В ДЕРЕВЕ



Рядом с прямоугольником, изображающим узел, приписан номер узла в массиве узлов. Номера узлов — это значения «для примера», кроме значения 0 — первый элемент массива узлов словаря. Закрашены узлы, с которыми будет проведено сравнение при поиске «b» в дереве.

Рис. 3.12

Поскольку для *cNilIndex* = 0 определен элемент в массиве *Dic.Nodes*, и одновременно ссылка на этот элемент является признаком конца поиска по событию «нет больше элементов в дереве», то можно уменьшить число операций сравнения при поиске, создав «барьер» в *Dic.Bytes*^[*cNilIndex*], рис. 3.12. На *Dic.Bytes*^[*cNilIndex*] ссылаются ВСЕ «пустые ссылки», поместив в этот элемент значение *aByte*, мы можем ограничиться проверкой *Dic.Bytes*^-[*aIndex*]=*aByte* с гарантией, что это равенство выполнится в ЛЮБОМ случае при поиске по дереву. Это произойдет либо в случае присутствия *aByte* в дереве, либо при достижении пустой ссылки (*aIndex=cNilIndex*). В обоих случаях произойдет правильный возврат значения: индекс элемента, если *aByte* есть в дереве, или *cNilIndex* = 0 если его там нет.

Вариант ассемблерной реализации *FindNodeInBinaryTree* можно найти в «LZBase.pas», но код на Паскале достаточно эффективен и быстр.

3.4.2. Вставка нового байта в двоичное дерево

Добавление узла в упорядоченное идеально сбалансированное бинарное дерево с сохранением балансировки — заметно более сложная задача:

```
function AddNodeToBinaryTree( { принимает в качестве аргументов: }
          aByte:byte;         { байт для вставки }
          ARootIndex:tIndex; { корень дерева }
      var Dic:tDictionary     { ссылка на данные словаря }
):tNodeIndex;
{ ВОЗВРАЩАЕТ:
   номер вставленного узла,
   иначе cNilIndex - нет свободного места в словаре. }

  var NewNodeIndex:tNodeIndex;
```

```
begin
  { получить свободный узел }
  NewNodeIndex:=AllocateNode(Dic, ARootIndex);
  { вернуть его номер в качестве результата выполнения функции }
  AddNodeToBinaryTree:=NewNodeIndex;
  { если свободный узел получен }
  if NewNodeIndex<>cNilIndex then begin
    { записать в него данные }
    Dic.Bytes^[NewNodeIndex]:=aByte;
    { и добавить узел в дерево }
    AddNode(NewNodeIndex, @Dic.Successors^[ARootIndex], Dic);
  end;
end;
```

На первый взгляд — не сложнее, чем *FindNodeInBinaryTree*, но данная функция использует две дополнительных функции: *AllocateNode* и *AddNode*. Если *AllocateNode* — это функция, которая присваивает начальные значения первому свободному в словаре узлу и помечает его как занятый — тоже проста (см. «LZBase.pas»), то *AddNode* — функция, которая вставляет новый узел в дерево и, при необходимости, балансирует дерево — намного больше и сложнее (~180 строк, вместо 10 строк в *FindNodeInBinaryTree*, см. «LZBase.pas»).

Разобрать здесь устройство и работу *AddNode* не представляется возможным. Если есть желание, то ознакомится с ее исходным кодом можно в файле в «LZBase.pas». Попробуйте разобраться с ее работой самостоятельно. Эффективность кода *AddNode* неочевидна, и вполне возможно, что эту процедуру можно написать лучше. Если у Вас есть желание и способность создать более эффективный (более быстрый) вариант *AddNode*, то см. задания повышенной сложности к этой лабораторной работе.

3.4.3. Декодирование фрагмента по словарю

Декодирование узла словаря в последовательность байт рассмотрим, чтобы лучше понять некоторые сложности процесса декодировки.

Это единственная процедура, использующая массив *Predecessors* для восстановления последовательности байт по коду фрагмента. Напомним, что код фрагмента — это индекс (номер) последнего узла, принадлежащего фрагменту, в массиве узлов словаря.

Проход по фрагменту в словаре возможен только от конца к началу, иначе пройти фрагмент невозможно. Если Вы полагаете иначе, то можете попробовать создать свой вариант *DecodeNode*.

```
function DecodeNode( { принимает в качестве аргументов: }
           ANodeIndex:tNodeIndex; { начальный (правильнее конечный) узел }
       var ABuffer;               { ссылка на буфер для декодированных данных }
           ABufferSize:tBufferIndex; { размер буфера }
     const Dic:tDictionary           { ссылка на данные словаря }
):tSignedBufferIndex;
{ ВОЗВРАЩАЕТ:
   значение >= 0 - декодировано до конца, значение = размеру данных в ABuffer;
   значение < 0  - недекодировано, значение = размер недостающего места в
                   буфере(ABufferSize - длина_фрагмента). }

  var
```

```
    i,j:tBufferIndex;
    b:tByte;
    PBytes:tPBytesArray;
    PPredecessors:tPLineReferenciesArray;
begin
{Инициализируем локальные указатели на данные дерева - это для ускорения работы}
  PBytes:=Dic.Bytes;
  PPredecessors:=Dic.Predecessors;
  { Декодируем, но в буфер символы попадают в обратном порядке  }
  i:=0;
  { пока не кончился фрагмент и в буфере есть свободное место }
  while (ANodeIndex>cRootIndex) and (i<ABufferSize) do begin
    { помещаем в свободный байт буфера очередной байт фрагмента }
    tBuffer(ABuffer)[i]:=PBytes^[ANodeIndex];
    { изменяем номер свободного байта в буфере}
    Inc(i);
    { устанавливаем номер очередного узла фрагмента }
    ANodeIndex:=PPredecessors^[ANodeIndex];
  end;
  if (ANodeIndex<=cRootIndex) then begin
  { декодировали успешно }
    DecodeNode:=i;
    { инвертируем порядок символов в буфере  }
    Dec(i); j:=0;
    while (j<i) do begin
      b:=tBuffer(ABuffer)[i];
      tBuffer(ABuffer)[i]:=tBuffer(ABuffer)[j];
      tBuffer(ABuffer)[j]:=b;
      Dec(i); Inc(j);
    end;
  end else begin  { НЕдекодировали - недостаточно места в буфере }
    { считаем, сколько не хватает }
    i:=0;
    repeat
      Inc(i);
      ANodeIndex:=PPredecessors^[ANodeIndex];
    until (ANodeIndex=cRootIndex);
    { возвращаем недостающий размер }
    DecodeNode:=-i;
  end;
end;
```

Основная проблема, о которой необходимо помнить, — слишком длинный фрагмент может не поместиться в буфер и при упаковке/распаковке следует согласовывать максимальную длину фрагмента и длину буфера.

Вариант ассемблерной реализации *DecodeNode* можно найти в «LZBase.pas», код на ассемблере более эффективен (быстр) за счет оптимизации некоторых операций.

3.5. Процедура кодировки LZ78

Кодировка в алгоритме Лемпеля-Зива отличается от метода Хаффмана принципиально. В методе Хаффмана необходимо сначала обработать все данные — вычислить частоту вхождения для символов, а только потом их можно закодировать более экономно (упаковать), т.е. данные должны быть просмотрены дважды[7]. В алгоритме Лемпеля-Зива данные кодируются за однократный

[7] Существует адаптивный вариант метода Хаффмана.

просмотр. Поэтому основная процедура кодировки принимает в качестве аргумента ОДИН БАЙТ, а не массив байтов как это было в методе Хаффмана. Для упаковки данных любой длины, эта процедура должна быть вызвана с каждым из байтов данных последовательно.

Процедура упаковки оформлена в виде метода объекта *LZDictionary* (см. «LZDic.pas», «LZDic1.pas» и «LZDic0.pas») и называется *ByteToCodes*. Исходный код этой процедуры находится в модуле «LZDic1.pas».

Чтобы выполнить ОСНОВНОЕ задание к этой лабораторной работы — необходимо понимать логику работы этой процедуры, поэтому рассмотрим процесс кодирования несколько подробнее (см. также пункт 2.3).

Функция *ByteToCodes* определена так

```
function tLZDictionary.ByteToCodes(
            aByte:tByte;
      var   aCodes:tLZCodesArray
):tCodesCount;
```

и осуществляет упаковку ввода. Ей передают очередной байт *aByte* из входных данных и она возвращает коды (от 0 до 5 шт.) в массиве *aCodes*. Тип *tLZCodesArray* декларирован в «LZTypes.pas» так

```
tLZCodesArray=record
    Code:tCodeArray;
    Length:tCodeLengthArray;
end;
```

где

```
  { счетчик кодов }
  tCodesCount=0..6;
  { индекс кодов }
  tCodesIndex=0..5;
  { массив значений кодов }
  tCodeArray=array[tCodesIndex] of tIndex;
  { массив длин кодов }
  tCodeLengthArray=array[tCodesIndex] of tIndexBitsCounter.
```

Функция *ByteToCodes* возвращает количество кодов, если возвращено

- *значение* = 0, то фрагмент найден и выводить код(ы) не надо, в переменой *aCodes* ничего не изменяется;
- *значение* > 0, то фрагмент не найден, в переменой *aCodes* возвращаются коды и их длины в количестве, равном возвращенному *значению*.

В переменой *aCodes* могут находиться до 5-и кодов, как это перечислено в табл. 3.1.

Таблица 3.1

Коды возврата *ByteToCodes*

Кол-во кодов	Коды
1	код фрагмента

Кол-во кодов	Коды
2	код "ОЧИСТИТЬ СЛОВАРЬ" +байт
2	код "НОВЫЙ БАЙТ"+байт
3	код "УВЕЛИЧИТЬ ДЛИНУ КОДА"+число бит увеличения кода+код фрагмента
3	код фрагмента +код "ОЧИСТИТЬ СЛОВАРЬ"+байт
5	код "УВЕЛИЧИТЬ ДЛИНУ КОДА"+число бит увеличения кода+код фрагмента+код "ОЧИСТИТЬ СЛОВАРЬ"+байт

В промежутках между вызовами процедура *ByteToCodes* хранит свои данные в структуре словаря (см. пункт 3.2) и дополнительных переменных (см. определение типа *tLZDictionary* в «LZDic0.pas»):

```
prDicFlags:tFlags;            { внутренние флаги }
prDic:tDictionary;            { данные словаря }
prLastCode:tIndex;            { последний код = код текущего фрагмента }
prLastDecodedFragmentFirstByte:tByte;  { первый байт последнего декодированного
                                         фрагмента (только для декодировки) }
prCodeLength:tCodeLength;    { текущая длина кода}
prMaxCode:tNodeIndex;        { максимальное значение кода для текущей длины кода
                               (только при кодировании) }
prFragmentLength:tIndex;     { длина текущего фрагмента }
prFragmentLengthLimit:tIndex; { максимально допустимая длина фрагмента }
prCurrentMaxFragmentLength, { достигнутый максимум длины фрагмента для текущего
                              словаря (только при кодировании)}
prCurrentAbsMaxFragmentLength:tIndex; { максимум длины фрагмента за сессию
                                        кодирования  (только для кодировки) }
prCompressedSize:tLZCompressedSize; { упакованный размер данных (только при
                                      кодировании) }
prInitialSizeInBytes:tInt;    { исходный размер данных  (только при кодировании)}
prResetCount:tUINT;           { число сбросов словаря (только при кодировании)}
prSize,prNotSize:tNodeIndex; { размер словаря и признак инициализированности
                               объекта (используется только при инициализации
                               объекта) }
```

определения типов см. «ComTypes.pas», «LZTypes.pas» и «LZDic0.pas».

При вызове *ByteToCodes* должна выполнить следующие операции.

<table>
<tr><td colspan="4">Увеличить счетчик числа байт, переданных процедуре упаковки (prInitialSizeInBytes).</td></tr>
<tr><td colspan="4">Провести поиск продолжения aByte для текущего фрагмента в словаре (prLastCode = конец текущего фрагмента).</td></tr>
<tr><td colspan="4">Проверить, найдено ли продолжение фрагмента?</td></tr>
<tr><td>Найдено.</td><td colspan="3">Не найдено — фрагмент закончился.</td></tr>
<tr><td rowspan="2">Запомнить текущий фрагмент в prLastCode.</td><td colspan="3">Добавляем новое значение байта к корню словаря?</td></tr>
<tr><td>ДА</td><td colspan="2">НЕТ — добавление нового байта к некорневому узлу.</td></tr>
<tr><td>Вернуть 0.</td><td>Записать код «СБРОС</td><td colspan="2">Возвращать код?</td></tr>
<tr><td></td><td></td><td>ДА — возвращать код надо.</td><td>НЕТ — воз-</td></tr>
</table>

<table>
<tr><td rowspan="11"></td><td>СЛОВАРЯ» в aCodes.</td><td colspan="3"></td><td rowspan="4">вращать код не надо, предыдущий код «НОВЫЙ БАЙТ».</td></tr>
<tr><td>Записать новый байт в aCodes.</td><td colspan="3">Код превышает текущий максимальный размер кода?</td></tr>
<tr><td rowspan="2">Запомнить текущий фрагмент в prLastCode.</td><td>ДА — записать код</td><td rowspan="2">НЕТ</td></tr>
<tr><td>«УВЕЛИЧИТЬ ДЛИНУ КОДА» и размер увеличения длины в aCodes.</td></tr>
<tr><td>Вернуть число кодов (2).</td><td colspan="3">Записать код фрагмента в aCodes.</td></tr>
<tr><td rowspan="10"></td><td colspan="5">Длина фрагмента не превосходит максимально допустимую?</td></tr>
<tr><td colspan="4">ДА</td><td rowspan="9">НЕТ</td></tr>
<tr><td colspan="4">Добавить в словарь продолжение фрагмента.</td></tr>
<tr><td colspan="4">Свободное место в словаре закончилось?</td></tr>
<tr><td colspan="2">ДА</td><td colspan="2">НЕТ</td></tr>
<tr><td colspan="2">Записать код «СБРОС СЛОВАРЯ» в aCodes.</td><td colspan="2" rowspan="3">Увеличить текущую длину фрагмента prFragmentLength на 1</td></tr>
<tr><td colspan="2">Сбросить словарь.</td></tr>
<tr><td colspan="2">Записать байт aByte в aCodes.</td></tr>
<tr><td colspan="5">Установить словарь на начало (корень) для поиска фрагмента.</td></tr>
<tr><td colspan="5">Есть в словаре начало фрагмента aByte?</td></tr>
<tr><td colspan="2">ДА</td><td colspan="3">НЕТ</td></tr>
<tr><td colspan="2">Запомнить текущий фрагмент в prLastCode.</td><td colspan="3">Добавить байт aByte к корневому узлу словаря.</td></tr>
<tr><td rowspan="3">Вернуть число кодов.</td><td colspan="3">Свободное место в словаре закончилось?</td></tr>
<tr><td>ДА</td><td>НЕТ</td></tr>
<tr><td>Записать код «СБРОС СЛОВАРЯ» в aCodes.</td><td>Записать код «НОВЫЙ БАЙТ» в</td></tr>
<tr><td></td><td>Сбросить сло-</td></tr>
</table>

<table>
<tr><td rowspan="4"></td><td rowspan="4"></td><td rowspan="4"></td><td>варь.</td><td>aCodes.</td></tr>
<tr><td colspan="2">Записать байт aByte в aCodes.</td></tr>
<tr><td colspan="2">Вернуть число кодов.</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td></td><td colspan="4">Вычислить увеличение размера кодированных данных и запомнить в prCompressedSize.</td></tr>
<tr><td colspan="5">ВЫХОД из процедуры.</td></tr>
</table>

3.6. Процедура завершения кодировки LZ78

Конец кодируемых данных может присутствовать в словаре, тогда возврат *ByteToCodes* для последнего байта данных будет 0 и код последнего фрагмента не попадет в кодированные данные. При завершении кодирования данных необходимо принудительно вернуть код последнего фрагмента, иначе данные будут усечены.

Для избежания такой ситуации, при завершении кодирования следует вызвать *GetLastCodes*, которая возвратит оставшиеся коды. Для этого придется написать дополнительную процедуру *GetLastCodes*, которая вернет код для последнего фрагмента. Функция *GetLastCodes* определена так

```
function  GetLastCodes(var aCodes:tLZCodesArray):tCodesCount;
```

она возвращает число кодов и коды в массиве *aCodes*.

Процедура упаковки оформлена в виде метода объекта *tLZDictionary* (см. «LZDic.pas», «LZDic1.pas» и «LZDic0.pas») и называется *GetLastCodes*. Исходный код этой процедуры находится в модуле «LZDic1.pas».

При вызове *GetLastCodes* должна выполнить следующие операции.

<table>
<tr><td colspan="4">Текущий код это код корня словаря?</td></tr>
<tr><td>ДА</td><td colspan="3">НЕТ</td></tr>
<tr><td rowspan="7">Вернуть 0.</td><td colspan="3">Возвращать код?</td></tr>
<tr><td colspan="2">ДА — возвращать код надо.</td><td rowspan="3">НЕТ — возвращать код не надо, предыдущий код «НОВЫЙ БАЙТ».</td></tr>
<tr><td colspan="2">Код превышает текущий максимальный размер кода?</td></tr>
<tr><td>ДА — записать код</td><td>НЕТ</td></tr>
<tr><td>«УВЕЛИЧИТЬ ДЛИНУ КОДА» в aCodes.</td><td></td><td rowspan="3">Вернуть 0.</td></tr>
<tr><td colspan="2">Записать код фрагмента в aCodes.
Вернуть число кодов.</td></tr>
<tr><td colspan="2">Вычислить увеличение размера кодированных данных и запомнить в prCompressedSize.</td></tr>
</table>

Процедура *GetLastCodes* вызывается однократно (при завершении кодирования данных) и использует те же самые структуры данных, что и *ByteToCodes*.

3.7. Процедура декодировки LZ78

При декодировке данных необходимо одновременно восстанавливать словарь, как это описано в пункте 2.3. Собственно декодировку фрагмента по коду и словарю производит процедура *DecodeNode*, описанная в пункте 3.4.3.

Основная задача декодировщика — восстанавливать словарь в точности также, как словарь заполнялся при кодировании.

Для этого реализована процедура *CodeToBytes*, которая восстанавливает байты фрагмента по коду и одновременно пополняет словарь. Функция *CodeToBytes* определена так

```
function  CodeToBytes(
                      ACode:tIndex;
                  var ABuffer;
                      ABufferSize:tBufferIndex;
):tSignedBufferIndex.
```

Ей передают очередной код *ACode* из кодированных данных, указатель на буфер *ABuffer* для декодированных данных, размер буфера *ABufferSize*, указатель на переменную для возврата размера данных *ADataSize* и она возвращает:

- *значение* ≥ 0, если код декодирован, и байты данных помещены в массив *ABuffer*, тогда *значение* равно числу байт, записанных в *ABuffer*;
- *значение* < 0, если код недекодирован, *значение* равно размеру недостающего пространства в *ABuffer*.

Если возвращено значение меньшее нуля, то *ACode* считается недекодированным, словарь не изменяется, и небходимо повторно вызвать *CodeToBytes*(*ACode*, ...) с новым буфером достаточного размера.

Процедура распаковки оформлена в виде метода объекта *tLZDictionary* (см. «LZDic.pas», «LZDic1.pas» и «LZDic0.pas») и называется *CodeToBytes*. Исходный код этой процедуры находится в модуле «LZDic1.pas».

В промежутках между вызовами процедура *CodeToBytes* хранит свои данные в структуре словаря (см. пункт 3.2) и дополнительных переменных, см. определение типа *tLZDictionary* в «LZDic0.pas» и пункт 3.5.

При вызове *CodeToBytes* должна выполнить следующие операции:

<table>
<tr><td colspan="3">Предыдущий код «НОВЫЙ БАЙТ» или «СБРОС СЛОВАРЯ»?</td></tr>
<tr><td>ДА</td><td colspan="2">НЕТ</td></tr>
<tr><td>Вернуть код как байт в буфере. Длина данных aDataSize=1.</td><td colspan="2">Предыдущий код «УВЕЛИЧИТЬ ДЛИНУ КОДА»?</td></tr>
<tr><td>Сбросить состояние: предыду-</td><td>ДА</td><td>НЕТ</td></tr>
</table>

<table>
<tr><td colspan="2">щий код «НОВЫЙ БАЙТ» или «СБРОС СЛОВАРЯ».</td><td></td><td colspan="2"></td></tr>
<tr><td colspan="2">Текущий фрагмент корень словаря?</td><td>Увеличить текущую длину кода на значение aCode.</td><td colspan="2">ACode равен «СБРОС СЛОВАРЯ» или «НОВЫЙ БАЙТ» или «УВЕЛИЧИТЬ ДЛИНУ КОДА»?</td></tr>
<tr><td>ДА</td><td>НЕТ</td><td rowspan="4">Сбросить состояние:
предыдущий код

«УВЕЛИЧИТЬ ДЛИНУ КОДА».</td><td>ДА</td><td>НЕТ — это код фрагмента</td></tr>
<tr><td></td><td>Добавить продолжение фрагмента в словарь.</td><td>Запомнить код.</td><td rowspan="3">Декодировать код.
см. ниже</td></tr>
<tr><td colspan="2">Добавить продолжение фрагмента к корню словаря.</td><td rowspan="2">Сбросить словарь, если код «СБРОС СЛОВАРЯ».</td></tr>
<tr><td colspan="2">Запомнить байт как первый байт последнего декодированного фрагмента.</td></tr>
<tr><td colspan="5">ВЫХОД</td></tr>
</table>

Операция декодировки кода должна обрабатывать особый случай (см. пункт 2.3 и табл. 2.2):

<table>
<tr><td colspan="3">Декодировать код.</td></tr>
<tr><td colspan="3">Код есть в словаре?</td></tr>
<tr><td colspan="2">ДА</td><td>НЕТ — обработка особого случая</td></tr>
<tr><td colspan="2">Декодировать.</td><td rowspan="3">Добавить в словарь к текущему фрагменту продолжение в виде начала текущего (последнего декодированного) фрагмента[8].</td></tr>
<tr><td colspan="2">Длина фрагмента меньше максимальной?</td></tr>
<tr><td>ДА</td><td>НЕТ</td></tr>
<tr><td>Добавить продолжение фрагмента в словарь.</td><td></td><td>Декодировать.</td></tr>
<tr><td colspan="3">Запомнить первый символ декодированного фрагмента.</td></tr>
<tr><td colspan="3">Запомнить текущий фрагмент.</td></tr>
<tr><td colspan="3">Запомнить длину текущего фрагмента.</td></tr>
<tr><td colspan="3">ВЫХОД</td></tr>
</table>

Процедура декодировки может быть заметно ускорена, если вспомнить, что массив *Successors* и двоичное дерево необходимы только при кодировании, и восстанавливать их при декодировании нет необходимости. Поэтому, имеется возможность использовать только один массив *Predecessors*. В таком случае, при декодировании невозможен поиск в словаре. Выигрыш от такого решения

[8] При реализации декодировки так, как описано выше: возможен отказ от декодирования из-за нехватки размера буфера. Перед тем как пополнить словарь, следует убедится, что размер буфера достаточен для декодировки. Иначе словарь будет пополнен, но декодировка не пройдет (будет возвращено значение не 0), тогда при следующей попытке декодировать этот же код будет ОШИБОЧНО предпринята попытка снова пополнить словарь.

значительный — работа декодировщика быстрее в 2÷3 раза, если не восстанавливать двоичное дерево[9].

4. ЗАДАНИЕ ДЛЯ САМОСТОЯТЕЛЬНОГО ВЫПОЛНЕНИЯ

4.1. Краткое описание тестовой программы для метода LZ78

Для выполнения лабораторной работы вам предоставлен работоспособный программный код, реализующий алгоритм LZ78, 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.

Программа состоит из следующих частей.

1) Программных модулей, реализующих LZ78:
   - COMTYPES.PAS — общие определения типов,
   - LZTYPES.PAS — определения типов данных для LZ78,
   - LZBASE.PAS — поиск в словаре, вставка продолжения фрагмента в словарь и декодирование фрагмента по словарю LZ78,
   - LZDIC0.PAS — вспомогательные процедуры кодировки/декодировки,
   - LZDIC1.PAS — основные процедуры кодировки/декодировки,
   - LZCODEC.PAS — кодировка/декодировка буфера в ОЗУ;
   - LZBITBUF.PAS — битовый буфер для работы с порциями бит не кратными 8.
2) Программных модулей, реализующих сжатие файла:
   - FILEBUF.PAS — буфер для чтения/записи файла,
   - LZFILE.PAS — сжатие файла методом LZ78.
3) Программы сжатия файла:
   - FCMDLINE.PAS — разбор командной строки программы,
   - PRGFUNCS.PAS — вспомогательные функции для программы,
   - LZFTEST.PAS — главная программа упаковки/распаковки файла,

[9] Прдоставленная программа может быть перекомпилирована в двух вариантах: 1) с частичным восстановлением словаря и 2) полным восстановлением словаря при декодировке. По-умолчанию компилируется вариант 1. Для включения полного восстановления словаря удалите определение символа условной компиляции «Fast1» и полностью перестройте (build) программу.

- TEST.PAS, TEST2.PAS, TESTCDC.PAS, TESTDIC.PAS — тестирующие программы для различных модулей метода LZ78 (только для BP 7.0 и работоспособность этих программ не гарантирована).

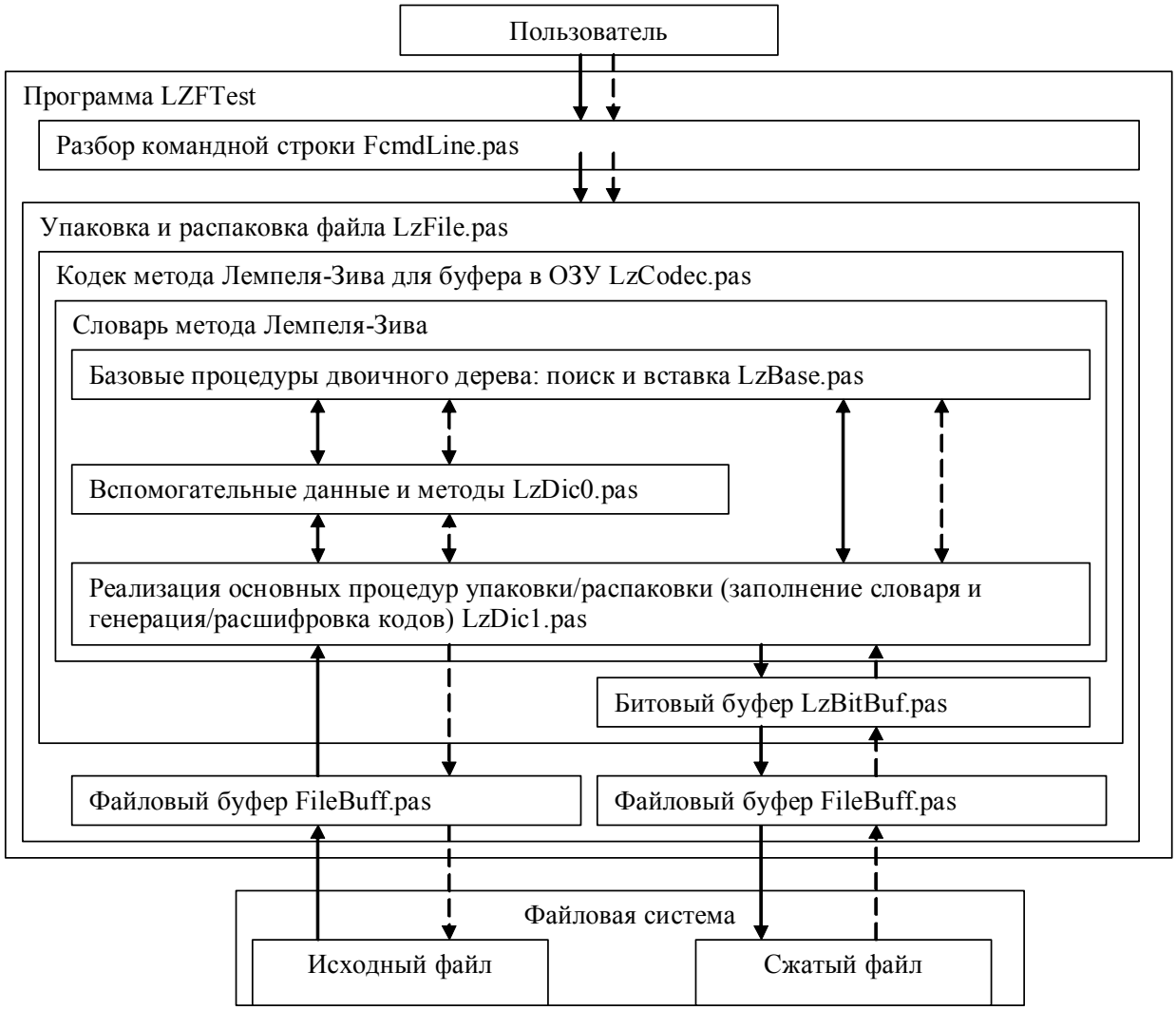
4) Вспомогательных программных модулей:
   - LZMISCFS.PAS — вспомогательные функции для LZ78,
   - UMISCFUN.PAS — вспомогательные функции для программы,
   - xStrings.pas — вспомогательные функции работы со строками,
   - LZTESTS.PAS — вспомогательные функции отладки для LZ78 (только для BP 7.0).

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

- «Сжатие данных. Метод Лемпеля-Зива» — основная папка, здесь находятся:
  - «Компрессия данных или измерение и избыточность информации. Метод Лемпеля -Зива.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.
    - «LZ78» — модули алгоритма Лемпеля-Зива 78, 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.

Схема взаимодействия модулей для программы LZFTest показана на рис. 4.1, на рисунке приведены только основные модули.

СХЕМА ВЗАИМОДЕЙСТВИЯ МОДУЛЕЙ В ТЕСТОВОЙ ПРОГРАММЕ LZFTEST



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

Рис. 4.1

4.2. КОМПИЛЯЦИЯ ТЕСТОВОЙ ПРОГРАММЫ В BP 7.0

Для получения возможности модифицировать код программы в BP 7.0:

1) Скопируйте папку «Сжатие данных. Метод Лемпеля-Зива» на локальный диск в любое место.
2) Запустите «Far» или «Norton Commander» и переместитесь в папку «Сжатие данных. Метод Лемпеля-Зива\Программа».

3) Запустите файл «RunMe.bat» — в системе появится новый диск «P:»[*] .
4) Переместитесь на диск P: — в действительности это папка «Сжатие данных. Метод Лемпеля-Зива\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 «LZFTest.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 методом LZ78 (по умолчанию упаковывается файл «TEST.txt» в файл «TEST._tx»).
2) Распаковывать упакованный методом LZ78 файл (по умолчанию распаковывается файл «TEST._tx» в файл «TEST.!tx»).
3) Упаковывать методом LZ78, сразу распаковывать и сравнить результат распаковки и оригинальный файл (см. рис. 4.2).

Дополнительно вычисляются некоторые характеристики компрессии.

Программа предназначена для тестирования метода LZ78 на реальных данных (файлах).

Программа дает представление о работе архиваторов и может быть использована для сравнения эффективности сжатия LZ78 с коммерческими архиваторами.

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

```
>LZFTEST.exe
```

4.3. Компиляция тестовой программы в Delphi 5.0

Для получения возможности модифицировать код программы в Delphi 5.0:

1) Скопируйте папку «Сжатие данных. Метод Лемпеля-Зива» на локальный диск в любое место.
2) Переместитесь в папку «Сжатие данных. Метод Лемпеля-Зива\Программа\Win - Delphi5».

---

[*] 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 LZFTEST.pas.

3) Щелкните (дважды) мышкой на файле «LZFTest.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 «LZFTest.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 методом LZ78 (по умолчанию упаковывается файл «TEST.txt» в файл «TEST._tx», расположенные в той же папке, где и «LZFTest.exe»).
2) Распаковывать упакованный методом LZ78 файл (по умолчанию распаковывается файл «TEST._tx» в файл «TEST.!tx»).
3) Упаковывать методом LZ78, сразу распаковывать и сравнить результат распаковки и оригинальный файл (см. рис. 4.2).

Дополнительно вычисляются некоторые характеристики компрессии.

Программа предназначена для тестирования метода LZ78 на реальных данных (файлах).

Программа дает представление о работе архиваторов и может быть использована для сравнения эффективности сжатия LZ78 с коммерческими архиваторами.

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

```
>LZFTEST.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 *LZFTEST*

```
Программа LZFileTest запущена (real mode)...
 Разбор командной строки...
 Завершен разбор командной строки.
  Размер словаря, симв.: 16366, макс.длина фрагмента: 16366
 Упаковка: "test.txt" -> "test._tx" (повтор, раз: 1)... завершено.
  Время на 1 упаковку, мс: 1705.0
  Отношение (Размер упакованного файла)/(Размер исходного): 0.2377
  Сбросов словаря: 2
  Макс. длина фрагмента: 55
 Распаковка: "test._tx" -> "test.!tx" (повтор, раз: 1)... завершено.
  Время на 1 распаковку, мс: 825.0
 Проверка. Сравнение "test.txt" и "test.!tx"... завершено.
  Правильно.
программа LZFileTest завершена.
```

Рис. 4.2

4.4. 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

1. Научиться компилировать программу и запускать тестовую программу, см. пункты 4.2 и 4.3.
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 на менее 1% по сравнению с готовым компилированным вариантом тестовой программы LZFTest.exe, который предоставлен вам вместе с исходным кодом. Этот вариант откомпилирован с параметрами, обеспечивающими максимальное быстродействие. Сравнение проводится при равных размерах словаря и максимального фрагмента. 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 LZ78 не лучшим образом реализует алгоритм и вы можете самостоятельно написать лучший вариант, то Вам предоставляется это право (на любом языке, кроме ассемблера).
2. Единственное требование — создать аналог «LZFTest.pas» для сравнения с примером программы.

Вариант 3*

1. Реализовать иную конструкцию словаря для LZ78 (процедуры модуля LZBase). Например, использовать для таблицы-продолжения линейный список (лучше упорядоченный), вместо двоичного дерева.
2. Сравнить быстродействие для вашего словаря с таблицей-продолжением в форме бинарного дерева. Сделать выводы о предпочтительной конструкции словаря.

Вариант 4*

1. Реализовать усовершенствование LZSS применительно к LZ78, т.е. модифицировать программу так, чтобы она выводила в упакованные данные коды, только если код короче, чем кодируемый фрагмент, иначе выводила бы кодируемый фрагмент как последовательность байт исходных данных.
2. Сравнить эффективность (степень сжатия и быстродействие) усовершенствования LZSS с исходным LZ78.

---

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

Вариант 5*

1. Спроектировать данные, описать алгоритм и написать процедуры для метода LZ77. В качестве образца использовать код программы, демонстрирующей сжатие данных методом LZ78, см. «LZDic0.pas» и др.
2. Проверить работоспособность метода LZ77.
3. Сравнить быстродействие и эффективность[10] метода LZ77 с методом LZ78.

Вариант 6*

1. Подробно описать алгоритм для любого метода сжатия без потерь (**кроме метода Хаффмана и LZ78**) с конкретным примером ручного вычисления для небольшой порции данных (см., например, пункт 2.3). 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 данных методом LZ78.
3. Попробовать проверить работоспособность метода.
4. Попробовать сравнить быстродействие и эффективность метода с методом LZ78.

Вариант 7*

1. Усовершенствовать сжатие методом LZ78. Например, реализовать алгоритм адаптивного сброса словаря (LZW) и т.п. Возможны ваши собственные варианты усовершенствований, но следует подтвердить их работоспособность простым и очевидным примером или работающей программой.
2. Проверить работоспособность усовершенствований на примере файлового сжатия (см. «LZFTest.pas»).
3. Сравнить быстродействие и эффективность усовершенствованого сжатия с методом LZ78 (см. «LZFTest.pas»).

Вариант 8*

1. Создать комбинированный архиватор LZ78+дожатие методом Хаффмана (или иным).
2. Проверить работоспособность усовершенствований на примере файлового сжатия (см. «LZFTest.pas»).

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

3. Сравнить быстродействие и эффективность усовершенствованого сжатия с методом LZ78 (см. «LZFTest.pas»).

Вариант 9*

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 (см. «LZFTest.pas»).

4.5. 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.

4.6. 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 алгоритма Лемпеля-Зива.
4) Получите практические навыки разработки и кодирования алгоритма LZ78.

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. Поиск в больших массивах информации: Методические указания к лабораторной работе / О. Е. Александров. Екатеринбург: УГТУ, 200?. ? с.
3. СТП УГТУ-УПИ 1-96. Общие требования и правила оформления дипломных и курсовых проектов (работ). 1996. 34 с. Группа Т51.

ОБСУЖДЕНИЕ ВЫПОЛНЕНИЯ ЗАДАНИЯ №1

Содержание

П.1. Общие замечания

Для выполнения задания 1 необходимо понимать, что делаютследующие процедуры:

1) «ByteToCodes» — процедура кодировки LZ78, см. пункт 3.5;
2) «GetLastCodes» — процедура завершения кодировки LZ78, см. пункт 3.6;
3) «CodeToBytes» — процедура декодировки LZ78, см. пункт 3.7;

Определения (заголовки) этих процедур находятся в файле «LZDic.pas», а определения типов данных — в файлах «LZTypes.pas» и «LZDic0.pas».

Порядок выполнения следующий.

1) Прежде всего, следует подробно ознакомиться с главами 2 и 3 настоящего руководства и понять суть алгоритма LZ78 и используемой здесь модификации LZ78.
2) Потом, следует научиться запускать тестовую программу для проверки работоспособности метода LZ78 (см. пункт 4.2).
3) Затем, следует ознакомиться с определениями и понять структуру данных используемых в файлах «LZTypes.pas» и «LZDic0.pas».
4) После чего, можно приступать к изучению собственно функций упаковки/распаковки.
5) Ну и наконец, можно поразмыслить об усовершенствованиях метода.

Ниже рассмотрена логика работы функций упаковки/распаковки.

П.2. Процедура ByteToCodes

Рассмотрим процесс кодирования, см. пункт 3.5. В файле «LZDic.pas» процедура *ByteToCodes* определена так

```
function tLZDictionary.ByteToCodes(
                  AByte:tByte;
            var   ACodes:tLZCodesArray
):tCodesCount;
begin
```

```
  { Это обращение к работающей версии процедуры кодировки.
  ВАМ необходимо написать собственную реализацию процедуры кодировки.
  Для этого закомментируйте следующую строку }

  ByteToCodes:=Inherited ByteToCodes(AByte, ACodes);

  { и напишите ниже свой вариант процедуры кодировки.
  ------------------------------------------------------}
end;
```

Здесь строка

```
ByteToCodes:=Inherited ByteToCodes(AByte, ACodes);
```

обращается к отлаженному варианту процедуры кодирования из модуля «LZDic1», Вам этот код недоступен. Прежде всего, закомментируйте эту строку или удалите совсем:

```
function tLZDictionary.ByteToCodes(
                  AByte:tByte;
            var   ACodes:tLZCodesArray
):tCodesCount;
begin
end;
```

Теперь следует написать свой вариант кода. Процедура *ByteToCodes* определена как метод объекта, т.е. в ней доступны данные и методы объекта *tLZDictionary*, см. модули «LZDic.pas» и «LZDic0.pas»:

```
tLZDictionary =object
    public
      { доступные извне методы (процедуры) и данные}
      …
     function ByteToCodes(AByte:tByte; var ACodes:tLZCodesArray):tCodesCount;
      …
    private
      { доступные только здесь методы (процедуры) и данные}
      …

      …
end;
```

В определении объекта присутствуют методы (процедуры и функции) и данные объекта (выше приведена только часть определения, полное определение см. «LZDic0.pas» и «LZDic.pas»). Все методы имеют доступ к данным объекта. В нашем случае данными служат (см. «LZDic0.pas»)

```
prDicFlags:tFlags;           { внутренние флаги }
prDic:tDictionary;           { данные словаря }
prLastCode:tIndex;           { последний код = код текущего фрагмента }
prLastDecodedFragmentFirstByte:tByte;  { первый байт последнего декодированного
                                         фрагмента (только для декодировки) }
prCodeLength:tCodeLength;    { текущая длина кода }
prMaxCode:tNodeIndex;        { максимальное значение кода для текущей длины кода}
prFragmentLength:tIndex;     { длина текущего фрагмента }
prFragmentLengthLimit:tIndex; { максимально допустимая длина фрагмента }
prCurrentMaxFragmentLength, { достигнутый максимум длины фрагмента для текущего
                              словаря }
prCurrentAbsMaxFragmentLength:tIndex; { максимум длины фрагмента за сессию
                                        кодирования }
prCompressedSize:tLZCompressedSize; { упакованный размер данных (только при
                                      кодировании) }
prInitialSizeInBytes:tInt;   { исходный размер данных  (только при кодировании)}
```

При необходимости Вы можете создать собственные данные, дополнительно к существующим. ВНИМАНИЕ! Собственные данные следует размещать в модуле «LZDic.pas», **модуль «LZDic0.pas» исправлять и модифицировать нельзя**!

Для написания процедуры следует уяснить, что она получает в качестве входных данных, и что она должна выдать в качестве результата. Данные подразделяются на локальные (не сохраняются после выхода из процедуры) и глобальные (сохраняются после выхода из процедуры и вновь доступны при следующем вызове).

При ВХОДЕ в процедуру *ByteToCodes*, параметры процедуры:

1) *aByte* — очередной байт из потока данных для кодирования (сжатия);
2) *aCodes* — указывает на массив для возврата кодов.

Глобальные данные:

1) *prLastCode* — содержит код текущего фрагмента;
2) *prCodeLength* — содержит текущую длину кода;
3) *prMaxCode* — содержит максимальное значение кода, которое можно записать с текущей длиной кода;
4) *prDicFlags* — содержат флаги состояния процесса кодирования (*fNewByte*, *fIncCodeLength*, *fEncodeRegime*, *fDecodeRegime*). При кодировании нам нужен только один флаг *fNewByte*, если он установлен в *prDicFlags*, то предыдущим фрагментом был новый байт (байт, которого нет в словаре);
5) *prInitialSizeInBytes* — исходный размер данных (счетчик байтов, поступивших на вход в процедуру;
6) *prCompressedSize* — сведения о размере упакованных данных, хранит длину упакованных данных в битах;
7) *prFragmentLength* — длина текущего фрагмента;
8) *prFragmentLengthLimit* — максимальная допустимая длина фрагмента;
9) *prCurrentMaxFragmentLength* — достигнутый максимум длины фрагмента для текущего словаря.

При ВЫХОДЕ из процедуры *ByteToCodes* возвращается 0 или положительное целое число.

Изменение параметров процедуры *ByteToCodes*:

- Если возвращен 0, то *aCodes* не изменено;
- Если возвращено значение >0, то в *aCodes* помещено соответствующее число кодов.

Глобальные данные

1) *prDicFlags* — изменяется в соответствии с ходом процесса кодирования, используется только флаг *fNewByte*;
2) *prLastCode* — изменяется на новое значение, соответствующее текущему фрагменту;

3) *prCodeLength* — может быть увеличено, если возвращаемый код слишком велик;
4) *prMaxCode* — может быть увеличено, если возвращаемый код слишком велик.
5) *prFragmentLength* — устанавливается длина текущего фрагмента;
6) *prCurrentMaxFragmentLength* — устанавливается достигнутый максимум длины фрагмента для текущего словаря (не обязательно для работы программы);
7) *prInitialSizeInBytes* — увеличивается на 1;
8) *prCompressedSize* — увеличивается в соответствии с ростом размера упакованных данных (числом и длиной возвращенных кодов).

Рассмотрим теперь кодирование *ByteToCodes*.

```
function tLZDictionary.ByteToCodes(
                         AByte:tByte;
                  var    ACodes:tLZCodesArray
):tCodesCount;

var { определим вспомогательные локальные данные }
      i:tCodesCount; { индекс кода в ACodes - следующая свободная ячейка }
      c:tIndex;      { вспомогательная переменная для хранения текущего
                       фрагмента}
begin
{ увеличим длину исходных данных на 1 }
  Inc(prInitialSizeInBytes);
{ поиск продолжения фрагмента, который запомнен в prLastCode }
  c:=Find(prLastCode, AByte);
{ проверка: найдено ли продолжение фрагмента? }
  if c<>cNilIndex then begin
  { продолжение фрагмента найдено - продолжаем поиск }
    ByteToCodes:=0;
    { запоминаем продолжение фрагмента в prLastCode }
    prLastCode:=c;
    { увеличиваем длину текущего фрагмента в prFragmentLength }
    Inc(prFragmentLength);
    { ВСЕ. выходим из ByteToCodes }
  end else begin
 { продолжение фрагмента не найдено - фрагмент закончился  }
     { Будет выведен код(ы), поэтому инициализируем индекс для массива
       кодов. Номер i первого свободного кода в массиве aCodes равен 0 }
    i:=0;
    { проверка на добавление нового значения байта к корню }
    if prLastCode=cRootIndex then begin
       { добавление нового значения байта к корневому узлу -
       этот участок программы вызывается только 1 раз после
       начала кодирования (SetEncodeRegime) }
       { добавляем AByte к корню словаря и запоминаем фрагмент в prLastCode}
       prLastCode:=AddRoot(AByte);
       { записываем в ACodes код Reset }
```

```
  ACodes.Code[i]:=cCode_ResetDictionary;
  { записываем в ACodes длину кода }
  ACodes.Length[i]:=prCodeLength;
  { увеличиваем индекс i = номер свободной ячейки в aCodes }
  Inc(i);
  { записываем в ACodes код AByte, длину кода и увеличиваем индекс i }
  ACodes.Code[i]:=AByte; ACodes.Length[i]:=8; Inc(i);
  { записываем в prCompressedSize размеры данных (кодов) }
  xInc(prCompressedSize,(prCodeLength+8)); { см. LZMiscFs.pas }
  { устанавливаем флаг fNewByte - закодирован новый байт }
  Include(prDicFlags,fNewByte);
  { увеличиваем длину текущего фрагмента в prFragmentLength }
  Inc(prFragmentLength);
end else begin
{ добавление нового байта к некорневому узлу }
  { проверяем: нужна ли запись кода? }
  if (fNewByte in prDicFlags) then begin
  { выводить код не надо - предыдущий код NewByte }
   { убираем флаг fNewByte }
    Exclude(prDicFlags,fNewByte);
  end else begin
   { надо выводить код }
   { проверка длины кода }
    if c>prMaxCode then begin
    { код превышает текущий максимальный размер кода -
      надо увеличить длину кода}
     { записываем в ACodes код IncCodeLength, длину кода и
       увеличиваем индекс i }
      ACodes.Code[i]:=cCode_IncCodeLength; ACodes.Length[i]:=...;
      ...;
     { записываем в prCompressedSize размеры данных (кодов) }
      xInc(prCompressedSize,(prCodeLength+5));
     { вычисляем и записываем в ACodes приращение длины кода 5, и
       увеличиваем индекс i }
      ACodes.Code[i]:=IncrementCodeLength(c); ...; ...;
    end;
    { записываем в ACodes собственно код фрагмента, длину кода
      и увеличиваем индекс i }
    ACodes.Code[i]:=c; ...; ...;
    { записываем в prCompressedSize размеры данных (кодов) }
    xInc(...,(prCodeLength));
  end;

  { Проверяем: длина фрагмента превосходит максимально допустимую? }
  if (prFragmentLength<prFragmentLengthLimit) then begin
   { Нет, пытаемся добавить в словарь новое продолжение фрагмента c}
    if Add(c,AByte)=cNilIndex then begin
   { свободное место в словаре закончилось - сбросить словарь }
     { записываем в prCompressedSize размеры данных (кодов) }
```

```
      xInc(...,(prCodeLength+8));
    { записываем в ACodes код Reset, длину кода и увеличиваем индекс i}
      ...; ...; ...;
    { сбрасываем словарь }
      ResetDictionary;
    { добавляем AByte к корню словаря
      и запоминаем фрагмент в prLastCode }
      prLastCode:=AddRoot(AByte);
    { записываем в ACodes код AByte, длину 8 и увеличиваем индекс i }
      ...; ...; ...;
    { устанавливаем флаг fNewByte - закодирован новый байт }
      Include(prDicFlags,...);
    { увеличиваем длину текущего фрагмента в prFragmentLength }
      ...;
    { возвращаем количество кодов в aCodes }
      ByteToCodes:=i;
    { выходим из процедуры ByteToCodes }
      Exit; { это скачок на последний End процедуры }
  end else begin
    if prCurrentMaxFragmentLength<prFragmentLength then begin
    { увеличение максимальной достигнутой длины фрагмента }
      prCurrentMaxFragmentLength:=prFragmentLength;
    end;
  end;
end;
{ ищем в словаре начало фрагмента }
{ установка словаря на начало для поиска фрагмента }
ResetFragment;
{ поиск в словаре фрагмента, начинающегося с AByte }
c:=Find(..., ...);
if c<>cNilIndex then begin
 { фрагмент найден }
  { запоминаем фрагмент в prLastCode }
   prLastCode:=c;
 { увеличиваем длину текущего фрагмента в prFragmentLength }
   Inc(prFragmentLength);
end else begin
{ фрагмент не найден - добавляем новый байт в словарь }
  { пытаемся добавить в словарь новый байт AByte к корневому узлу }
   c:=AddRoot(...);
  { записываем в prCompressedSize размеры данных (кодов) }
   ...(...,(prCodeLength+8));
  { записываем в ACodes длину кода }
   ...;
   if c=cNilIndex then begin
    { свободное место в словаре закончилось }
     { записываем в ACodes код Reset }
      ...;
    { сбрасываем словарь }
      ...;
```

```
          { добавляем AByte к корню словаря }
           c:=...;
         end else begin
        { добавили успешно }
          { записываем в ACodes код NewByte }
           ...;
         end;
        { увеличиваем индекс i }
         ...;
        { запоминаем фрагмент в prLastCode }
         ...;
       { записываем в ACodes код AByte, длину 8 и увеличиваем индекс i }
         ...; ...; ...;
        { устанавливаем флаг fNewByte - закодирован новый байт }
         ...;
        { увеличиваем длину текущего фрагмента в prFragmentLength }
         ...;
      end;
    end;
    { возвращаем количество кодов в массиве aCodes }
    ByteToCodes:=i;
  end;
end;
```

П.3. Процедура GetLastCodes

Рассмотрим подробно процесс завершения кодирования, см. пункт 3.6. В файле «LZDic.pas» процедура *GetLastCodes* определена так

```
function  tLZDictionary.GetLastCodes(var aCodes:tLZCodesArray):tCodesCount;
begin
  { Это обращение к работающей версии процедуры кодировки.
  ВАМ необходимо написать собственную реализацию процедуры кодировки.
  Для этого закомментируйте следующую строку }

  GetLastCodes:=Inherited GetLastCodes(aCodes);

  { и напишите ниже свой вариант процедуры кодировки.
  ----------------------------------------------------}
end;
```

Здесь строка

```
GetLastCodes:=Inherited GetLastCodes(aCodes);
```

обращается к отлаженному варианту завершения кодирования из модуля «LZDic1.pas». Прежде всего, закомментируйте эту строку или удалите совсем:

```
function  tLZDictionary.GetLastCodes(var aCodes:tLZCodesArray):tCodesCount;
begin
end;
```

Теперь следует написать свой вариант кода. Процедура *GetLastCodes* определена как метод объекта, т.е. в ней доступны данные и методы объекта *tLZDictionary* из модуля «LZDic.pas»:

```
tLZDictionary =object
```

```
    public
      { доступные извне методы (процедуры) и данные}
      …
      function GetLastCodes(var aCodes:tLZCodesArray):tCodesCount;
      …
    private
      { доступные только здесь методы (процедуры) и данные}
      …

      …
end;
```

В определении объекта присутствуют методы (процедуры и функции) и данные объекта (выше приведена только часть определения, полное определение см. «LZDic0.pas» и «LZDic.pas»). Все методы имеют доступ к данным объекта (см. предыдущий раздел). При необходимости можно создать собственные данные, дополнительно к существующим. ВНИМАНИЕ! Собственные данные следует размещать в модуле «LZDic.pas», **модуль «LZDic0.pas» исправлять и модифицировать нельзя**!

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

ВХОД в процедуру:

Локальные данные. Параметры процедуры:

1) *aCodes* — указывает на массив для возврата кодов.

Глобальные данные, используются:

1) *prLastCode* — содержит код текущего фрагмента.
2) *prCodeLength* — содержит текущую длину кода.
3) *prMaxCode* — содержит максимальное значение кода, которое можно записать с текущей длиной кода.
4) *prDicFlags* — содержит флаги состояния процесса кодирования (*fNewByte*, *fIncCodeLength*, *fEncodeRegime*, *fDecodeRegime*). При кодировании нам нужен только один флаг *fNewByte*, если он установлен в *prDicFlags*, то предыдущим фрагментом был новый байт.
5) *prCompressedSize* — содержит сведения о размере упакованных данных.

При ВЫХОДЕ из процедуры возвращается целое число 0 или ненулевое положительное. Параметры процедуры:

- Если возвращен 0, то *aCodes* не изменяется;
- Если возвращено значение >0, то в *aCodes* помещено соответствующее число кодов.

Глобальные данные

1) *prDicFlags* — нужно сбросить флаг *fNewByte,* если он был установлен;
2) *prLastCode* — изменяется на значение *cRootNode*;
3) *prCodeLength* — может быть увеличено, если возвращаемый код слишком велик;

4) *prMaxCode* — может быть увеличено, если возвращаемый код слишком велик.

Например, *tLZDictionary.GetLastCodes* может выглядеть так:

```
function  tLZDictionary.GetLastCodes(var aCodes:tLZCodesArray):tCodesCount;
  { локальные переменные }
  var
     i:tCodesCount; { индекс свободного места в aCodes }
begin
  { проверка на добавление нового значения байта к корню }
  if prLastCode=cRootIndex {см. LZTypes.pas} then begin
   { ничего не делать }
    GetLastCodes:=0;
  end else begin
    { запись кода }
    if (fNewByte in prDicFlags) then begin
      { записывать не надо - предыдущий код "новый байт" }
      Exclude(prDicFlags,fNewByte);
      GetLastCodes:=0;
    end else begin
      i:=0;
      { проверка длины кода и его изменение }
      if prLastCode>prMaxCode then begin
      { код превышает текущий максимальный размер кода -
      увеличить длину кода}
        aCodes.Code[i]:=cCode_IncCodeLength {см. LZTypes.pas};
        aCodes.Length[i]:=prCodeLength; Inc(i);
        xInc(prCompressedSize,(prCodeLength+5)); {см. LZMiscFs.pas}
        aCodes.Code[i]:=IncrementCodeLength(prLastCode); {см. LZDic0.pas}
        aCodes.Length[i]:=5; Inc(i);
      end;
      aCodes.Code[i]:=prLastCode; aCodes.Length[i]:=prCodeLength; Inc(i);
      xInc(prCompressedSize,prCodeLength); {см. LZMiscFs.pas}
      GetLastCodes:=i;
    end;
    prLastCode=cRootIndex;
  end;
end;
```

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

Заключение

Выше описан самый простой вариант реализации для одной процедуры алгоритма LZ78. При выполнении лабораторной работы Вы можете улучшить его.

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