6
Dictionary Methods

Statistical compression methods use a statistical model of the data, which is why the quality of compression they achieve depends on how good that model is. Dictionary-based compression methods do not use a statistical model, nor do they use variable-length codes. Instead they select strings of symbols and encode each string as a *token* using a dictionary. The dictionary holds strings of symbols, and it may be static or dynamic (adaptive). The former is permanent, sometimes permitting the addition of strings but no deletions, whereas the latter holds strings previously found in the input stream, allowing for additions and deletions of strings as new input is being read.

Given a string of n symbols, a dictionary-based compressor can, in principle, compress it down to nH bits where H is the entropy of the string. Thus, dictionary-based compressors are entropy encoders, but only if the input file is very large. For most files in practical applications, dictionary-based compressors produce results that are good enough to make this type of encoder very popular. Such encoders are also general purpose, performing on images and audio data as well as they perform on text.

The simplest example of a static dictionary is a dictionary of the English language used to compress English text. Imagine a dictionary containing perhaps half a million words (without their definitions). A word (a string of symbols terminated by a space or a punctuation mark) is read from the input stream and the dictionary is searched. If a match is found, an index to the dictionary is written into the output stream. Otherwise, the uncompressed word itself is written. (This is an example of *logical compression*.)

As a result, the output stream contains indexes and raw words, and it is important to distinguish between them. One way to achieve this is to reserve an extra bit in every item written. In principle, a 19-bit index is sufficient to specify an item in a $2^{19} = 524{,}288$-word dictionary. Thus, when a match is found, we can write a 20-bit token that consists of a flag bit (perhaps a zero) followed by a 19-bit index. When no match is found, a flag of 1 is written, followed by the size of the unmatched word, followed by the word itself.

D. Salomon, G. Motta, *Handbook of Data Compression*, 5th ed.
DOI 10.1007/978-1-84882-903-9_6,

Example: Assuming that the word `bet` is found in dictionary entry 1025, it is encoded as the 20-bit number 0|00000001000000000001. Assuming that the word `xet` is not found, it is encoded as 1|0000011|01111000|01100101|01110100. This is a 4-byte number where the 7-bit field 0000011 indicates that three more bytes follow.

Assuming that the size is written as a 7-bit number, and that an average word size is five characters, an uncompressed word occupies, on average, six bytes (= 48 bits) in the output stream. Compressing 48 bits into 20 is excellent, provided that it happens often enough. Thus, we have to answer the question how many matches are needed in order to have overall compression? We denote the probability of a match (the case where the word is found in the dictionary) by P. After reading and compressing N words, the size of the output stream will be $N[20P+48(1-P)] = N[48-28P]$ bits. The size of the input stream is (assuming five characters per word) $40N$ bits. Compression is achieved when $N[48-28P] < 40N$, which implies $P > 0.29$. We need a matching rate of 29% or better to achieve compression.

◇ **Exercise 6.1:** What compression factor do we get with $P = 0.9$?

As long as the input stream consists of English text, most words will be found in a 500,000-word dictionary. Other types of data, however, may not do that well. A file containing the source code of a computer program may contain "words" such as `cout`, `xor`, and `malloc` that may not be found in an English dictionary. A binary file normally contains gibberish when viewed in ASCII, so very few matches may be found, resulting in considerable expansion instead of compression.

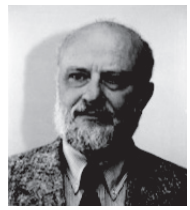
This shows that a static dictionary is not a good choice for a general-purpose compressor. It may, however, be a good choice for a special-purpose one. Consider a chain of hardware stores, for example. Their files may contain words such as `nut`, `bolt`, and `paint` many times, but words such as `peanut`, `lightning`, and `painting` will be rare. Special-purpose compression software for such a company may benefit from a small, specialized dictionary containing, perhaps, just a few hundred words. The computers in each branch would have a copy of the dictionary, making it easy to compress files and send them between stores and offices in the chain.

In general, an adaptive dictionary-based method is preferable. Such a method can start with an empty dictionary or with a small, default dictionary, add words to it as they are found in the input stream, and delete old words because a big dictionary slows down the search. Such a method consists of a loop where each iteration starts by reading the input stream and breaking it up (parsing it) into words or phrases. It then should search the dictionary for each word and, if a match is found, write a token on the output stream. Otherwise, the uncompressed word should be written and also added to the dictionary. The last step in each iteration checks whether an old word should be deleted from the dictionary. This may sound complicated, but it has two advantages:

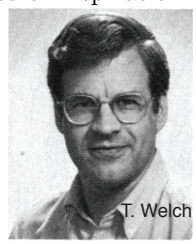
1. It involves string search and match operations, rather than numerical computations. Many programmers prefer that.
2. The decoder is simple (this is an asymmetric compression method). In statistical compression methods, the decoder is normally the exact opposite of the encoder (symmetric compression). In an adaptive dictionary-based method, however, the decoder has to read its input stream, determine whether the current item is a token or uncompressed data, use tokens to obtain data from the dictionary, and output the final, uncompressed

data. It does not have to parse the input stream in a complex way, and it does not have to search the dictionary to find matches. Many programmers like that, too.


J. Ziv


A. Lempel

The LZW Trio. Having one's name attached to a scientific discovery, technique, or phenomenon is considered a special honor in science. Having one's name associated with an entire field of science is even more so. This is what happened to Jacob Ziv and Abraham Lempel. In the late 1970s these two researchers developed the first methods, LZ77 and LZ78, for dictionary-based compression. Their ideas have been a source of inspiration to many researchers who generalized, improved, and combined them with RLE and statistical methods to form many popular lossless compression methods for text, images, and audio. Close to 30 such methods are described in this chapter, some in great detail. Of special interest is the popular LZW algorithm, partly devised by Terry Welch (Section 6.13), which has extended LZ78 and made it practical and popular.


T. Welch

I love the dictionary, Kenny, it's the only book with the words in the right place.

—Paul Reynolds as Colin Mathews in *Press Gang* (1989)

6.1 String Compression

In general, compression methods based on strings of symbols can be more efficient than methods that compress individual symbols. To understand this, the reader should first review Exercise A.4. This exercise shows that in principle, better compression is possible if the symbols of the alphabet have very different probabilities of occurrence. We use a simple example to show that the probabilities of strings of symbols vary more than the probabilities of the individual symbols constituting the strings.

We start with a 2-symbol alphabet a_1 and a_2, with probabilities $P_1 = 0.8$ and $P_2 = 0.2$, respectively. The average probability is 0.5, and we can get an idea of the variance (how much the individual probabilities deviate from the average) by calculating the sum of absolute differences $|0.8 - 0.5| + |0.2 - 0.5| = 0.6$. Any variable-length code would assign 1-bit codes to the two symbols, so the average length of the code is one bit per symbol.

We now generate all the strings of two symbols. There are four of them, shown in Table 6.1a, together with their probabilities and a set of Huffman codes. The average probability is 0.25, so a sum of absolute differences similar to the one above yields

$$|0.64 - 0.25| + |0.16 - 0.25| + |0.16 - 0.25| + |0.04 - 0.25| = 0.78.$$

String	Probability	Code
a_1a_1	$0.8 \times 0.8 = 0.64$	0
a_1a_2	$0.8 \times 0.2 = 0.16$	11
a_2a_1	$0.2 \times 0.8 = 0.16$	100
a_2a_2	$0.2 \times 0.2 = 0.04$	101

(a)

String	Probability	Code
$a_1a_1a_1$	$0.8 \times 0.8 \times 0.8 = 0.512$	0
$a_1a_1a_2$	$0.8 \times 0.8 \times 0.2 = 0.128$	100
$a_1a_2a_1$	$0.8 \times 0.2 \times 0.8 = 0.128$	101
$a_1a_2a_2$	$0.8 \times 0.2 \times 0.2 = 0.032$	11100
$a_2a_1a_1$	$0.2 \times 0.8 \times 0.8 = 0.128$	110
$a_2a_1a_2$	$0.2 \times 0.8 \times 0.2 = 0.032$	11101
$a_2a_2a_1$	$0.2 \times 0.2 \times 0.8 = 0.032$	11110
$a_2a_2a_2$	$0.2 \times 0.2 \times 0.2 = 0.008$	11111

(b)

Str. size	Variance of prob.	Avg. size of code
1	0.6	1
2	0.78	0.78
3	0.792	0.728

(c)

Table 6.1: Probabilities and Huffman Codes for a Two-Symbol Alphabet.

The average size of the Huffman code is $1 \times 0.64 + 2 \times 0.16 + 3 \times 0.16 + 3 \times 0.04 = 1.56$ bits per string, which is 0.78 bits per symbol.

In the next step we similarly create all eight strings of three symbols. They are shown in Table 6.1b, together with their probabilities and a set of Huffman codes. The average probability is 0.125, so a sum of absolute differences similar to the ones above yields

$$|0.512 - 0.125| + 3|0.128 - 0.125| + 3|0.032 - 0.125| + |0.008 - 0.125| = 0.792.$$

The average size of the Huffman code in this case is $1 \times 0.512 + 3 \times 3 \times 0.128 + 3 \times 5 \times 0.032 + 5 \times 0.008 = 2.184$ bits per string, which equals 0.728 bits per symbol.

As we keep generating longer and longer strings, the probabilities of the strings differ more and more from their average, and the average code size gets better (Table 6.1c). This is why a compression method that compresses strings, rather than individual symbols, can, in principle, yield better results. This is also the reason why the various dictionary-based methods are in general better and more popular than the Huffman method and its variants (see also Section 7.19). The above conclusion is a fundamental result of rate-distortion theory, that part of information theory that deals with data compression.

6.2 Simple Dictionary Compression

The topic of this section is a simple, two-pass method, related to me by Ismail Mohamed (see Preface to the 3rd edition of the complete reference). The first pass reads the source file and prepares a list of all the different bytes found. The second pass uses this list to actually compress the data bytes. Here are the steps in detail.

1. The source file is read and a list is prepared of the distinct bytes encountered. For each byte, the number of times it occurs in the source file (its frequency) is also included in the list.

2. The list is sorted in descending order of the frequencies. Thus, it starts with byte values that are common in the file, and it ends with bytes that are rare. Since the list consists of distinct bytes, it can have at most 256 elements.

3. The sorted list becomes the dictionary. It is written on the compressed file, preceded by its length (a 1-byte integer).

4. The source file is read again byte by byte. Each byte is located in the dictionary (by a direct search) and its index is noted. The index is a number in the interval $[0, 255]$, so it requires between 1 and 8 bits (but notice that most indexes will normally be small numbers because common byte values are stored early in the dictionary). The index is written on the compressed file, preceded by a 3-bit code denoting the index's length. Thus, code 000 denotes a 1-bit index, code 001 denotes a 2-bit index, and so on up to code 111, which denotes an 8-bit index.

The compressor maintains a short, 2-byte buffer where it collects the bits to be written on the compressed file. When the first byte of the buffer is filled, it is written on the file and the second byte is moved to the first byte.

Decompression is straightforward. The decompressor starts by reading the length of the dictionary, then the dictionary itself. It then decodes each byte by reading its 3-bit code, followed by its index value. The index is used to locate the next data byte in the dictionary.

Compression is achieved because the dictionary is sorted by the frequency of the bytes. Each byte is replaced by a quantity of between 4 and 11 bits (a 3-bit code followed by 1 to 8 bits). A 4-bit quantity corresponds to a compression ratio of 0.5, while an 11-bit quantity corresponds to a compression ratio of 1.375, an expansion. The worst case is a file where all 256 byte values occur and have a uniform distribution. The compression ratio in such a case is the average

$$(2\times 4+2\times 5+4\times 6+8\times 7+16\times 8+32\times 9+64\times 10+128\times 11)/(256\times 8)$$
$$=2562/2048=1.2509765625,$$

indicating expansion! (Actually, slightly worse than 1.25, because the compressed file also includes the dictionary, whose length in this case is 257 bytes.) Experience indicates typical compression ratios of about 0.5.

The probabilities used here were obtained by counting the numbers of codes of various sizes. Thus, there are two 4-bit codes 000|0 and 000|1, two 5-bit codes 001|10 and 001|11, four 6-bit codes 010|100, 010|101, 010|110 and 010|111, eight 7-bit codes 011|1000, 011|1001, 011|1010, 011|1011, 011|1100, 011|1101, 011|1110, and 011|1111, and so on, up to 128 11-bit codes.

The downside of the method is slow compression; a result of the two passes the compressor performs combined with the slow search (slow, because the dictionary is not sorted by byte values, so binary search cannot be used). Decompression, in contrast, is not slow.

◇ **Exercise 6.2:** Design a reasonable organization for the list maintained by this method.

6.3 LZ77 (Sliding Window)

An important source of redundancy in digital data is repeating phrases. This type of redundancy exists in all types of data—audio, still images, and video—but is easiest to visualize for text. Figure 6.2 lists two paragraphs from this book and makes it obvious that certain phrases, such as words, parts of words, and other bits and pieces of text repeat several times or even many times. Phrases tend to repeat in the same paragraph, but repetitions are often found in paragraphs that are distant from one another. Thus, phrases that occur again and again in the data constitute the basis of all the dictionary-based compression algorithms.

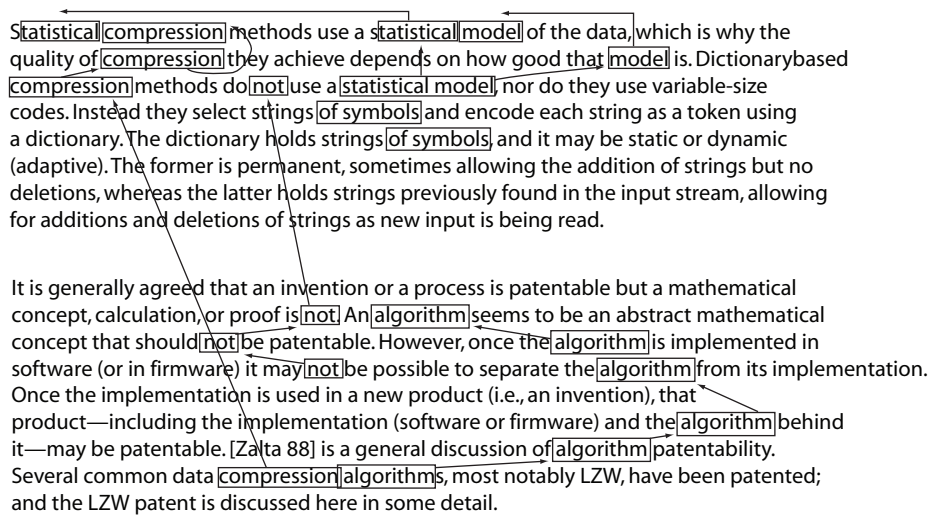


Figure 6.2: Repeating Phrases in Text.

The principle of LZ77 (sometimes also referred to as LZ1) [Ziv and Lempel 77] is to use part of the previously-seen input stream as the dictionary. The encoder maintains a window to the input stream and shifts the input in that window from right to left as strings of symbols are being encoded. Thus, the method is based on a *sliding window*. The window below is divided into two parts. The part on the left is the *search buffer*. This is the current dictionary, and it includes symbols that have recently been input and encoded. The part on the right is the *look-ahead buffer*, containing text yet to be

encoded. In practical implementations the search buffer is some thousands of bytes long, while the look-ahead buffer is only tens of bytes long. The vertical bar between the `t` and the `e` below represents the current dividing line between the two buffers. We assume that the text `sir␣sid␣eastman␣easily␣t` has already been compressed, while the text `eases␣sea␣sick␣seals` still needs to be compressed.

← coded text...`sir␣sid␣eastman␣easily␣t`|`eases␣sea␣sick␣seals`...← text to be read

The encoder scans the search buffer backwards (from right to left) looking for a match for the first symbol `e` in the look-ahead buffer. It finds one at the `e` of the word `easily`. This `e` is at a distance (offset) of 8 from the end of the search buffer. The encoder then matches as many symbols following the two `e`'s as possible. Three symbols `eas` match in this case, so the length of the match is 3. The encoder then continues the backward scan, trying to find longer matches. In our case, there is one more match, at the word `eastman`, with offset 16, and it has the same length. The encoder selects the longest match or, if they are all the same length, the last one found, and prepares the token (16, 3, `e`).

Selecting the last match, rather than the first one, simplifies the encoder, because it only has to keep track of the latest match found. It is interesting to note that selecting the first match, while making the program somewhat more complex, also has an advantage. It selects the smallest offset. It would seem that this is not an advantage, because a token should have room for the largest possible offset. However, it is possible to follow LZ77 with Huffman, or some other statistical coding of the tokens, where small offsets are assigned shorter codes. This method, proposed by Bernd Herd, is called LZH. Having many small offsets implies better compression in LZH.

◇ **Exercise 6.3:** How does the decoder know whether the encoder selects the first match or the last match?

In general, an LZ77 token has three parts: offset, length, and next symbol in the look-ahead buffer (which, in our case, is the *second* `e` of the word `teases`). The offset can also be thought of as the distance between the previous and the current occurrences of the string being compressed. This token is written on the output stream, and the window is shifted to the right (or, alternatively, the input stream is moved to the left) four positions: three positions for the matched string and one position for the next symbol.

...`sir␣`|`sid␣eastman␣easily␣tease`|`s␣sea␣sick␣seals...`|...

If the backward search yields no match, an LZ77 token with zero offset and length and with the unmatched symbol is written. This is also the reason a token has a third component. Tokens with zero offset and length are common at the beginning of any compression job, when the search buffer is empty or almost empty. The first five steps in encoding our example are the following:

	`sir␣sid␣eastman␣`	⇒	(0,0,"`s`")
`s`	`ir␣sid␣eastman␣e`	⇒	(0,0,"`i`")
`si`	`r␣sid␣eastman␣ea`	⇒	(0,0,"`r`")
`sir`	`␣sid␣eastman␣eas`	⇒	(0,0,"`␣`")
`sir␣`	`sid␣eastman␣easi`	⇒	(4,2,"`d`")

◇ **Exercise 6.4:** What are the next two steps?

Clearly, a token of the form $(0, 0, \ldots)$, which encodes a single symbol, does not provide good compression. It is easy to estimate its length. The size of the offset is $\lceil \log_2 S \rceil$, where S is the length of the search buffer. In practice, the search buffer may be a few thousand bytes long, so the offset size is typically 10–12 bits. The size of the "length" field is similarly $\lceil \log_2(L-1) \rceil$, where L is the length of the look-ahead buffer (see below for the -1). In practice, the look-ahead buffer is only a few tens of bytes long, so the size of the "length" field is just a few bits. The size of the "symbol" field is typically 8 bits, but in general, it is $\lceil \log_2 A \rceil$, where A is the alphabet size. The total size of the 1-symbol token $(0, 0, \ldots)$ may typically be $11 + 5 + 8 = 24$ bits, much longer than the raw 8-bit size of the (single) symbol it encodes.

Here is an example showing why the "length" field may be longer than the size of the look-ahead buffer:

...Mr.␣alf␣eastman␣easily␣grows␣alf|alfa␣in␣his␣garden... .

The first symbol `a` in the look-ahead buffer matches the five `a`'s in the search buffer. It seems that the two extreme `a`'s match with a length of 3 and the encoder should select the last (leftmost) of them and create the token (28,3,"`a`"). In fact, it creates the token (3,4,"␣"). The four-symbol string `alfa` in the look-ahead buffer is matched with the last three symbols `alf` in the search buffer **and** the first symbol `a` in the look-ahead buffer. The reason for this is that the decoder can handle such a token naturally, without any modifications. It starts at position 3 of its search buffer and copies the next four symbols, one by one, extending its buffer to the right. The first three symbols are copies of the old buffer contents, and the fourth one is a copy of the first of those three. The next example is even more convincing (and only somewhat contrived):

···alf␣eastman␣easily␣yells␣A|AAAAAAAAAA AAAAAH... .

The encoder creates the token (1,9,A), matching the first nine copies of `A` in the look-ahead buffer and including the tenth `A`. This is why, in principle, the length of a match can be up to the size of the look-ahead buffer minus 1.

The decoder is much simpler than the encoder (LZ77 is therefore an asymmetric compression method). It has to maintain a buffer, equal in size to the encoder's window. The decoder inputs a token, finds the match in its buffer, writes the match and the third token field on the output stream, and shifts the matched string and the third field into the buffer. This implies that LZ77, or any of its variants, is useful in cases where a file is compressed once (or just a few times) and is decompressed often. A rarely-used archive of compressed files is a good example.

At first it seems that this method does not make any assumptions about the input data. Specifically, it does not pay attention to any symbol frequencies. A little thinking, however, shows that because of the nature of the sliding window, the LZ77 method always compares the look-ahead buffer to the recently-input text in the search buffer and never to text that was input long ago (and has therefore been flushed out of the search buffer). Thus, the method implicitly assumes that patterns in the input data occur close together. Data that satisfies this assumption will compress well.

The basic LZ77 method was improved in several ways by researchers and programmers during the 1980s and 1990s. One way to improve it is to use variable-size "offset"

and "length" fields in the tokens. Another way is to increase the sizes of both buffers. Increasing the size of the search buffer makes it possible to find better matches, but the trade-off is an increased search time. A large search buffer therefore requires a more sophisticated data structure that allows for fast search (Section 6.13.2). A third improvement has to do with sliding the window. The simplest approach is to move all the text in the window to the left after each match. A faster method is to replace the linear window with a *circular queue*, where sliding the window is done by resetting two pointers (Section 6.3.1). Yet another improvement is adding an extra bit (a flag) to each token, thereby eliminating the third field (Section 6.4). Of special notice is the hash table employed by the Deflate algorithm (Section 6.25.3) to search for matches.

6.3.1 A Circular Queue

The circular queue is a basic data structure. Physically, it is a linear array, but it is used as a circular array. Figure 6.3 illustrates a simple example. It shows a 16-byte array with characters appended at the "end" and deleted from the "start." Both the start and end positions move, and two pointers, `s` and `e`, point to them all the time. In (a) the queue consists of the eight characters `sid␣east`, with the rest of the buffer empty. In (b) all 16 bytes are occupied, and `e` points to the end of the buffer. In (c), the first letter `s` has been deleted and the `l` of `easily` inserted. Notice how pointer `e` is now located *to the left* of `s`. In (d), the two letters `id` have been deleted just by moving the `s` pointer; the characters themselves are still present in the array but have been effectively deleted. In (e), the two characters `y␣` have been appended and the `e` pointer moved. In (f), the pointers show that the buffer ends at `teas` and starts at `tman`. Inserting new characters into the circular queue and moving the pointers is thus equivalent to shifting the contents of the queue. No actual shifting or moving is necessary, though.

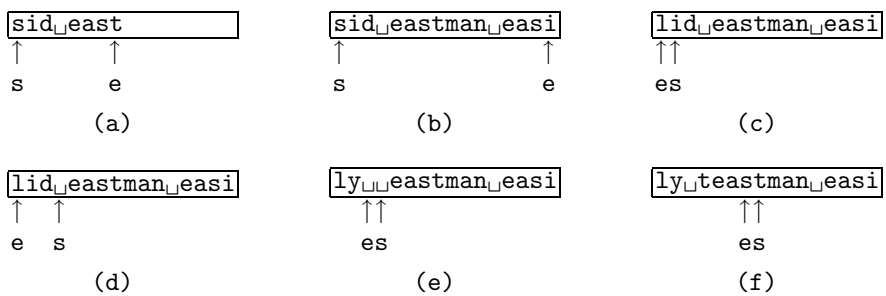


Figure 6.3: A Circular Queue.

More information on circular queues can be found in most texts on data structures.

From the dictionary

Circular. (1) Shaped like or nearly like a circle. (2) Defining one word in terms of another that is itself defined in terms of the first word. (3) Addressed or distributed to a large number of persons.

6.3.2 LZR

The data compression literature is full of references to LZ77 and LZ78. There is, however, also an LZ76 algorithm [Lempel and Ziv 76]. It is little known because it deals with the complexity of text, rather than with compressing the text. The point is that LZ76 measures complexity by looking for previously-found strings of text. It can therefore be extended to become a text compression algorithm. The LZR method described here, proposed by [Rodeh et al. 81], is such an extension.

LZR is a variant of the basic LZ77 method, where the lengths of both the search and look-ahead buffers are unbounded. In principle, such a method will always find the best possible match to the string in the look-ahead buffer, but it is immediately obvious that any practical implementation of LZR will have to deal with the finite memory space and run-time available.

The space problem can be handled by allocating more and more space to the buffers until no more space is left, and then either (1) keeping the buffers at their current sizes or (2) clearing the buffers and starting with empty buffers. The time constraint may be more severe. The basic algorithm of LZ76 uses linear search and requires $O(n^2)$ time to process an input string of n symbols. The developers of LZR propose to reduce these space and time complexities to $O(n)$ by employing the special suffix trees proposed by [McCreight 76], but readers of this chapter will have many opportunities to see that other LZ methods, such as LZ78 and its varieties, offer similar compression performance, while being simpler to implement.

The data structure developed by McCreight is based on a complicated multiway tree. Those familiar with tree data structures know that the basic operations on a tree are (1) adding a new node, (2) modifying an existing node, and (3) deleting a node. Of these, deletion is normally the most problematic since it tends to change the entire structure of a tree.

Thus, instead of deleting nodes when the symbols represented by them overflow off the left end of the search buffer, LZR simply marks a node as deleted. It also constructs several trees and deletes a tree when all its nodes have beed marked as deleted (i.e., when all the symbols represented by the tree have slid off the search buffer). As a result, LZR implementation is complex.

Another downside of LZR is the sizes of the offset and length fields of an output triplet. With buffers of unlimited sizes, these fields may become big. LZR handles this problem by encoding these fields in a variable-length code. Specifically, the Even–Rodeh codes of Section 3.7 are used to encode these fields. The length of the Even–Rodeh code of the integer n is close to $2\log_2 n$, so even for n values in the millions, the codes are about 20 bits long.

6.4 LZSS

LZSS is an efficient variant of LZ77 developed by Storer and Szymanski in 1982 [Storer and Szymanski 82]. It improves LZ77 in three directions: (1) It holds the look-ahead buffer in a circular queue, (2) it creates tokens with two fields instead of three, and (3) it holds the search buffer (the dictionary) in a binary search tree, (this data structure was incorporated in LZSS by [Bell 86]).

The second improvement of LZSS over LZ77 is in the tokens created by the encoder. An LZSS token contains just an offset and a length. If no match was found, the encoder emits the uncompressed code of the next symbol instead of the wasteful three-field token $(0, 0, \ldots)$. To distinguish between tokens and uncompressed codes, each is preceded by a single bit (a flag).

In practice, the search buffer may be a few thousand bytes long, so the offset field would typically be 11–13 bits. The size of the look-ahead buffer should be selected such that the total size of a token would be 16 bits (2 bytes). For example, if the search buffer size is 2 Kbyte $(= 2^{11})$, then the look-ahead buffer should be 32 bytes long $(= 2^5)$. The offset field would be 11 bits long and the length field, 5 bits (the size of the look-ahead buffer). With this choice of buffer sizes the encoder will emit either 2-byte tokens or 1-byte uncompressed ASCII codes. But what about the flag bits? A good practical idea is to collect eight output items (tokens and ASCII codes) in a small buffer, then output one byte consisting of the eight flags, followed by the eight items (which are 1 or 2 bytes long each).

The third improvement has to do with binary search trees. A binary search tree is a binary tree where the left subtree of every node A contains nodes smaller than A, and the right subtree contains nodes greater than A. Since the nodes of our binary search trees contain strings, we first need to know how to compare two strings and decide which one is "bigger." This is easily understood by imagining that the strings appear in a dictionary or a lexicon, where they are sorted alphabetically. Clearly, the string `rote` precedes the string `said` since `r` precedes `s` (even though `o` follows `a`), so we consider `rote` smaller than `said`. This concept is called *lexicographic order* (ordering strings lexicographically).

What about the string ␣`abc`? Most modern computers use ASCII codes to represent characters (although more and more use Unicode, discussed in Section 11.12, and some older IBM, Amdahl, Fujitsu, and Siemens mainframe computers use the old, 8-bit EBCDIC code developed by IBM), and in ASCII the code of a blank space precedes those of the letters, so a string that starts with a space will be smaller than any string that starts with a letter. In general, the *collating sequence* of the computer determines the sequence of characters arranged from small to big. Figure 6.4 shows two examples of binary search trees.

Notice the difference between the (almost) balanced tree in Figure 6.4a and the skewed one in Figure 6.4b. They contain the same 14 nodes, but they look and behave very differently. In the balanced tree any node can be found in at most four steps. In the skewed tree up to 14 steps may be needed. In either case, the maximum number of steps needed to locate a node equals the height of the tree. For a skewed tree (which is really the same as a linked list), the height is the number of elements n; for a balanced tree, the height is $\lceil \log_2 n \rceil$, a much smaller number. More information on the properties

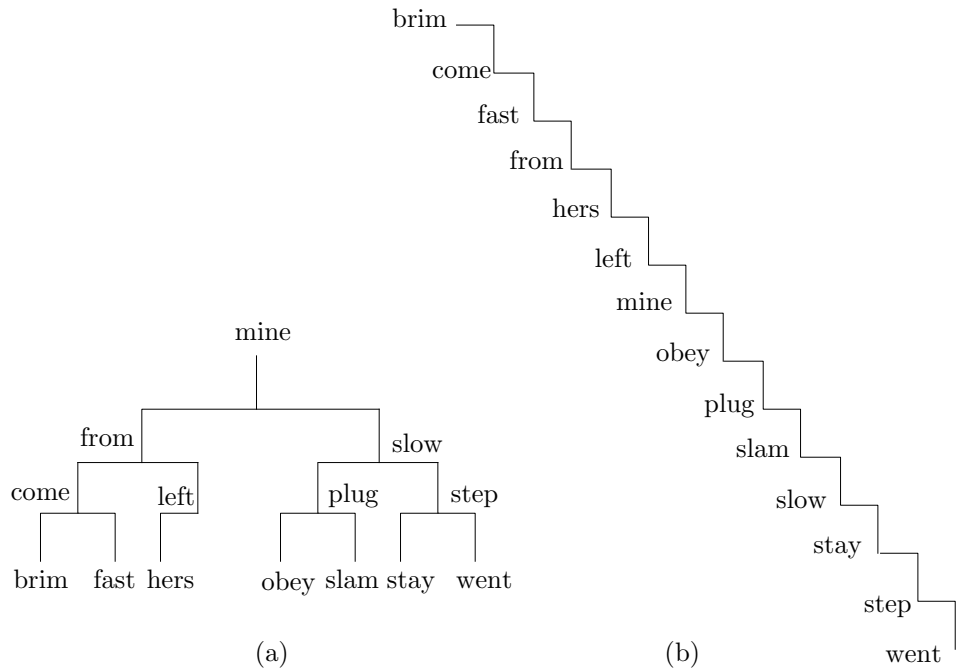


Figure 6.4: Two Binary Search Trees.

of binary search trees may be found in any text on data structures.

Here is an example showing how a binary search tree can be used to speed up the search of the dictionary. We assume an input stream with the short sentence `sid␣eastman␣clumsily␣teases␣sea␣sick␣seals`. To keep the example simple, we assume a window of a 16-byte search buffer followed by a 5-byte look-ahead buffer. After the first $16 + 5$ characters have been input, the sliding window is

`sid␣eastman␣clum|sily␣`teases␣sea␣sick␣seals

with the string `teases␣sea␣sick␣seals` still waiting to be input.

The encoder scans the search buffer, creating the 12 five-character strings of Table 6.5 (12 since $16 - 5 + 1 = 12$), which are inserted into the binary search tree, each with its offset.

The first symbol in the look-ahead buffer is `s`, so the encoder searches the tree for strings that start with an `s`. Two are found, at offsets 16 and 10, and the first of them, `sid␣e` (at offset 16) provides a longer match.

(We now have to sidetrack and discuss the case where a string in the tree completely matches that in the look-ahead buffer. In that case the encoder should go back to the search buffer, to attempt to match longer strings. In principle, the maximum length of a match can be $L - 1$.)

In our example, the match is of length 2, and the 2-field token $(16, 2)$ is emitted. The encoder now has to slide the window two positions to the right, and update the tree. The new window is

sid␣e	16
id␣ea	15
d␣eas	14
␣east	13
eastm	12
astma	11
stman	10
tman␣	09
man␣c	08
an␣cl	07
n␣clu	06
␣clum	05

Table 6.5: Five-Character Strings.

sid␣eastman␣clumsi|ly␣te|ases␣sea␣sick␣seals

The tree should be updated by deleting strings sid␣e and id␣ea, and inserting the new strings clums and lumsi. If a longer, k-letter, string is matched, the window has to be shifted k positions, and the tree should be updated by deleting k strings and adding k new strings, but which ones?

A little thinking shows that the k strings to be deleted are the first ones in the search buffer before the shift, and the k strings to be added are the last ones in it after the shift. A simple procedure for updating the tree is to prepare a string consisting of the first five letters in the search buffer, find it in the tree, and delete it. Then slide the buffer one position to the right (or shift the data to the left), prepare a string consisting of the last five letters in the search buffer, and append it to the tree. This should be repeated k times.

Since each update deletes and adds the same number of strings, the tree size never changes, except at the start and end of the compression. It always contains T nodes, where T is the length of the search buffer minus the length of the look-ahead buffer plus 1 ($T = S - L + 1$). The shape of the tree, however, may change significantly. As nodes are being added and deleted, the tree may change its shape between a completely skewed tree (the worst case for searching) and a balanced one, the ideal shape for searching.

When the encoder starts, it outputs the first L symbols of the input in raw format and shifts them from the look-ahead buffer to the search buffer. It then fills up the remainder of the search buffer with blanks. In our example, the initial tree will consist of the five strings ␣␣␣␣s, ␣␣␣si, ␣␣sid, ␣sid␣, and sid␣e. As compression progresses, the tree grows to $S - L + 1 = 12$ nodes, and stays at that size until it shrinks toward the end.

6.4.1 LZB

LZB is an extension of LZSS. This method, proposed in [Bell 87], is the result of evaluating and comparing several data structures and variable-length codes with an eye toward improving the performance of LZSS.

In LZSS, the length of the offset field is fixed and is large enough to accommodate any pointers to the search buffer. If the size of the search buffer is S, then the offset

field is $s \stackrel{\text{def}}{=} \lceil \log_2 S \rceil$ bits long. However, when the encoder starts, the search buffer is empty, so the offset fields of the first tokens output do not use all the bits allocated to them. In principle, the size of the offset field should start at one bit. This is enough as long as there are only two symbols in the search buffer. The offset should then be increased to two bits until the search buffer contains four symbols, and so on. As more and more symbols are shifted into the search buffer and tokens are output, the size of the offset field should be increased. When the search buffer is between 25% and 50% full, the size of the offset should still be $s-1$ and should be increased to s only when the buffer is 50% full or more.

Even this sophisticated scheme can be improved upon to yield slightly better compression. Section 2.9 introduces phased-in codes and shows how to construct a set of pointers of two sizes to address an array of n elements even when n is not a power of 2. LZB employs phased-in codes to encode the size of the offset field at any point in the encoding. As an example, when there are nine symbols in the search buffer, LZB uses the seven 3-bit codes and two 4-bit codes 000, 001, 010, 011, 100, 101, 110, 1110, and 1111.

The second field of an LZSS token is the match length l. This is normally a fairly small integer, but can sometimes have large values. Recall that LZSS computes a value p and outputs a token only if it requires fewer bits than emitting p raw characters. Thus, the smallest value of l is $p+1$. LZB employs the Elias gamma code of Section 3.4 to encode the length field. A match of length i is encoded as the gamma code of the integer $i-p+1$. The length of the gamma code of the integer n is $1+2\lfloor \log_2 n \rfloor$ and it is ideal for cases where n appears in the input with probability $1/(2n^2)$. As a simple example, assuming that $p=3$, a match of five symbols is encoded as the gamma code of $5-3+1=3$ which is 011.

6.4.2 SLH

SLH is another variant of LZSS, described in [Brent 87]. It employs a circular buffer and it outputs raw symbols and pairs as does LZSS. The main difference is that SLH is a two-pass algorithm where the first pass employs a hash table to locate the best match and to count frequencies, and the second pass encodes the offsets and the raw symbols with Huffman codes prepared from the frequencies counted by the first pass.

We start with a few details about the hash table H. The first pass starts with an empty H. As strings are searched, matches are either found in H or are not found and are then added to H. What is actually stored in a location of H when a string is added, is a pointer to the string in the search buffer. In order to find a match for a string s of symbols, s is passed through a hashing function that returns a pointer to H. Thus, a match to s can be located (or verified as not found) in H in one step. (This discussion of hash tables ignores collisions, and detailed descriptions of this data structure can be found in many texts.)

We denote by p the break-even point, where a token representing a match of p or more symbols is shorter than the raw symbols themselves. At a general point in the encoding process, the search buffer has symbols a_1 through a_j and the look-ahead buffer has symbols a_{j+1} through a_n. We select the p-symbol string $s = a_{j+1}a_{j+2}\ldots a_{j+p-1}$, hash it to a pointer q and check $H[q]$. If a match is not found, then s is added to H (i.e., an offset P is stored in H and is updated as the buffers are shifted and s moves to the

left). If a match is found, the offset P found in H and the match length are prepared as the token pair (P, p) representing the best match so far. This token is saved but is not immediately output. In either case, match found or not found, the match length is incremented by 1 and the process is repeated in an attempt to find a longer match.

In practice, the search buffer is implemented as a circular queue. Each time it is shifted, symbols are moved out of its "left" end and matches containing these symbols have to be deleted from H, which further complicates this algorithm.

The second pass of SLH encodes the offsets and the raw characters with Huffman codes. The frequencies counted in the first pass are first normalized to the interval $[0, 511]$ and a table with 512 Huffman codes is prepared (this table has to be written on the output, but its size is only a few hundred bytes). The implementation discussed in [Brent 87] assumes that the data symbols are bytes and that S, the size of the search buffer, is less than 2^{18}. Thus, an offset (a pointer to the search buffer) is at most 18 bits long. An 18-bit offset is broken up into two 9-bit parts and each part is Huffman encoded separately. Each raw character is also Huffman encoded.

The developer of this method does not indicate how an 18-bit offset should be split in two and precisely what frequencies are counted by the first pass. If that pass counts the frequencies of the raw input characters, then these characters will be efficiently encoded by the Huffman codes, but the same Huffman codes will not correspond to the frequencies of the 9-bit parts of the offset. It is possible to count the frequencies of the 9-bit parts and prepare two separate Huffman code tables for them, but other LZ methods offer the same or better compression performance, while being simpler to implement and evaluate.

The developer of this method does not explain the name SLH. The excellent book *Text Compression* [Bell et al. 90] refers to this method as LZH, but our book already uses LZH as the name of an LZ77 variant proposed by Bernd Herd, which leads to ambiguity. Even data compression experts cannot always avoid confusion.

> An expert is a man who tells you a simple thing in a confused way in such a fashion as to make you think the confusion is your own fault.
>
> —William Castle

6.4.3 LZARI

The following is quoted from [Okumura 98].

During the summer of 1988, I [Haruhiko Okumura] wrote another compression program, LZARI. This program is based on the following observation: Each output of LZSS is either a single character or a ⟨position,length⟩ pair. A single character can be coded as an integer between 0 and 255. As for the ⟨length⟩ field, if the range of ⟨length⟩ is 2 to 257, say, it can be coded as an integer between 256 and 511. Thus, I can say that there are 512 kinds of "characters," and the "characters" 256 through 511 are accompanied by a ⟨position⟩ field. These 512 "characters" can be Huffman-coded, or better still, algebraically coded. The ⟨position⟩ field can be coded in the same manner. In LZARI, I used an adaptive algebraic compression to encode the "characters," and static algebraic compression to encode the ⟨position⟩ field. (There were several versions of LZARI; some of them were slightly different from the above description.) The compression of LZARI was very tight, though rather slow.

6.5 LZPP

The original LZ methods were published in 1977 and 1978. They established the field of dictionary-based compression, which is why the 1980s were the golden age of this approach to compression, Most LZ algorithms were developed during this decade, but LZPP, the topic of this section, is an exception. This method [Pylak 03] is a modern, sophisticated algorithm that extends LZSS in several directions and has been inspired by research done and experience gained by many workers in the 1990s. LZPP identifies several sources of redundancy in the various quantities generated and manipulated by LZSS and exploits these sources to obtain better overall compression.

Virtually all LZ methods output offsets (also referred to as pointers or indexes) to a dictionary, and LZSS is no exception. However, many experiments indicate that the distribution of index values is not uniform and that most indexes are small, as illustrated by Figure 6.6. Thus, the set of indexes has large entropy (for the concatenated Calgary Corpus, [Pylak 03] gives this entropy as 8.954), which implies that the indexes can be compressed efficiently with an entropy encoder. LZPP employs a modified range encoder (Section 5.10.1) for this purpose. The original range coder has been modified by adding a circular buffer, an escape symbol, and an exclusion mechanism. In addition, each of the bytes of an index is compressed separately, which simplifies the range coder and allows for 3-byte indexes (i.e., a 16 MB dictionary).

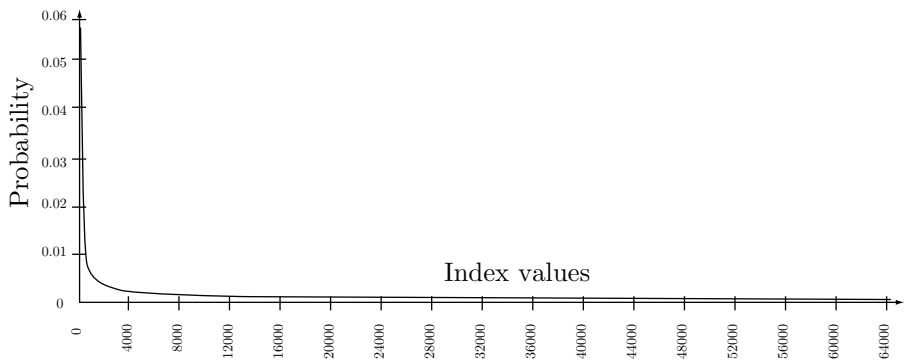


Figure 6.6: Distribution of Typical LZ Index Values (After [Pylak 03]).

LZSS outputs either raw characters or pairs (match length, index). It turns out that the distribution of match lengths is also far from uniform, which makes it possible to compress them too with an entropy encoder. Figure 6.7 shows the distribution of match lengths for the small, 21 KB file `obj1`, part of the Calgary Corpus. It is clear that short matches, of three and four characters, are considerably more common than longer matches, thereby resulting in large entropy of match lengths. The same range coder is used by LZPP to encode the lengths, allowing for match lengths of up to 64 KB.

The next source of redundancy in LZSS is the distribution of flag values. Recall that LZSS associates a flag with each raw character (a 0 flag) and each pair (a flag of 1) it outputs. Flag values of 2 and 3 are also used and are described below. The

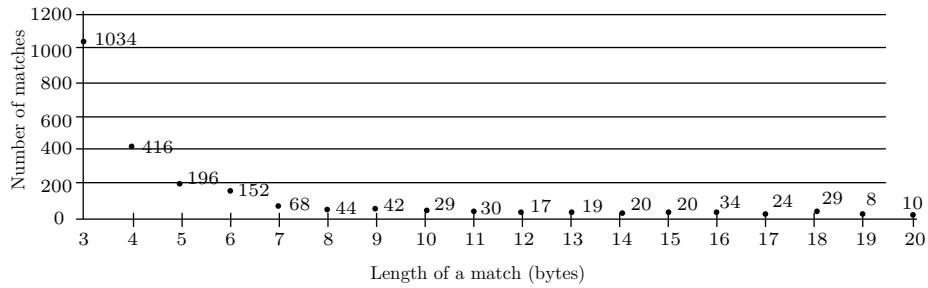


Figure 6.7: Distribution of Match Lengths (After [Pylak 03]).

redundancy in flag values exists because the probabilities of the two values are not 0.5 but vary considerably during encoding.

It is intuitively clear that when the encoder starts, few matches exist, many raw characters are output, and most flags are therefore 0. As encoding progresses, more and more matches are found, so more flags of 1 are output. Thus, the probability of a 0 flag starts high, and slowly drops to a low value, hopefully much less than 0.5.

This probability is illustrated in Figure 6.8, which shows how the probability of a 0 flag drops from almost 1 at the start, to around 0.2 after about 36,000 characters of the input have been read and processed. The input for the figure was file `paper1` of the Calgary Corpus.

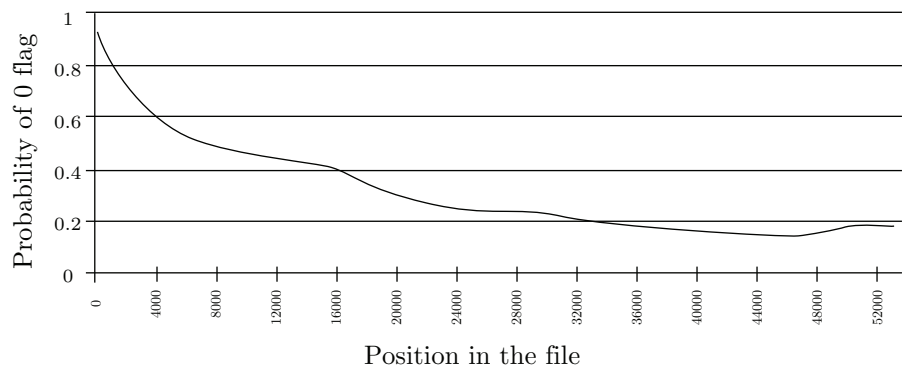


Figure 6.8: Probability of a 0 Flag (After [Pylak 03]).

LZPP uses the same range encoder to compress the flags, which improves its compression pereformance by up to 2%. The description in this section shows that LZPP employs four different flags (as opposed to the two flags employed by LZSS), which makes the compression of flags even more meaningful.

Another source of redundancy in LZSS is the raw characters. LZSS outputs raw characters, while LZPP compresses these characters with the same range encoder.

It should be mentioned that the range encoder employed by LZPP is a variant of adaptive arithmetic coding. Its main input is an item to be encoded, but it must also receive a probability for the item. LZPP uses order-0 and order-1 contexts to estimate the probabilities of the indexes, flags, and characters it sends to the range encoder. The concept of contexts and their various orders is covered in Section 5.14, but here is what order-0 context means. Suppose that the current raw character is `q`. LZPP counts the number of times each character has been encoded and emitted as a raw character token. If `q` has been encoded and output 35 times as a raw character token, then the order-0 probability estimated for this occurrence of `q` is $35/t$, where t is the total number of raw character tokens output so far.

Figure 6.7 shows that match lengths of 3 are much more common than any other lengths, which is why LZPP treats 3-byte matches differently. It constructs and maintains a special data structure (a FIFO queue) D that holds a certain number of 3-byte matches, with the number of occurrences of each. When a new 3-byte match is found in the dictionary, the encoder searches D. If the match is already there, the encoder outputs the single item index3, where index3 is the index to the match in D. A flag of 2 is associated with this item. An index to D is encoded by an entropy coder before it is output, and the important point is that the occurrence count of the match is used by LZPP to compute a probability which is sent to the entropy coder together with index3.

The queue D is also exploited for a further small performance improvement. If LZPP outputs two consecutive raw characters, say, `t` and `h`, it knows that the next character x (the leftmost one in the look-ahead buffer) cannot be the last character of any of the 3-byte matches `th`y in D. Thus, if the trigrams `the`, `tha`, `thi`, and `tho` are in D, the next character cannot be `e`, `a`, or `i`. These characters are *excluded* when the x is encoded. When the encoder computes the probability of the x, it excludes the frequency counts for `e`, `a`, and `i`.

The following illustrates another type of exclusion. If the search and look-ahead buffers hold the following characters:

← coded text...`...␣while␣these␣let␣|themselves␣out␣...`...← text to be read

and the strings `the` in the two buffers are matched, then LZPP knows that the character following `the` in the look-ahead buffer cannot be an `s`. Thus, the symbol `s` is excluded when LZPP computes a probability for encoding the length and index for string `the`.

It has already been mentioned that LZPP handles 3-byte matches in a special way. Experiments with LZPP indicate that other short matches, those of four and five bytes, should also be treated differently. Specifically, it has been noticed that when a 4-byte or a 5-byte match is distant (i.e., has a large index), the resulting pair (length, index) does not yield good compression. As a result, LZPP limits the indexes for short matches. If the smallest index of a 4-byte match exceeds 255, the match is ignored. Similarly, 5-byte matches are ignored if the smallest index exceeds 65,535.

The last source of redundancy in LZSS that is addressed by LZPP is symbol context. The concept of context is central to the PPM algorithm and is treated in detail in Section 5.14. The reader is advised to read the first few paragraphs of that section before continuing.

The idea is that characters read from the input file are not random. Certain characters (such as `e`, `t`, and `a` in English) occur more often than others (such as `j`, `q`, and `z`).

Also certain pairs and triplets (digrams and trigrams) of characters are common while others are rare. LZPP employs order-1 contexts to encode raw characters. It maintains a two-dimensional array C of 256×256 entries where it records the occurrences of digrams.

Thus, if the leftmost character in the look-ahead buffer is `h` and there are no 3-byte or longer matches for it, LZPP is supposed to output the `h` as a raw character (suitably encoded). Before it does that, it locates the rightmost character in the search buffer, say, `t`, and examines the frequency count in row `t` column `h` of array C. If the count c found there (the number of previous occurrences of the digram `th`) is nonzero, the `h` is encoded with probability c/a, where a is the number of characters read so far from the input file. A flag of 3 is associated with the resulting bits. If c is zero (the digram `th` hasn't been seen so far), the `h` is encoded and is output with a 0 flag. This is how LZPP handles the famous zero-probability problem in order-1 contexts.

The escape mechanism. An escape character is added to the alphabet, and is used to encode those raw characters that have zero frequencies. LZPP maintains two counts for each character of the alphabet. The main count is maintained in array C. It is the number of occurrences of the character so far and it may be zero. The secondary count is 1 plus the number of times the character has been coded with the escape mechanism. The secondary count is always nonzero and is used when a character is encoded with the escape. Notice that this count is maintained only for the most-recent 4,096 characters encoded with an escape.

When a character with zero main count is to be emitted, the escape flag is encoded first, followed by the character itself, encoded with a special probability that is computed by considering only the characters with zero occurrences and adding their secondary counts.

A 64 KB hash table is used by the encoder to locate matches in the dictionary. The hash function is the well-known CRC-32 (Section 6.32). The leftmost four characters in the look-ahead buffer are considered a 32-bit integer and the value returned by CRC-32 for this integer is truncated to 16 bits and becomes an index to the 64 MB dictionary. Further search in the dictionary is needed to find matches longer than four bytes.

Tests performed on the Calgary Corpus comparing several compression methods indicate that the compression ratios produced by LZPP are about 30% better than those of LZSS and about 1.5–2% worse than those of WinRAR (Section 6.22), a commercial product which is also based on LZSS. The developer of LZPP also notes that compression algorithms (including the commercial RK software) that are based on PPMZ (Section 5.14.7) outperform LZPP as well as any other dictionary-based algorithm.

6.5.1 Deficiencies

Before we discuss LZ78, let's summarize the deficiencies of LZ77 and its variants. It has already been mentioned that LZ77 uses the built-in implicit assumption that patterns in the input data occur close together. Data streams that don't satisfy this assumption compress poorly. A common example is text where a certain word, say `economy`, occurs often but is uniformly distributed throughout the text. When this word is shifted into the look-ahead buffer, its previous occurrence may have already been shifted out of the search buffer. A better algorithm would save commonly-occurring strings in the dictionary and not simply slide it all the time.

Another disadvantage of LZ77 is the limited size L of the look-ahead buffer. The size of matched strings is limited to $L-1$, but L must be kept small because the process of matching strings involves comparing individual symbols. If L were doubled in size, compression would improve, since longer matches would be possible, but the encoder would be much slower when searching for long matches. The size S of the search buffer is also limited. A large search buffer results in better compression but slows down the encoder, because searching takes longer (even with a binary search tree). Increasing the sizes of the two buffers also means creating longer tokens, thereby reducing compression efficiency. With two-byte tokens, compressing a two-character string into one token results in two bytes plus a flag. Writing the two characters as two raw ASCII codes results in two bytes plus two flags, a very small difference in size. The encoder should, in such a case, use the latter choice and write the two characters in uncompressed form, saving time and wasting just one bit. We say that the encoder has a two-byte *breakeven* point. With longer tokens, the breakeven point increases to three bytes.

6.6 Repetition Times

Frans Willems, one of the developers of context-tree weighting (Section 5.16), is also the developer of this original (although not very efficient) dictionary-based method. The input may consist of any symbols, but the method is described here and also in [Willems 89] for binary input. The input symbols are grouped into words of length L each that are placed in a sliding buffer. The buffer is divided into a look-ahead buffer with words still to be compressed, and a search buffer containing the B most-recently processed words. The encoder tries to match the leftmost word in the look-ahead buffer to the contents of the search buffer. Only one word in the look-ahead buffer is matched in each step. If a match is found, the distance (offset) of the word from the start of the match is denoted by m and is encoded by a 2-part prefix code that's written on the compressed stream. Notice that there is no need to encode the number of symbols matched, because exactly one word is matched. If no match is found, a special code is written, followed by the L symbols of the unmatched word in raw format.

The method is illustrated by a simple example. We assume that the input symbols are bits. We select $L = 3$ for the length of words, and a search buffer of length $B = 2^L - 1 = 7$ containing the seven most-recently processed bits. The look-ahead buffer contains just the binary data, and the commas shown here are used only to indicate word boundaries.

$$\leftarrow \text{coded input}\ldots \boxed{0100100}\boxed{100,000,011,111,011,101,001}\ldots \leftarrow \text{input to be read}$$

It is obvious that the leftmost word "100" in the look-ahead buffer matches the rightmost three bits in the search buffer. The repetition time (the offset) for this word is therefore $m = 3$. (The biggest repetition time is the length B of the search buffer, 7 in our example.) The buffer is now shifted one word (three bits) to the left to become

$$\leftarrow \ldots 010\boxed{0100100}\boxed{000,011,111,011,101,001,\ldots}\ldots \leftarrow \text{input to be read}$$

The repetition time for the current word "000" is $m = 1$ because each bit in this word is matched with the bit immediately to its left. Notice that it is possible to match

the leftmost 0 of the next word "011" with the bit to its left, but this method matches exactly one word in each step. The buffer is again shifted L positions to become

$$\leftarrow \ldots 010010\boxed{0100000|011,111,011,101,001,.....}\ldots \leftarrow \text{input to be read}$$

There is no match for the next word "011" in the search buffer, so m is set to a special value that we denote by 8* (meaning; greater than or equal 8). It is easy to verify that the repetition times of the remaining three words are 6, 4, and 8*.

Each repetition time is encoded by first determining two integers p and q. If $m = 8^*$, then p is set to L; otherwise p is selected as the integer that satisfies $2^p \le m < 2^{p+1}$. Notice that p is located in the interval $[0, L-1]$. The integer q is determined by $q = m - 2^p$, which places it in the interval $[0, 2^p - 1]$. Table 6.9 lists the values of m, p, q, and the prefix codes used for $L = 3$.

m	p	q	Prefix	Suffix	Length
1	0	0	00	none	2
2	1	0	01	0	3
3	1	1	01	1	3
4	2	0	10	00	4
5	2	1	10	01	4
6	2	2	10	10	4
7	2	3	10	11	4
8*	3	—	11	word	5

Table 6.9: Repetition Time Encoding Table for $L = 3$.

Once p and q are known, a prefix code for m is constructed and is written on the compressed stream. It consists of two parts, a prefix and a suffix, that are the binary values of p and q, respectively. Since p is in the interval $[0, L-1]$, the prefix requires $\log(L+1)$ bits. The length of the suffix is p bits. The case $p = L$ is different. Here, the suffix is the raw value (L bits) of the word being compressed.

The compressed stream for the seven words of our example consists of the seven codes

$$01|1, 00, 11|011, 00, 10|10, 10|00, 11|001, \ldots,$$

where the vertical bars separate the prefix and suffix of a code. Notice that the third and seventh words (011 and 001) are included in the codes in raw format.

It is easy to see why this method generates prefix codes. Once a code has been assigned (such as 01|0, the code of $m = 2$), that code cannot be the prefix of any other code because (1) some of the other codes are for different values of p and thus do not start with 01, and (2) codes for the same p do start with 01 but must have different values of q, so they have different suffixes.

The compression performance of this method is inferior to that of LZ77, but it is interesting for the following reasons.

1. It is universal and optimal. It does not use the statistics of the input stream, and its performance asymptotically approaches the entropy of the input as the input stream gets longer.

2. It is shown in [Cachin 98] that this method can be modified to include data hiding (steganography).

6.7 QIC-122

QIC was an international trade association, incorporated in 1987, whose mission was to encourage and promote the widespread use of quarter-inch tape cartridge technology (hence the acronym QIC; see also `http://www.qic.org/html`). The association ceased to exist in 1998, when the technology it promoted became obsolete.

The QIC-122 compression standard is an LZ77 variant that has been developed by QIC for text compression on 1/4-inch data cartridge tape drives. Data is read and shifted into a 2048-byte ($= 2^{11}$) input buffer from right to left, such that the first character is the leftmost one. When the buffer is full, or when all the data has been read into it, the algorithm searches from left to right for repeated strings. The output consists of raw characters and of tokens that represent strings already seen in the buffer. As an example, suppose that the following data have been read and shifted into the buffer:

The first character `A` is obviously not a repetition of any previous string, so it is encoded as a raw (ASCII) character (see below). The next character `B` is also encoded as raw. The third character `A` is identical to the first character but is also encoded as raw since repeated strings should be at least two characters long. Only with the fourth character `A` we do have a repeated string. The string of five `A`'s from position 4 to position 8 is identical to the one from position 3 to position 7. It is therefore encoded as a string of length 5 at offset 1. The offset in this method is the distance between the start of the repeated string and the start of the original one.

The next character `C` at position 9 is encoded as raw. The string `ABA` at positions 10–12 is a repeat of the string at positions 1–3, so it is encoded as a string of length 3 at offset $10 - 1 = 9$. Finally, the string `BABA` at positions 13–16 is encoded with length 4 at offset 2, since it is a repetition of the string at positions 10–13.

◇ **Exercise 6.5:** Suppose that the next four characters of data are `CAAC`

How will they be encoded?

A raw character is encoded as 0 followed by the 8 ASCII bits of the character. A string is encoded as a token that starts with 1 followed by the encoded offset, followed by the encoded length. Small offsets are encoded as 1, followed by 7 offset bits; large offsets are encoded as 0 followed by 11 offset bits (recall that the buffer size is 2^{11}). The length is encoded according to Table 6.10. The 9-bit string 110000000 is written, as an end marker, at the end of the output stream.

Bytes	Length			Bytes	Length				
2	00			17	11	11	1001		
3	01			18	11	11	1010		
4	10			19	11	11	1011		
5	11	00		20	11	11	1100		
6	11	01		21	11	11	1101		
7	11	10		22	11	11	1110		
8	11	11	0000	23	11	11	1111	0000	
9	11	11	0001	24	11	11	1111	0001	
10	11	11	0010	25	11	11	1111	0010	
11	11	11	0011	⋮					
12	11	11	0100	37	11	11	1111	1110	
13	11	11	0101	38	11	11	1111	1111	0000
14	11	11	0110	39	11	11	1111	1111	0001
15	11	11	0111		etc.				
16	11	11	1000						

Table 6.10: Values of the <length> Field.

◇ **Exercise 6.6:** How can the decoder identify the end marker?

When the search algorithm arrives at the right end of the buffer, it shifts the buffer to the left and inputs the next character into the rightmost position of the buffer. The decoder is the reverse of the encoder (symmetric compression).

> When I saw each luminous creature in profile, from the point of view of its body, its egglike shape was like a gigantic asymmetrical yoyo that was standing edgewise, or like an almost round pot that was resting on its side with its lid on. The part that looked like a lid was the front plate; it was perhaps one-fifth the thickness of the total cocoon.
>
> —Carlos Castaneda, *The Fire From Within* (1984)

Figure 6.11 is a precise description of the compression process, expressed in BNF, which is a metalanguage used to describe processes and formal languages unambiguously. BNF uses the following *metasymbols*:

`::=` The symbol on the left is defined by the expression on the right.
`<expr>` An expression still to be defined.
| A logical OR.
`[]` Optional. The expression in the brackets may occur zero or more times.
`()` A comment.
`0,1` The bits 0 and 1.

(Special applications of BNF may require more symbols.)

Table 6.12 shows the results of encoding `ABAAAAAACABABABA` (a 16-symbol string). The reader can easily verify that the output stream consists of the 10 bytes

```
20 90 88 38 1C 21 E2 5C 15 80.
```

```
(QIC-122 BNF Description)
<Compressed-Stream>::=[<Compressed-String>] <End-Marker>
<Compressed-String>::= 0<Raw-Byte> | 1<Compressed-Bytes>
<Raw-Byte>         ::=<b><b><b><b><b><b><b><b> (8-bit byte)
<Compressed-Bytes> ::=<offset><length>
<offset>           ::= 1<b><b><b><b><b><b><b> (a 7-bit offset)
                       |
          0<b><b><b><b><b><b><b><b><b><b><b><b> (an 11-bit offset)
<length>           ::= (as per length table)
<End-Marker>       ::=110000000 (Compressed bytes with offset=0)
<b>                ::=0|1
```

Figure 6.11: BNF Definition of QIC-122.

Raw byte "A"	0 01000001
Raw byte "B"	0 01000010
Raw byte "A"	0 01000001
String "AAAAA" offset=1	1 1 0000001 1100
Raw byte "C"	0 01000011
String "ABA" offset=9	1 1 0001001 01
String "BABA" offset=2	1 1 0000010 10
End-Marker	1 1 0000000

Table 6.12: Encoding the Example.

6.8 LZX

In 1995, Jonathan Forbes and Tomi Poutanen developed an LZ variant (possibly influenced by Deflate) that they dubbed LZX [LZX 09]. The main feature of the first version of LZX was the way it encoded the match offsets, which can be large, by segmenting the size of the search buffer. They implemented LZX on the Amiga personal computer and included a feature where data was grouped into large blocks instead of being compressed as a single unit.

At about the same time, Microsoft devised a new installation media format that it termed, in analogy to a file cabinet, *cabinet files*. A cabinet file has an extension name of `.cab` and may consist of several data files concatenated into one unit and compressed. Initially, Microsoft used two compression methods to compress cabinet files, MSZIP (which is just another name for Deflate) and Quantum, a large-window dictionary-based encoder that employs arithmetic coding. Quantum was developed by David Stafford.

Microsoft later used cabinet files in its Cabinet Software Development Kit (SDK). This is a software package that provides software developers with the tools required to employ cabinet files in any applications that they implement.

In 1997, Jonathan Forbes went to work for Microsoft and modified LZX to compress cabinet files. Microsoft published an official specification for cabinet files, including MSZIP and LZX, but excluding Quantum. The LZX description contained errors to

such an extent that it wasn't possible to create a working implementation from it.

LZX documentation is available in executable file `Cabsdk.exe` located at `http://download.microsoft.com/download/platformsdk/cab/2.0/w98nt42kmexp/en-us/`. After unpacking this executable file, the documentation is found in file `LZXFMT.DOC`.

LZX is also used to compress `chm` files. Microsoft Compiled HTML (`chm`) is a proprietary format initially developed by Microsoft (in 1997, as part of its Windows 98) for online help files. It was supposed to be the successor to the Microsoft WinHelp format. It seems that many Windows users liked this compact file format and started using it for all types of files, as an alternative to PDF. The popular `lit` file format is an extension of `chm`.

A `chm` file is identified by a `.chm` file name extension. It consists of web pages written in a subset of HTML and a hyperlinked table of contents. A `chm` file can have a detailed index and table-of-contents, which is one reason for the popularity of this format. Currently, there are `chm` readers available for both Windows and Macintosh.

LZX is a variant of LZ77 that writes on the compressed stream either unmatched characters or pairs (offset, length). What is actually written on the compressed stream is variable-length codes for the unmatched characters, offsets, and lengths. The size of the search buffer is a power of 2, between 2^{15} and 2^{21}. LZX uses static canonical Huffman trees (Section 5.2.8) to provide variable-length, prefix codes for the three types of data. There are 256 possible character values, but there may be many different offsets and lengths. Thus, the Huffman trees have to be large, but any particular cabinet file being compressed by LZX may need just a small part of each tree. Those parts are written on the compressed stream. Because a single cabinet file may consist of several data files, each is compressed separately and is written on the compressed stream as a block, including those parts of the trees that it requires. The other important features of LZX are listed here.

Repeated offsets. The developers of LZX noticed that certain offsets tend to repeat; i.e., if a certain string is compressed to a pair $(74, \text{length})$, then there is a good chance that offset 74 will be used again soon. Thus, the three special codes 0, 1, and 2 were allocated to encode three of the most-recent offsets. The actual offset associated with each of those codes varies all the time. We denote by $R0$, $R1$, and $R2$ the most-recent, second most-recent, and third most-recent offsets, respectively (these offsets must themselves be nonrepeating; i.e., none should be 0, 1, or 2). We consider $R0$, $R1$, and $R2$ a short list and update it similar to an LRU (least-recently used) queue. The three quantities are initialized to 1 and are updated as follows. Assume that the current offset is X, then

$$\begin{aligned}\text{if } X \neq R0 \text{ and } X \neq R1 \text{ and } X \neq R2, \text{ then } R2 \leftarrow R1,\ R1 \leftarrow R0,\ R0 \leftarrow X,\\ \text{if } X = R0, \text{ then nothing,}\\ \text{if } X = R1, \text{ then swap } R0 \text{ and } R1,\\ \text{if } X = R2, \text{ then swap } R0 \text{ and } R2.\end{aligned}$$

Because codes 0, 1, and 2 are allocated to the three special recent offsets, an offset of 3 is allocated code 5, and, in general, an offset x is assigned code $x+2$. The largest offset is the size of the search buffer minus 3, and its assigned code is the size of the search buffer minus 1.

Encoder preprocessing. LZX was designed to compress Microsoft cabinet files, which are part of the Windows operating system. Computers using this system are generally based on microprocessors made by Intel, and throughout the 1980s and 1990s, before the introduction of the pentium, these microprocessors were members of the well-known 80x86 family. The encoder preprocessing mode of LZX is selected by the user when the input stream is an executable file for an 80x86 computer. This mode converts 80x86 CALL instructions to use absolute instead of relative addresses.

Output block format. LZX outputs the compressed data in blocks, where each block contains raw characters, offsets, match lengths, and the canonical Huffman trees used to encode the three types of data. A canonical Huffman tree can be reconstructed from the path length of each of its nodes. Thus, only the path lengths have to be written on the output for each Huffman tree. LZX limits the depth of a Huffman tree to 16, so each tree node is represented on the output by a number in the range 0 to 16. A 0 indicates a missing node (a Huffman code that's not used by this block). If the tree has to be bigger, the current block is written on the output and compression resumes with fresh trees. The tree nodes are written in compressed form. If several consecutive tree nodes are identical, then run-length encoding is used to encode them. The three numbers 17, 18, and 19 are used for this purpose. Otherwise the difference (modulo 17) between the path lengths of the current node and the previous node is written. This difference is in the interval $[0, 16]$. Thus, what's written on the output are the 20 5-bit integers 0 through 19, and these integers are themselves encoded by a Huffman tree called a pre-tree. The pre-tree is generated dynamically according to the frequencies of occurrence of the 20 values. The pre-tree itself has to be written on the output, and it is written as 20 4-bit integers (a total of 80 bits) where each integer indicates the path length of one of the 20 tree nodes. A path of length zero indicates that one of the 20 values is not used in the current block.

The offsets and match lengths are themselves compressed in a complex process that involves several steps and is summarized in Figure 6.13. The individual steps involve many operations and use several tables that are built into both encoder and decoder. However, because LZX is not an important compression method, these steps are not discussed here.

6.9 LZ78

The LZ78 method (which is sometimes referred to as LZ2) [Ziv and Lempel 78] does not use any search buffer, look-ahead buffer, or sliding window. Instead, there is a dictionary of previously encountered strings. This dictionary starts empty (or almost empty), and its size is limited only by the amount of available memory. The encoder outputs two-field tokens. The first field is a pointer to the dictionary; the second is the code of a symbol. Tokens do not contain the length of a string, since this is implied in the dictionary. Each token corresponds to a string of input symbols, and that string is added to the dictionary after the token is written on the compressed stream. Nothing is ever deleted from the dictionary, which is both an advantage over LZ77 (since future strings can be compressed even by strings seen in the distant past) and a liability (because the dictionary tends to grow fast and to fill up the entire available memory).

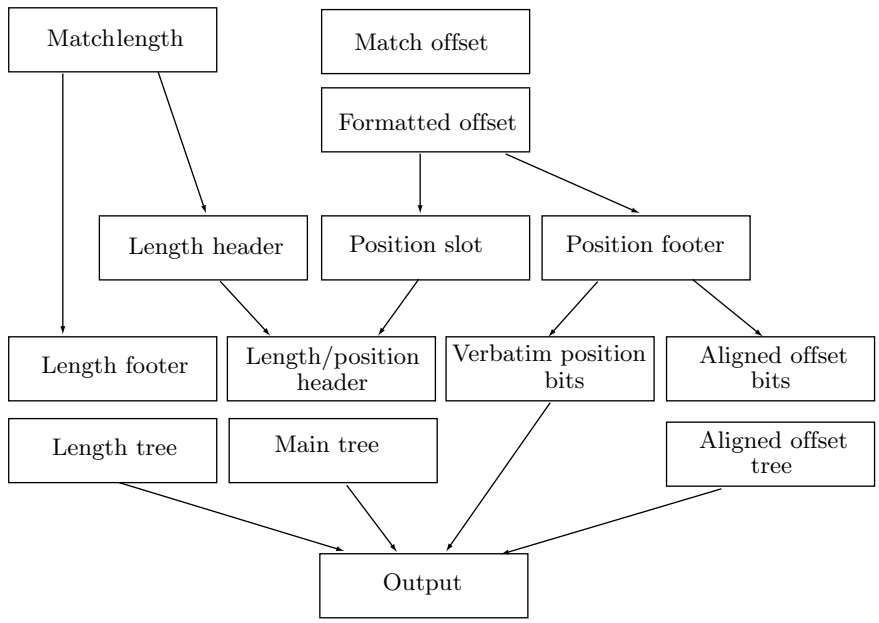


Figure 6.13: LZX Processing of Offsets and Lengths.

The dictionary starts with the null string at position zero. As symbols are input and encoded, strings are added to the dictionary at positions 1, 2, and so on. When the next symbol `x` is read from the input stream, the dictionary is searched for an entry with the one-symbol string `x`. If none are found, `x` is added to the next available position in the dictionary, and the token $(0, \texttt{x})$ is output. This token indicates the string "null `x`" (a concatenation of the null string and `x`). If an entry with `x` is found (at, say, position 37), the next symbol `y` is read, and the dictionary is searched for an entry containing the two-symbol string `xy`. If none are found, then string `xy` is added to the next available position in the dictionary, and the token $(37, \texttt{y})$ is output. This token indicates the string `xy`, since 37 is the dictionary position of string `x`. The process continues until the end of the input stream is reached.

In general, the current symbol is read and becomes a one-symbol string. The encoder then tries to find it in the dictionary. If the symbol is found in the dictionary, the next symbol is read and concatenated with the first to form a two-symbol string that the encoder then tries to locate in the dictionary. As long as those strings are found in the dictionary, more symbols are read and concatenated to the string. At a certain point the string is not found in the dictionary, so the encoder adds it to the dictionary and outputs a token with the last dictionary match as its first field, and the last symbol of the string (the one that caused the search to fail) as its second field. Table 6.14 shows the first 14 steps in encoding the string

`sir␣sid␣eastman␣easily␣teases␣sea␣sick␣seals`

◇ **Exercise 6.7:** Complete Table 6.14.

In each step, the string added to the dictionary is the one being encoded, minus

Dictionary		Token	Dictionary		Token
0	null				
1	"s"	(0,"s")	8	"a"	(0,"a")
2	"i"	(0,"i")	9	"st"	(1,"t")
3	"r"	(0,"r")	10	"m"	(0,"m")
4	"␣"	(0,"␣")	11	"an"	(8,"n")
5	"si"	(1,"i")	12	"␣ea"	(7,"a")
6	"d"	(0,"d")	13	"sil"	(5,"l")
7	"␣e"	(4,"e")	14	"y"	(0,"y")

Table 6.14: First 14 Encoding Steps in LZ78.

its last symbol. In a typical compression run, the dictionary starts with short strings, but as more text is being input and processed, longer and longer strings are added to it. The size of the dictionary can either be fixed or may be determined by the size of the available memory each time the LZ78 compression program is executed. A large dictionary may contain more strings and thus allow for longer matches, but the trade-off is longer pointers (and thus bigger tokens) and slower dictionary search.

A good data structure for the dictionary is a tree, but not a binary tree. The tree starts with the null string as the root. All the strings that start with the null string (strings for which the token pointer is zero) are added to the tree as children of the root. In the above example those are `s`, `i`, `r`, ␣, `d`, `a`, `m`, `y`, `e`, `c`, and `k`. Each of them becomes the root of a subtree as shown in Figure 6.15. For example, all the strings that start with `s` (the four strings `si`, `sil`, `st`, and `s(eof)`) constitute the subtree of node `s`.

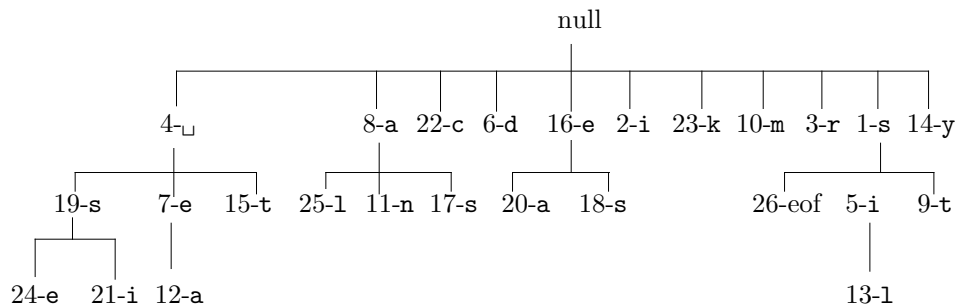


Figure 6.15: An LZ78 Dictionary Tree.

Assuming an alphabet with 8-bit symbols, there are 256 different symbols, so in principle, each node in the tree could have up to 256 children. The process of adding a child to a tree node should thus be dynamic. When the node is first created, it has no children and it should not reserve any memory space for them. As a child is added to the node, memory space should be claimed for it. Since no nodes are ever deleted, there is no need to reclaim memory space, which simplifies the memory management task somewhat.

Such a tree makes it easy to search for a string and to add strings. To search for `sil`, for example, the program looks for the child `s` of the root, then for the child `i` of `s`, and so on, going down the tree. Here are some examples:

1. When the `s` of `sid` is input in step 5, the encoder finds node "1-`s`" in the tree as a child of "null". It then inputs the next symbol `i`, but node `s` does not have a child `i` (in fact, it has no children at all at this point), so the encoder adds node "5-`i`" as a child of "1-`s`", which effectively adds the string `si` to the tree.

2. When the blank space between `eastman` and `easily` is input in step 12, a similar situation happens. The encoder finds node "4-␣", inputs `e`, finds "7-`e`", inputs `a`, but "7-`e`" does not have "`a`" as a child, so the encoder adds node "12-`a`", which effectively adds the string "␣`ea`" to the tree.

A tree of the type described here is called a *trie*. In general, a trie is a tree in which the branching structure at any level is determined by just part of a data item, not the entire item (Section 5.14.5). In the case of LZ78, each string added to the tree effectively adds just one symbol, and does that by adding a branch.

Since the total size of the tree is limited, it may fill up during compression. This, in fact, happens all the time except when the input stream is unusually small. The original LZ78 method does not specify what to do in such a case, so we list a few possible solutions.

1. The simplest solution is to freeze the dictionary at that point. No new nodes should be added, the tree becomes a static dictionary, but it can still be used to encode strings.

2. Delete the entire tree once it gets full and start with a new, empty tree. This solution effectively breaks the input into blocks, each with its own dictionary. If the content of the input varies from block to block, this solution will produce good compression, since it will eliminate a dictionary with strings that are unlikely to be used in the future. We can say that this solution implicitly assumes that future symbols will benefit more from new data than from old (the same implicit assumption used by LZ77).

3. The UNIX `compress` utility (Section 6.14) uses a more complex solution.

4. When the dictionary is full, delete some of the least-recently-used entries, to make room for new ones. Unfortunately there is no good algorithm to decide which entries to delete, and how many (but see the *reuse* procedure in Section 6.23).

The LZ78 decoder works by building and maintaining the dictionary in the same way as the encoder. It is therefore more complex than the LZ77 decoder. Thus, LZ77 decoding is simpler than its encoding, but LZ78 encoding and decoding involve the same complexity.

6.10 LZFG

Edward Fiala and Daniel Greene have developed several related compression methods [Fiala and Greene 89] that are hybrids of LZ77 and LZ78. All their methods are based on the following scheme. The encoder generates a compressed file with tokens and literals (raw ASCII codes) intermixed. There are two types of tokens: a *literal* and a *copy*. A literal token indicates that a string of literals follows, a copy token points to a string previously seen in the data. The string `the␣boy␣on␣my␣right␣is␣the␣right␣boy` produces, when encoded,

(literal 23)`the␣boy␣on␣my␣right␣is␣`(copy 4 23)(copy 6 13)(copy 3 29),

where the three copy tokens refer to the strings `the␣`, `right␣`, and "`boy`", respectively. The LZFG methods are best understood by considering how the decoder operates. The decoder starts with a large empty buffer in which it generates and shifts the decompressed stream. When the decoder inputs a (literal 23) token, it inputs the next 23 bytes as raw ASCII codes into the buffer, shifting the buffer such that the last byte input will be the rightmost one. When the decoder inputs (copy 4 23) it copies the string of length 4 that starts 23 positions from the right end of the buffer. The string is then appended to the buffer, while shifting it. Two LZFG variants, denoted by A1 and A2, are described here.

The A1 scheme employs 8-bit literal tokens and 16-bit copy tokens. A literal token has the format `0000nnnn`, where `nnnn` indicates the number of ASCII bytes following the token. Since the 4-bit `nnnn` field can have values between 0 and 15, they are interpreted as meaning 1 to 16. The longest possible string of literals is therefore 16 bytes. The format of a copy token is `sssspp...p`, where the 4-bit nonzero `ssss` field indicates the length of the string to be copied, and the 12-bit `pp...p` field is a displacement showing where the string starts in the buffer. Since the `ssss` field cannot be zero, it can have values only between 1 and 15, and they are interpreted as string lengths between 2 and 16. Displacement values are in the range $[0, 4095]$ and are interpreted as $[1, 4096]$.

The encoder starts with an empty search buffer, 4,096 bytes long, and fills up the look-ahead buffer with input data. At each subsequent step it tries to create a copy token. If nothing matches in that step, the encoder creates a literal token. Suppose that at a certain point the buffer contains

←text already encoded..`..  ..xyz`|`abcd..  ..`..← text yet to be input

The encoder tries to match the string `abc...` in the look-ahead buffer to various strings in the search buffer. If a match is found (of at least two symbols), a copy token is written on the compressed stream and the data in the buffers is shifted to the left by the size of the match. If a match is not found, the encoder starts a literal with the `a` and left-shifts the data one position. It then tries to match `bcd...` to the search buffer. If it finds a match, a literal token is output, followed by a byte with the `a`, followed by a match token. Otherwise, the `b` is appended to the literal and the encoder tries to match from `cd...` Literals can be up to 16 bytes long, so the string `the␣boy␣on␣my...` above is encoded as

(literal 16)`the␣boy␣on␣my␣ri`(literal 7)`ght␣is␣`(copy 4 23)(copy 6 13)(copy 3 29).

The A1 method borrows the idea of the sliding buffer from LZ77 but also behaves like LZ78, because it creates two-field tokens. This is why it can be considered a hybrid

of the two original LZ methods. When A1 starts, it creates mostly literals, but when it gets up to speed (fills up its search buffer), it features strong adaptation, so more and more copy tokens appear in the compressed stream.

The A2 method uses a larger search buffer (up to 21K bytes long). This improves compression, because longer copies can be found, but raises the problem of token size. A large search buffer implies large displacements in copy tokens; long copies imply large "length" fields in those tokens. At the same time we expect both the displacement and the "length" fields of a typical copy token to be small, since most matches are found close to the beginning of the search buffer. The solution is to use a variable-length code for those fields, and A2 uses the general unary codes of Section 3.1. The "length" field of a copy token is encoded with a (2,1,10) code (Table 3.4), making it possible to match strings up to 2,044 symbols long. Notice that the (2,1,10) code is between 3 and 18 bits long.

The first four codes of the $(2, 1, 10)$ code are 000, 001, 010, and 011. The last three of these codes indicate match lengths of two, three, and four, respectively (recall that the minimum match length is 2). The first one (code 000) is reserved to indicate a literal. The length of the literal then follows and is encoded with code $(0, 1, 5)$. A literal can therefore be up to 63 bytes long, and the literal-length field in the token is encoded by between 1 and 10 bits. In case of a match, the "length" field is not 000 and is followed by the displacement field, which is encoded with the (10,2,14) code (Table 6.16). This code has 21K values, and the maximum code size is 16 bits (but see points 2 and 3 below).

n	$a = 10 + n \cdot 2$	nth codeword	Number of codewords	Range of integers
0	10	$0\underbrace{x...x}_{10}$	$2^{10} = 1K$	0–1023
1	12	$10\underbrace{xx...x}_{12}$	$2^{12} = 4K$	1024–5119
2	14	$11\underbrace{xx...xx}_{14}$	$2^{14} = 16K$	5120–21503
		Total	21504	

Table 6.16: The General Unary Code $(10, 2, 14)$.

Three more refinements are employed by the A2 method, to achieve slightly better (1% or 2%) compression.

1. A literal of maximum length (63 bytes) can immediately be followed by another literal or by a copy token of any length, but a literal of fewer than 63 bytes must be followed by a copy token matching *at least three symbols* (or by the end-of-file). This fact is used to shift down the (2,1,10) codes used to indicate the match length. Normally, codes 000, 001, 010, and 011 indicate no match, and matches of length 2, 3, and 4, respectively. However, a copy token following a literal token of fewer than 63 bytes uses codes 000, 001, 010, and 011 to indicate matches of length 3, 4, 5, and 6, respectively. This way the maximum match length can be 2,046 symbols instead of 2,044.
2. The displacement field is encoded with the $(10, 2, 14)$ code, which has 21K values and whose individual codes range in size from 11 to 16 bits. For smaller files, such large

displacements may not be necessary, and other general unary codes may be used, with shorter individual codes. Method A2 thus uses codes of the form $(10-d, 2, 14-d)$ for $d = 10, 9, 8, \ldots, 0$. For $d = 1$, code $(9, 2, 13)$ has $2^9 + 2^{11} + 2^{13} = 10{,}752$ values, and individual codes range in size from 9 to 15 bits. For $d = 10$ code $(0, 2, 4)$ contains $2^0 + 2^2 + 2^4 = 21$ values, and codes are between 1 and 6 bits long. Method A2 starts with $d = 10$ [meaning it initially uses code $(0, 2, 4)$] and a search buffer of size 21 bytes. When the buffer fills up (indicating an input stream longer than 21 bytes), the A2 algorithm switches to $d = 9$ [code $(1, 2, 5)$] and increases the search buffer size to 42 bytes. This process continues until the entire input stream has been encoded or until $d = 0$ is reached [at which point code (10,2,14) is used to the end]. A lot of work for a small gain in compression! (See the discussion of diminishing returns (*a word to the wise*) in the Preface.)

3. Each of the codes $(10-d, 2, 14-d)$ requires a search buffer of a certain size, from 21 up to 21K = 21,504 bytes, according to the number of codes it contains. If the user wants, for some reason, to assign the search buffer a different size, then some of the longer codes may never be used, which makes it possible to cut down a little the size of the individual codes. For example, if the user decides to use a search buffer of size 16K = 16,384 bytes, then code $(10, 2, 14)$ has to be used [because the next code $(9, 2, 13)$ contains just 10,752 values]. Code $(10, 2, 14)$ contains 21K = 21,504 individual codes, so the 5,120 longest codes will never be used. The last group of codes ("11" followed by 14 bits) in $(10, 2, 14)$ contains $2^{14} = 16,384$ different individual codes, of which only 11,264 will be used. Of the 11,264 codes the first 8,192 can be represented as "11" followed by $\lfloor \log_2 11,264 \rfloor = 13$ bits, and only the remaining 3,072 codes require $\lceil \log_2 11,264 \rceil = 14$ bits to follow the first "11". We thus end up with 8,192 15-bit codes and 3,072 16-bit codes, instead of 11,264 16-bit codes, a very small improvement.

These three improvements illustrate the great lengths that researchers are willing to go to in order to improve their algorithms ever so slightly.

> Experience shows that fine-tuning an algorithm to squeeze out the last remaining bits of redundancy from the data gives diminishing returns. Modifying an algorithm to improve compression by 1% may increase the run time by 10% (from the Introduction).

The LZFG "corpus" of algorithms contains four more methods. B1 and B2 are similar to A1 and A2 but faster because of the way they compute displacements. However, some compression ratio is sacrificed. C1 and C2 go in the opposite direction. They achieve slightly better compression than A1 and A2 at the price of slower operation. (LZFG has been patented, an issue that's discussed in Section 6.34.)

6.11 LZRW1

Developed by Ross Williams [Williams 91a] and [Williams 91b] as a simple, fast LZ77 variant, LZRW1 is also related to method A1 of LZFG (Section 6.10). The main idea is to find a match in one step, using a hash table. This is fast but not very efficient, since the match found is not always the longest. We start with a description of the algorithm, follow with the format of the compressed stream, and conclude with an example.

The method uses the entire available memory as a buffer and encodes the input stream in blocks. A block is read into the buffer and is completely encoded, then the next block is read and encoded, and so on. The length of the search buffer is 4K and that of the look-ahead buffer is 16 bytes. These two buffers slide along the input block in memory from left to right. Only one pointer, `p_src`, needs be maintained, pointing to the start of the look-ahead buffer. The pointer `p_src` is initialized to 1 and is incremented after each phrase is encoded, thereby moving both buffers to the right by the length of the phrase. Figure 6.17 shows how the search buffer starts empty, grows to 4K, and then starts sliding to the right, following the look-ahead buffer.

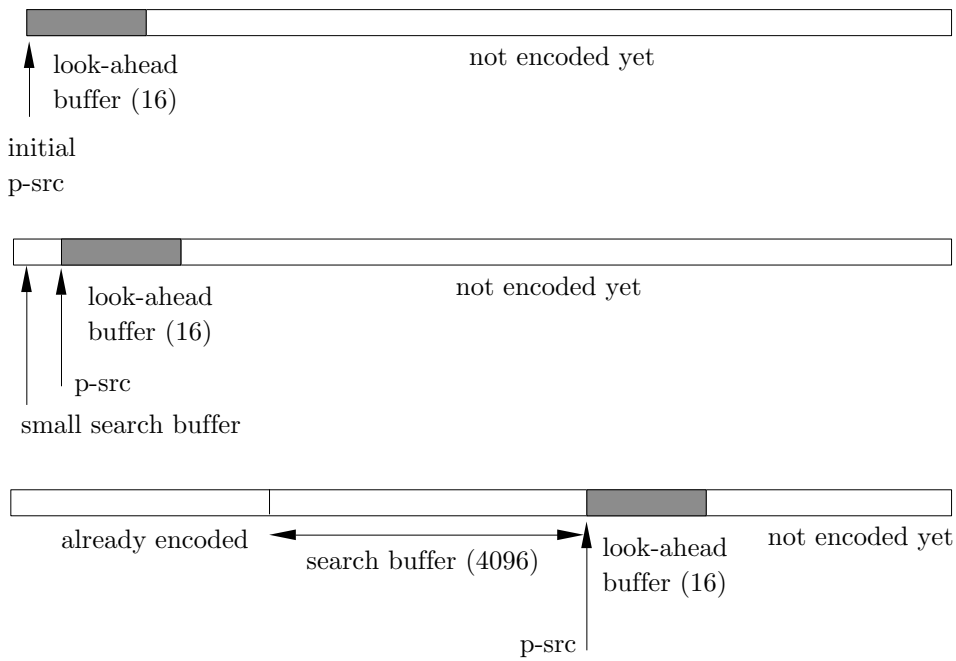


Figure 6.17: Sliding the LZRW1 Search- and Look-Ahead Buffers.

The leftmost three characters of the look-ahead buffer are hashed into a 12-bit number I, which is used to index an array of $2^{12} = 4{,}096$ pointers. A pointer P is retrieved and is immediately replaced in the array by `p_src`. If P points outside the search buffer, there is no match; the first character in the look-ahead buffer is output as a literal, and `p_src` is advanced by 1. The same thing is done if P points inside the

search buffer but to a string that does not match the one in the look-ahead buffer. If P points to a match of at least three characters, the encoder finds the longest match (at most 16 characters), outputs a match item, and advances `p_src` by the length of the match. This process is depicted in Figure 6.19. An interesting point to note is that the array of pointers does not have to be initialized when the encoder starts, since the encoder checks every pointer. Initially, all pointers are random, but as they are replaced, more and more of them point to real matches.

The output of the LZRW1 encoder (Figure 6.20) consists of groups, each starting with a 16-bit *control word*, followed by 16 items. Each item is either an 8-bit literal or a 16-bit copy item (a match) consisting of a 4-bit length field b (where the length is $b+1$) and a 12-bit offset (the a and c fields). The length field indicates lengths between 3 and 16. The 16 bits of the control word flag each of the 16 items that follow (a 0 flag indicates a literal and a flag of 1 indicates a match item). Obviously, groups have different lengths. The last group may contain fewer than 16 items.

The decoder is even simpler than the encoder, since it does not need the array of pointers. It maintains a large buffer using a `p_src` pointer in the same way as the encoder. The decoder reads a control word from the compressed stream and uses its 16 bits to read 16 items. A literal item is decoded by appending it to the buffer and incrementing `p_src` by 1. A copy item is decoded by subtracting the offset from `p_src`, fetching a string from the search buffer, of length indicated by the length field, and appending it to the buffer. Then `p_src` is incremented by the length.

Table 6.18 illustrates the first seven steps of encoding "`that␣thatch␣thaws`". The values produced by the hash function are arbitrary. Initially, all pointers are random (indicated by "any") but they are replaced by useful ones very quickly.

◇ **Exercise 6.8:** Summarize the last steps in a table similar to Table 6.18 and write the final compressed stream in binary.

`p_src`	3 chars	Hash index	P	Output	Binary output
1	`tha`	4	any→1	`t`	01110100
2	`hat`	6	any→2	`h`	01101000
3	`at␣`	2	any→3	`a`	01100001
4	`t␣t`	1	any→4	`t`	01110100
5	`␣th`	5	any→5	`␣`	00100000
6	`tha`	4	4→1	6,5	0000|0011|00000101
10	`ch␣`	3	any→10	`c`	01100011

Table 6.18: First Seven Steps of Encoding `that thatch thaws`.

Tests done by the original developer indicate that LZRW1 performs about 10% worse than LZC (the UNIX `compress` utility) but is four times faster. Also, it performs about 4% worse than LZFG (the A1 method) but runs ten times faster. It is therefore suited for cases where speed is more important than compression performance. A 68000

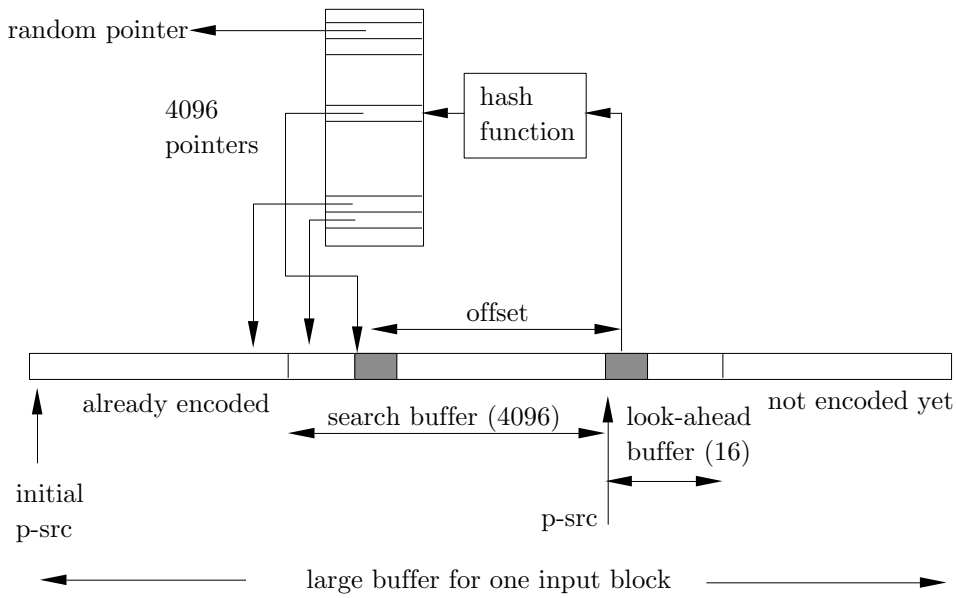


Figure 6.19: The LZRW1 Encoder.

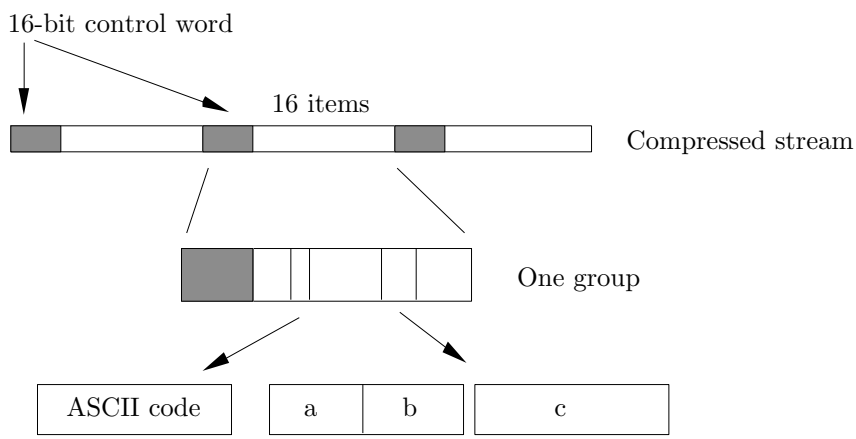


Figure 6.20: Format of the Output.

assembly language implementation has required, on average, the execution of only 13 machine instructions to compress, and four instructions to decompress, one byte.

◇ **Exercise 6.9:** Show a practical situation where compression speed is more important than compression ratio.

6.12 LZRW4

LZRW4 is a variant of LZ77, based on ideas of Ross Williams about possible ways to combine a dictionary method with prediction (Section 6.35). LZRW4 also borrows some ideas from LZRW1. It uses a 1 Mbyte buffer where both the search and look-ahead buffers slide from left to right. At any point in the encoding process, the order-2 context of the current symbol (the two most-recent symbols in the search buffer) is used to predict the current symbol. The two symbols constituting the context are hashed to a 12-bit number I, which is used as an index to a $2^{12} = 4{,}096$-entry array A of partitions. Each partition contains 32 pointers to the input data in the 1 Mbyte buffer (each pointer is therefore 20 bits long).

The 32 pointers in partition $A[I]$ are checked to find the longest match between the look-ahead buffer and the input data seen so far. The longest match is selected and is coded in 8 bits. The first 3 bits code the match length according to Table 6.21; the remaining 5 bits identify the pointer in the partition. Such an 8-bit number is called a *copy item*. If no match is found, a literal is encoded in 8 bits. For each item, an extra bit is prepared, a 0 for a literal and a 1 for a copy item. The extra bits are accumulated in groups of 16, and each group is output, as in LZRW1, preceding the 16 items it refers to.

3 bits:	000	001	010	011	100	101	110	111
length:	2	3	4	5	6	7	8	16

Table 6.21: Encoding the Length in LZRW4.

The partitions are updated all the time by moving "good" pointers toward the start of their partition. When a match is found, the encoder swaps the selected pointer with the pointer halfway toward the partition (Figure 6.22a,b). If no match is found, the entire 32-pointer partition is shifted to the left and the new pointer is entered on the right, pointing to the current symbol (Figure 6.22c).

> The Red Queen shook her head, "You may call it 'nonsense' if you like," she said, "but I've heard nonsense, compared with which that would be as sensible as a dictionary!"
>
> —Lewis Carroll, *Through the Looking Glass* (1872)

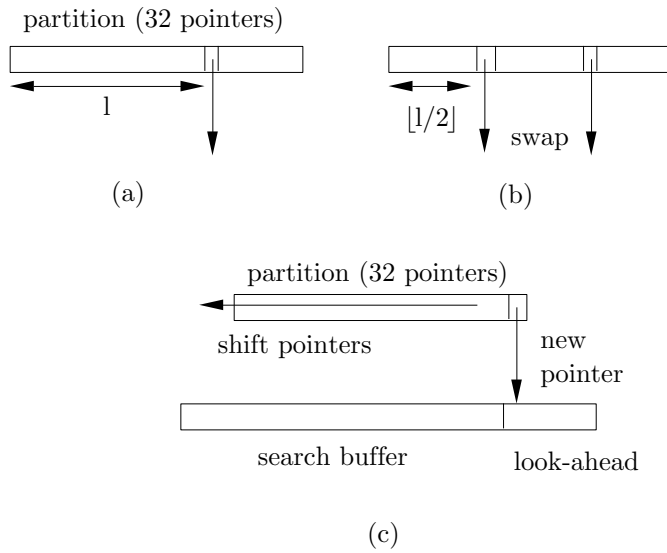


Figure 6.22: Updating an LZRW4 Partition.

6.13 LZW

This is a popular variant of LZ78, developed by Terry Welch in 1984 ([Welch 84] and [Phillips 92]). Its main feature is eliminating the second field of a token. An LZW token consists of just a pointer to the dictionary. To best understand LZW, we will temporarily forget that the dictionary is a tree, and will think of it as an array of variable-size strings. The LZW method starts by initializing the dictionary to all the symbols in the alphabet. In the common case of 8-bit symbols, the first 256 entries of the dictionary (entries 0 through 255) are occupied before any data is input. Because the dictionary is initialized, the next input character will always be found in the dictionary. This is why an LZW token can consist of just a pointer and does not have to contain a character code as in LZ77 and LZ78.

(LZW has been patented and for many years its use required a license. This issue is treated in Section 6.34.)

The principle of LZW is that the encoder inputs symbols one by one and accumulates them in a string `I`. After each symbol is input and is concatenated to `I`, the dictionary is searched for string `I`. As long as `I` is found in the dictionary, the process continues. At a certain point, adding the next symbol `x` causes the search to fail; string `I` is in the dictionary but string `Ix` (symbol `x` concatenated to `I`) is not. At this point the encoder (1) outputs the dictionary pointer that points to string `I`, (2) saves string `Ix` (which is now called a *phrase*) in the next available dictionary entry, and (3) initializes string `I` to symbol `x`. To illustrate this process, we again use the text string `sir␣sid␣eastman␣easily␣teases␣sea␣sick␣seals`. The steps are as follows:

0. Initialize entries 0–255 of the dictionary to all 256 8-bit bytes.
1. The first symbol `s` is input and is found in the dictionary (in entry 115, since this is

the ASCII code of s). The next symbol i is input, but si is not found in the dictionary. The encoder performs the following: (1) outputs 115, (2) saves string si in the next available dictionary entry (entry 256), and (3) initializes I to the symbol i.
2. The r of sir is input, but string ir is not in the dictionary. The encoder (1) outputs 105 (the ASCII code of i), (2) saves string ir in the next available dictionary entry (entry 257), and (3) initializes I to the symbol r.

Table 6.23 summarizes all the steps of this process. Table 6.24 shows some of the original 256 entries in the LZW dictionary plus the entries added during encoding of the string above. The complete output stream is (only the numbers are output, not the strings in parentheses) as follows:

115 (s), 105 (i), 114 (r), 32 (␣), 256 (si), 100 (d), 32 (␣), 101 (e), 97 (a), 115 (s), 116 (t), 109 (m), 97 (a), 110 (n), 262 (␣e), 264 (as), 105 (i), 108 (l), 121 (y),
32 (␣), 116 (t), 263 (ea), 115 (s), 101 (e), 115 (s), 259 (␣s), 263 (ea), 259 (␣s), 105 (i), 99 (c), 107 (k), 280 (␣se), 97 (a), 108 (l), 115 (s), eof.

Figure 6.25 is a pseudo-code listing of the algorithm. We denote by λ the empty string, and by <<a,b>> the concatenation of strings a and b.

The line "append <<di,ch>> to the dictionary" is of special interest. It is clear that in practice, the dictionary may fill up. This line should therefore include a test for a full dictionary, and certain actions for the case where it is full.

Since the first 256 entries of the dictionary are occupied right from the start, pointers to the dictionary have to be longer than 8 bits. A simple implementation would typically use 16-bit pointers, which allow for a 64K-entry dictionary (where $64\text{K} = 2^{16} = 65{,}536$). Such a dictionary will, of course, fill up very quickly in all but the smallest compression jobs. The same problem exists with LZ78, and any solutions used with LZ78 can also be used with LZW. Another interesting fact about LZW is that strings in the dictionary become only one character longer at a time. It therefore takes a long time to end up with long strings in the dictionary, and thus a chance to achieve really good compression. We can say that LZW adapts slowly to its input data.

◇ **Exercise 6.10:** Use LZW to encode the string alf␣eats␣alfalfa. Show the encoder output and the new entries added by it to the dictionary.

◇ **Exercise 6.11:** Analyze the LZW compression of the string "aaaa...".

A dirty icon (anagram of "dictionary")

6.13.1 LZW Decoding

To understand how the LZW decoder works, we recall the three steps the encoder performs each time it writes something on the output stream. They are (1) it outputs the dictionary pointer that points to string I, (2) it saves string Ix in the next available entry of the dictionary, and (3) it initializes string I to symbol x.

The decoder starts with the first entries of its dictionary initialized to all the symbols of the alphabet (normally 256 symbols). It then reads its input stream (which consists of pointers to the dictionary) and uses each pointer to retrieve uncompressed symbols from its dictionary and write them on its output stream. It also builds its dictionary in

I	in dict?	new entry	output	I	in dict?	new entry	output
s	Y			y	Y		
si	N	256-si	115 (s)	y␣	N	274-y␣	121 (y)
i	Y			␣	Y		
ir	N	257-ir	105 (i)	␣t	N	275-␣t	32 (␣)
r	Y			t	Y		
r␣	N	258-r␣	114 (r)	te	N	276-te	116 (t)
␣	Y			e	Y		
␣s	N	259-␣s	32 (␣)	ea	Y		
s	Y			eas	N	277-eas	263 (ea)
si	Y			s	Y		
sid	N	260-sid	256 (si)	se	N	278-se	115 (s)
d	Y			e	Y		
d␣	N	261-d␣	100 (d)	es	N	279-es	101 (e)
␣	Y			s	Y		
␣e	N	262-␣e	32 (␣)	s␣	N	280-s␣	115 (s)
e	Y			␣	Y		
ea	N	263-ea	101 (e)	␣s	Y		
a	Y			␣se	N	281-␣se	259 (␣s)
as	N	264-as	97 (a)	e	Y		
s	Y			ea	Y		
st	N	265-st	115 (s)	ea␣	N	282-ea␣	263 (ea)
t	Y			␣	Y		
tm	N	266-tm	116 (t)	␣s	Y		
m	Y			␣si	N	283-␣si	259 (␣s)
ma	N	267-ma	109 (m)	i	Y		
a	Y			ic	N	284-ic	105 (i)
an	N	268-an	97 (a)	c	Y		
n	Y			ck	N	285-ck	99 (c)
n␣	N	269-n␣	110 (n)	k	Y		
␣	Y			k␣	N	286-k␣	107 (k)
␣e	Y			␣	Y		
␣ea	N	270-␣ea	262 (␣e)	␣s	Y		
a	Y			␣se	Y		
as	Y			␣sea	N	287-␣sea	281 (␣se)
asi	N	271-asi	264 (as)	a	Y		
i	Y			al	N	288-al	97 (a)
il	N	272-il	105 (i)	l	Y		
l	Y			ls	N	289-ls	108 (l)
ly	N	273-ly	108 (l)	s	Y		
				s,eof	N		115 (s)

Table 6.23: Encoding `sir sid eastman easily teases sea sick seals`.

0	NULL	110	n	262	␣e	276	te
1	SOH	...		263	ea	277	eas
...		115	s	264	as	278	se
32	SP	116	t	265	st	279	es
...		...		266	tm	280	s
97	a	121	y	267	ma	281	␣se
98	b	...		268	an	282	ea␣
99	c	255	255	269	n␣	283	␣si
100	d	256	si	270	␣ea	284	ic
101	e	257	ir	271	asi	285	ck
...		258	r␣	272	il	286	k␣
107	k	259	␣s	273	ly	287	␣sea
108	l	260	sid	274	y␣	288	al
109	m	261	d␣	275	␣t	289	ls

Table 6.24: An LZW Dictionary.

```
for i:=0 to 255 do
  append i as a 1-symbol string to the dictionary;
append λ to the dictionary;
di:=dictionary index of λ;
repeat
  read(ch);
  if <<di,ch>> is in the dictionary then
    di:=dictionary index of <<di,ch>>;
  else
    output(di);
    append <<di,ch>> to the dictionary;
    di:=dictionary index of ch;
  endif;
until end-of-input;
```

Figure 6.25: The LZW Algorithm.

the same way as the encoder (this fact is usually expressed by saying that the encoder and decoder are *synchronized*, or that they work in *lockstep*).

In the first decoding step, the decoder inputs the first pointer and uses it to retrieve a dictionary item `I`. This is a string of symbols, and it is written on the decoder's output. String `Ix` needs to be saved in the dictionary, but symbol `x` is still unknown; it will be the first symbol in the next string retrieved from the dictionary.

In each decoding step after the first, the decoder inputs the next pointer, retrieves the next string `J` from the dictionary, writes it on the output, isolates its first symbol `x`, and saves string `Ix` in the next available dictionary entry (after checking to make sure string `Ix` is not already in the dictionary). The decoder then moves `J` to `I` and is ready for the next step.

In our `sir␣sid...` example, the first pointer that's input by the decoder is 115. This corresponds to the string `s`, which is retrieved from the dictionary, gets stored in `I`, and becomes the first item written on the decoder's output. The next pointer is 105, so string `i` is retrieved into `J` and is also written on the output. `J`'s first symbol is concatenated with `I`, to form string `si`, which does not exist in the dictionary, and is therefore added to it as entry 256. Variable `J` is moved to `I`, so `I` is now the string `i`. The next pointer is 114, so string `r` is retrieved from the dictionary into `J` and is also written on the output. `J`'s first symbol is concatenated with `I`, to form string `ir`, which does not exist in the dictionary, and is added to it as entry 257. Variable `J` is moved to `I`, so `I` is now the string `r`. The next step reads pointer 32, writes ␣ on the output, and saves string `r ␣`.

◇ **Exercise 6.12:** Decode the string `alf␣eats␣alfalfa` by using the encoding results from Exercise 6.10.

◇ **Exercise 6.13:** Assume a two-symbol alphabet with the symbols `a` and `b`. Show the first few steps for encoding and decoding the string "`ababab...`".

6.13.2 LZW Dictionary Structure

Up until now, we have assumed that the LZW dictionary is an array of variable-size strings. To understand why a trie is a better data structure for the dictionary we need to recall how the encoder works. It inputs symbols and concatenates them into a variable `I` as long as the string in `I` is found in the dictionary. At a certain point the encoder inputs the first symbol `x`, which causes the search to fail (string `Ix` is not in the dictionary). It then adds `Ix` to the dictionary. This means that each string added to the dictionary effectively adds just one new symbol, `x`. (Phrased another way; for each dictionary string of more than one symbol, there exists a "parent" string in the dictionary that's one symbol shorter.)

A tree similar to the one used by LZ78 is therefore a good data structure, because adding string `Ix` to such a tree is done by adding one node with x. The main problem is that each node in the LZW tree may have many children (this is a multiway tree, not a binary tree). Imagine the node for the letter `a` in entry 97. Initially it has no children, but if the string `ab` is added to the tree, node 97 gets one child. Later, when, say, the string `ae` is added, node 97 gets a second child, and so on. The data structure for the tree should therefore be designed such that a node could have any number of children, but without having to reserve any memory for them in advance.

One way of designing such a data structure is to house the tree in an array of nodes, each a structure with two fields: a symbol and a pointer to the parent node. A node has no pointers to any child nodes. Moving down the tree, from a node to one of its children, is done by a *hashing process* in which the pointer to the node and the symbol of the child are hashed to create a new pointer.

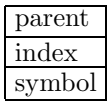
Suppose that string `abc` has already been input, symbol by symbol, and has been stored in the tree in the three nodes at locations 97, 266, and 284. Following that, the encoder has just input the next symbol `d`. The encoder now searches for string `abcd`, or, more specifically, for a node containing the symbol `d` whose parent is at location 284. The encoder hashes the 284 (the pointer to string `abc`) and the 100 (ASCII code of `d`) to create a pointer to some node, say, 299. The encoder then examines node 299. There are three possibilities:

1. The node is unused. This means that `abcd` is not yet in the dictionary and should be added to it. The encoder adds it to the tree by storing the parent pointer 284 and ASCII code 100 in the node. The result is the following:

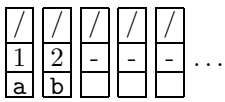
Node					
Address	:	97	266	284	299
Contents	:	(-:a)	(97:b)	(266:c)	(284:d)
Represents:		a	ab	abc	abcd

2. The node contains a parent pointer of 284 and the ASCII code of `d`. This means that string `abcd` is already in the tree. The encoder inputs the next symbol, say `e`, and searches the dictionary tree for string `abcde`.
3. The node contains something else. This means that another hashing of a pointer and an ASCII code has resulted in 299, and node 299 already contains information from another string. This is called a *collision*, and it can be dealt with in several ways. The simplest way to deal with a collision is to increment pointer 299 and examine nodes 300, 301,... until an unused node is found, or until a node with (284:`d`) is found.

In practice, we build nodes that are structures with three fields, a pointer to the parent node, the pointer (or index) created by the hashing process, and the code (normally ASCII) of the symbol contained in the node. The second field is necessary because of collisions. A node can therefore be illustrated by



We illustrate this data structure using string `ababab...` of Exercise 6.13. The dictionary is an array `dict` where each entry is a structure with the three fields `parent`, `index`, and `symbol`. We refer to a field by, for example, `dict[pointer].parent`, where `pointer` is an index to the array. The dictionary is initialized to the two entries `a` and `b`. (To keep the example simple we use no ASCII codes. We assume that `a` has code 1 and `b` has code 2.) The first few steps of the encoder are as follows:
Step 0: Mark all dictionary locations from 3 on as unused.



Step 1: The first symbol `a` is input into variable `I`. What is actually input is the code of `a`, which in our example is 1, so `I` = 1. Since this is the first symbol, the encoder assumes that it is in the dictionary and so does not perform any search.
Step 2: The second symbol `b` is input into `J`, so `J` = 2. The encoder has to search for string `ab` in the dictionary. It executes `pointer:=hash(I,J)`. Let's assume that the result is 5. Field `dict[pointer].index` contains "unused", since location 5 is still empty, so string `ab` is not in the dictionary. It is added by executing

```
dict[pointer].parent:=I;
dict[pointer].index:=pointer;
dict[pointer].symbol:=J;
```

with `pointer`=5. `J` is moved into `I`, so `I` = 2.

/	/	/	/	1
1	2	-	-	5
a	b			b

...

Step 3: The third symbol `a` is input into `J`, so `J` = 1. The encoder has to search for string `ba` in the dictionary. It executes `pointer:=hash(I,J)`. Let's assume that the result is 8. Field `dict[pointer].index` contains "unused", so string `ba` is not in the dictionary. It is added as before by executing

```
dict[pointer].parent:=I;
dict[pointer].index:=pointer;
dict[pointer].symbol:=J;
```

with `pointer`=8. `J` is moved into `I`, so `I` = 1.

/	/	/	/	1	/	/	2	/
1	2	-	-	5	-	-	8	-
a	b			b			a	

...

Step 4: The fourth symbol `b` is input into `J`, so `J`=2. The encoder has to search for string `ab` in the dictionary. It executes `pointer:=hash(I,J)`. We know from step 2 that the result is 5. Field `dict[pointer].index` contains 5, so string `ab` is in the dictionary. The value of `pointer` is moved into `I`, so `I` = 5.
Step 5: The fifth symbol `a` is input into `J`, so `J` = 1. The encoder has to search for string `aba` in the dictionary. It executes as usual `pointer:=hash(I,J)`. Let's assume that the result is 8 (a collision). Field `dict[pointer].index` contains 8, which looks good, but field `dict[pointer].parent` contains 2 instead of the expected 5, so the hash function knows that this is a collision and string `aba` is not in dictionary entry 8. It increments `pointer` as many times as necessary until it finds a dictionary entry with `index`=8 and `parent`=5 or until it finds an unused entry. In the former case, string `aba` is in the dictionary, and `pointer` is moved to `I`. In the latter case `aba` is not in the dictionary, and the encoder saves it in the entry pointed at by `pointer`, and moves `J` to `I`.

/	/	/	/	1	/	/	2	5	/
1	2	-	-	5	-	-	8	8	-
a	b			b			a	a	

...

Example: The 15 hashing steps for encoding the string `alf␣eats␣alfalfa` are

shown below. The encoding process itself is illustrated in detail in the answer to Exercise 6.10. The results of the hashing are arbitrary; they are not the results produced by a real hash function. The 12 trie nodes constructed for this string are shown in Figure 6.26.

1. Hash(l,97) → 278. Array location 278 is set to (97, 278, l).
2. Hash(f,108) → 266. Array location 266 is set to (108, 266, f).
3. Hash(␣,102) → 269. Array location 269 is set to (102,269,␣).
4. Hash(e,32) → 267. Array location 267 is set to (32, 267, e).
5. Hash(a,101) → 265. Array location 265 is set to (101, 265, a).
6. Hash(t,97) → 272. Array location 272 is set to (97, 272, t).
7. Hash(s,116) → 265. A collision! Skip to the next available location, 268, and set it to (116, 265, s). This is why the index needs to be stored.
8. Hash(␣,115) → 270. Array location 270 is set to (115, 270, ␣).
9. Hash(a,32) → 268. A collision! Skip to the next available location, 271, and set it to (32, 268, a).
10. Hash(l,97) → 278. Array location 278 already contains index 278 and symbol l from step 1, so there is no need to store anything else or to add a new trie entry.
11. Hash(f,278) → 276. Array location 276 is set to (278, 276, f).
12. Hash(a,102) → 274. Array location 274 is set to (102, 274, a).
13. Hash(l,97) → 278. Array location 278 already contains index 278 and symbol l from step 1, so there is no need to do anything.
14. Hash(f,278) → 276. Array location 276 already contains index 276 and symbol f from step 11, so there is no need to do anything.
15. Hash(a,276) → 274. A collision! Skip to the next available location, 275, and set it to (276, 274, a).

Readers who have carefully followed the discussion up to this point will be happy to learn that the LZW decoder's use of the dictionary tree-array is simple and no hashing is needed. The decoder starts, like the encoder, by initializing the first 256 array locations. It then reads pointers from its input stream and uses each to locate a symbol in the dictionary.

In the first decoding step, the decoder inputs the first pointer and uses it to retrieve a dictionary item I. This is a symbol that is now written by the decoder on its output stream. String Ix needs to be saved in the dictionary, but symbol x is still unknown; it will be the first symbol in the next string retrieved from the dictionary.

In each decoding step after the first, the decoder inputs the next pointer and uses it to retrieve the next string J from the dictionary and write it on the output stream. If the pointer is, say 8, the decoder examines field `dict[8].index`. If this field equals 8, then this is the right node. Otherwise, the decoder examines consecutive array locations until it finds the right one.

Once the right tree node is found, the `parent` field is used to go back up the tree and retrieve the individual symbols of the string *in reverse order*. The symbols are then placed in J in the right order (see below), the decoder isolates the first symbol x of J, and saves string Ix in the next available array location. (String I was found in the previous step, so only one node, with symbol x, needs be added.) The decoder then moves J to I and is ready for the next step.

265	266	267	268	269	270	271	272	273	274	275	276	277	278
/	/	/	/	/	/	/	/	/	/	/	/	/	97
-	-	-	-	-	-	-	-	-	-	-	-	-	278
													l
/	108	/	/	/	/	/	/	/	/	/	/	/	97
-	266	-	-	-	-	-	-	-	-	-	-	-	278
	f												l
/	108	/	/	102	/	/	/	/	/	/	/	/	97
-	266	-	-	269	-	-	-	-	-	-	-	-	278
	f			␣									l
/	108	32	/	102	/	/	/	/	/	/	/	/	97
-	266	267	-	269	-	-	-	-	-	-	-	-	278
	f	e		␣									l
101	108	32	/	102	/	/	/	/	/	/	/	/	97
265	266	267	-	269	-	-	-	-	-	-	-	-	278
a	f	e		␣									l
101	108	32	/	102	/	/	97	/	/	/	/	/	97
265	266	267	-	269	-	-	272	-	-	-	-	-	278
a	f	e		␣			t						l
101	108	32	116	102	/	/	97	/	/	/	/	/	97
265	266	267	265	269	-	-	272	-	-	-	-	-	278
a	f	e	s	␣			t						l
101	108	32	116	102	115	/	97	/	/	/	/	/	97
265	266	267	265	269	270	-	272	-	-	-	-	-	278
a	f	e	s	␣	␣		t						l
101	108	32	116	102	115	32	97	/	/	/	/	/	97
265	266	267	265	269	270	268	272	-	-	-	-	-	278
a	f	e	s	␣	␣	a	t						l
101	108	32	116	102	115	32	97	/	/	/	278	/	97
265	266	267	265	269	270	268	272	-	-	-	276	-	278
a	f	e	s	␣	␣	a	t				f		l
101	108	32	116	102	115	32	97	/	102	/	278	/	97
265	266	267	265	269	270	268	272	-	274	-	276	-	278
a	f	e	s	␣	␣	a	t		a		f		l
101	108	32	116	102	115	32	97	/	102	276	278	/	97
265	266	267	265	269	270	268	272	-	274	274	276	-	278
a	f	e	s	␣	␣	a	t		a	a	f		l

Figure 6.26: Growing An LZW Trie for "alf eats alfalfa".

Retrieving a complete string from the LZW tree therefore involves following the pointers in the `parent` fields. This is equivalent to moving *up* the tree, which is why the hash function is no longer needed.

Example: The previous example describes the 15 hashing steps in the encoding of string `alf␣eats␣alfalfa`. The last step sets array location 275 to (276,274,a) and writes 275 (a pointer to location 275) on the compressed stream. When this stream is read by the decoder, pointer 275 is the last item input and processed by the decoder. The decoder finds symbol `a` in the `symbol` field of location 275 (indicating that the string stored at 275 ends with an `a`) and a pointer to location 276 in the `parent` field. The decoder then examines location 276 where it finds symbol `f` and parent pointer 278. In location 278 the decoder finds symbol `l` and a pointer to 97. Finally, in location 97 the decoder finds symbol `a` and a null pointer. The (reversed) string is therefore `afla`. There is no need for the decoder to do any hashing or to use the `index` fields.

The last point to discuss is string reversal. Two commonly-used approaches are outlined here:

1. Use a stack. A stack is a common data structure in modern computers. It is an array in memory that is accessed at one end only. At any time, the item that was last pushed into the stack will be the first one to be popped out (last-in-first-out, or LIFO). Symbols retrieved from the dictionary are pushed into the stack. When the last one has been retrieved and pushed, the stack is popped, symbol by symbol, into variable `J`. When the stack is empty, the entire string has been reversed. This is a common way to reverse a string.
2. Retrieve symbols from the dictionary and concatenate them into `J` *from right to left*. When done, the string will be stored in `J` in the right order. Variable `J` must be long enough to accommodate the longest possible string, but then it has to be long enough even when a stack is used.

◇ **Exercise 6.14:** What is the longest string that can be retrieved from the LZW dictionary during decoding?

(A reminder. The troublesome issue of software patents and licenses is treated in Section 6.34.)

6.13.3 LZW in Practice

The publication of the LZW algorithm, in 1984, has strongly affected the data compression community and has influenced many people to come up with implementations and variants of this method. Some of the most important LZW variants and spin-offs are described here.

6.13.4 Differencing

The idea of differencing, or relative encoding, has already been mentioned in Section 1.3.1. This idea turns out to be useful in LZW image compression, since most adjacent pixels don't differ by much. It is possible to implement an LZW encoder that computes the value of a pixel relative to its predecessor and then encodes this difference. The decoder should, of course, be compatible and should compute the absolute value of a pixel after decoding its relative value.

6.13.5 LZW Variants

A word-based LZW variant is described in Section 11.6.2.

LZW is an adaptive data compression method, but it is slow to adapt to its input, since strings in the dictionary become only one character longer at a time. Exercise 6.11 shows that a string of one million `a`'s (which, of course, is highly redundant) produces dictionary phrases, the longest of which contains only 1,414 `a`'s.

The LZMW method, Section 6.15, is a variant of LZW that overcomes this problem. Its main principle is this: Instead of adding `I` plus one character of the next phrase to the dictionary, add `I` plus the entire next phrase to the dictionary.

The LZAP method, Section 6.16, is yet another variant based on this idea: Instead of just concatenating the last two phrases and placing the result in the dictionary, place all prefixes of the concatenation in the dictionary. More specifically, if `S` and `T` are the last two matches, add `St` to the dictionary for every nonempty prefix `t` of `T`, including `T` itself.

Table 6.27 summarizes the principles of LZW, LZMW, and LZAP and shows how they naturally suggest another variant, LZY.

Increment string by	Add a string to the dictionary: per phrase	per input char.
One character:	LZW	LZY
Several chars:	LZMW	LZAP

Table 6.27: LZW and Three Variants.

LZW adds one dictionary string per phrase and increments strings by one symbol at a time. LZMW adds one dictionary string per phrase and increments strings by several symbols at a time. LZAP adds one dictionary string per input symbol and increments strings by several symbols at a time. LZY, Section 6.18, fits the fourth cell of Table 6.27. It is a method that adds one dictionary string per input symbol and increments strings by one symbol at a time.

6.14 UNIX Compression (LZC)

In the vast UNIX world, `compress` used to be the most common compression utility (although GNU `gzip` has become more popular because it is free from patent claims, is faster, and provides superior compression). This utility (also called LZC) uses LZW with a growing dictionary. It starts with a small dictionary of just $2^9 = 512$ entries (with the first 256 of them already filled up). While this dictionary is being used, 9-bit pointers are written onto the output stream. When the original dictionary fills up, its size is doubled, to 1024 entries, and 10-bit pointers are used from this point. This process continues until the pointer size reaches a maximum set by the user (it can be set to between 9 and 16 bits, with 16 as the default value). When the largest allowed dictionary fills up, the program continues without changing the dictionary (which then becomes static), but with monitoring the compression ratio. If the ratio falls below a

predefined threshold, the dictionary is deleted, and a new 512-entry dictionary is started. This way, the dictionary never gets "too out of date."

Decoding is done by the `uncompress` command, which implements the LZC decoder. Its main task is to maintain the dictionary in the same way as the encoder.

Two improvements to LZC, proposed by [Horspool 91], are listed below:

1. Encode the dictionary pointers with the phased-in binary codes of Section 5.3.1. Thus if the dictionary size is $2^9 = 512$ entries, pointers can be encoded in either 8 or 9 bits.
2. Determine all the impossible strings at any point. Suppose that the current string in the look-ahead buffer is `abcd...` and the dictionary contains strings `abc` and `abca` but not `abcd`. The encoder will output, in this case, the pointer to `abc` and will start encoding a new string starting with `d`. The point is that after decoding `abc`, the decoder knows that the next string cannot start with an `a` (if it did, an `abca` would have been encoded, instead of `abc`). In general, if `S` is the current string, then the next string cannot start with any symbol `x` that satisfies "`Sx` is in the dictionary."

This knowledge can be used by both the encoder and decoder to reduce redundancy even further. When a pointer to a string should be output, it should be coded, and the method of assigning the code should eliminate all the strings that are known to be impossible at that point. This may result in a somewhat shorter code but is probably too complex to justify its use in practice.

6.14.1 LZT

LZT is an extension, proposed by [Tischer 87], of UNIX compress/LZC. The major innovation of LZT is the way it handles a full dictionary. Recall that LZC treats a full dictionary in a simple way. It continues without updating the dictionary until the compression ratio drops below a preset threshold, and then it deletes the dictionary and starts afresh with a new dictionary. In contrast, LZT maintains, in addition to the dictionary (which is structured as a hash table, as described in Section 6.13.2), a linked list of phrases sorted by the number of times a phrase is used. When the dictionary fills up, the LZT algorithm identifies the least-recently-used (LRU) phrase in the list and deletes it from both the dictionary and the list.

Maintaining the list requires several operations per phrase, which is why LZT is slower than LZW or LZC, but the fact that the dictionary is not completely deleted results in better compression performance (each time LZC deletes its dictionary, its compression performance drops significantly for a while).

There is some similarity between the simple LZ77 algorithm and the LZT encoder. In LZ77, the search buffer is shifted, which deletes the leftmost phrase regardless of how useful it is (i.e., how often it has appeared in the input). In contrast, the LRU approach of LZT deletes the least useful phrase, the one that has been used the least since its insertion into the dictionary. In both cases, we can visualize a window of phrases. In LZ77 the window is sorted by the order in which phrases have been input, while in LZT the window is sorted by the usefulness of the phrases. Thus, LZT adapts better to varying statistics in the input data.

Another innovation of LZT is its use of phased-in codes (Section 2.9). It starts with a dictionary of 512 entries, of which locations 0 through 255 are reserved for the 256 8-bit bytes (those are never deleted) and code 256 is reserved as an escape code. These 257 codes are encoded by 255 8-bit codes and two 9-bit codes. Once the first

input phrase has been identified and added to the dictionary, it (the dictionary) has 258 codes and they are encoded by 254 8-bit codes and four 9-bit codes. The use of phased-in codes is especially useful at the beginning, when the dictionary contains only a few useful phrases. As encoding progresses and the dictionary fills up, these codes contribute less and less to the overall performance of LZT.

The LZT dictionary is maintained as a hash table where each entry contains a phrase. Recall that in LZW (and also in LZC), if a phrase sI is in the dictionary (where s is a string and I is a single character), then s is also in the dictionary. This is why each dictionary entry in LZT has three fields (1) an output code (the code that has been assigned to a string s), (2) the (ASCII) code of an extension character I, and (3) a pointer to the list. When an entry is deleted, it is marked as free and the hash table is reorganized according to algorithm R in volume 3, Section 6.4 of [Knuth 73].

Another difference between LZC and LZT is the hash function. In LZC, the hash function takes an output code, shifts it one position to the left, and exclusive-ors it with the bits of the extension character. The hash function of LZT, on the other hand, reverses the bits of the output code before shifting and exclusive-oring it, which leads to fewer collisions and an almost perfect hash performance.

6.15 LZMW

This LZW variant, developed by V. Miller and M. Wegman [Miller and Wegman 85], is based on two principles:

1. When the dictionary becomes full, the least-recently-used dictionary phrase is deleted. There are several ways to select this phrase, and the developers suggest that any reasonable way of doing so would work. One possibility is to identify all the dictionary phrases S for which there are no phrases Sa (nothing has been appended to S, implying that S hasn't been used since it was placed in the dictionary) and delete the oldest of them. An auxiliary data structure has to be constructed and maintained in this case, pointing to dictionary phrases according to their age (the first pointer always points to the oldest phrase). The first 256 dictionary phrases should never be deleted.
2. Each phrase added to the dictionary is a concatenation of two strings, the previous match (S' below) and the current one (S). This is in contrast to LZW, where each phrase added is the concatenation of the current match and the first symbol of the next match. The pseudo-code algorithm illustrates this:

```
Initialize Dict to all the symbols of alphabet A;
i:=1; S':=null;
while i <= input size
 k:=longest match of Input[i] to Dict;
 Output(k);
 S:=Phrase k of Dict;
 i:=i+length(S);
 If phrase S'S is not in Dict, append it to Dict;
 S':=S;
endwhile;
```

By adding the concatenation S'S to the LZMW dictionary, dictionary phrases can grow by more than one symbol at a time. This means that LZMW dictionary phrases are more "natural" units of the input (for example, if the input is text in a natural language, dictionary phrases will tend to be complete words or even several words in that language). This, in turn, implies that the LZMW dictionary generally adapts to the input faster than the LZW dictionary.

Table 6.28 illustrates the LZMW method by applying it to the string

sir␣sid␣eastman␣easily␣teases␣sea␣sick␣seals.

LZMW adapts to its input faster than LZW but has the following three disadvantages:

1. The dictionary data structure cannot be the simple LZW trie, because not every prefix of a dictionary phrase is included in the dictionary. This means that the one-symbol-at-a-time search method used in LZW will not work. Instead, when a phrase S is added to the LZMW dictionary, every prefix of S must be added to the data structure, and every node in the data structure must have a tag indicating whether the node is in the dictionary or not.
2. Finding the longest string may require backtracking. If the dictionary contains aaaa and aaaaaaaa, we have to reach the eighth symbol of phrase aaaaaaab to realize that we have to choose the shorter phrase. This implies that dictionary searches in LZMW are slower than in LZW. This problem does not apply to the LZMW decoder.
3. A phrase may be added to the dictionary twice. This again complicates the choice of data structure for the dictionary.

◇ **Exercise 6.15:** Use the LZMW method to compress the string swiss miss.

◇ **Exercise 6.16:** Compress the string yabbadabbadabbadoo using LZMW.

6.16 LZAP

LZAP is an extension of LZMW. The "AP" stands for "All Prefixes" [Storer 88]. LZAP adapts to its input fast, like LZMW, but eliminates the need for backtracking, a feature that makes it faster than LZMW. The principle is this: Instead of adding the concatenation S'S of the last two phrases to the dictionary, add all the strings S't where t is a prefix of S (including S itself). Thus if $S' = a$ and $S = bcd$, add phrases ab,abc, and abcd to the LZAP dictionary. Table 6.29 shows the matches and the phrases added to the dictionary for yabbadabbadabbadoo.

In step 7 the encoder concatenates d to the two prefixes of ab and adds the two phrases da and dab to the dictionary. In step 9 it concatenates ba to the three prefixes of dab and adds the resulting three phrases bad, bada, and badab to the dictionary.

LZAP adds more phrases to its dictionary than does LZMW, so it takes more bits to represent the position of a phrase. At the same time, LZAP provides a bigger selection of dictionary phrases as matches for the input string, so it ends up compressing slightly better than LZMW while being faster (because of the simpler dictionary data structure, which eliminates the need for backtracking). This kind of trade-off is common in computer algorithms.

Step	Input	Out-put	S	Add to dict.	S'
	sir sid eastman easily teases sea sick seals				
1	s	115	s	—	—
2	i	105	i	256-si	s
3	r	114	r	257-ir	i
4	-	32	␣	258-r␣	r
5	si	256	si	259-␣si	␣
6	d	100	d	260-sid	si
7	-	32	␣	261-d␣	d
8	e	101	e	262-␣e	␣
9	a	97	a	263-ea	e
10	s	115	s	264-as	a
11	t	117	t	265-st	s
12	m	109	m	266-tm	t
13	a	97	a	267-ma	m
14	n	110	n	268-an	a
15	-e	262	␣e	269-n␣e	n
16	as	264	as	270-␣eas	␣e
17	i	105	i	271-asi	as
18	l	108	l	272-il	i
19	y	121	y	273-ly	l
20	-	32	␣	274-y␣	y
21	t	117	t	275-␣t	␣
22	ea	263	ea	276-tea	t
23	s	115	s	277-eas	ea
24	e	101	e	278-se	s
25	s	115	s	279-es	e
26	-	32	␣	280-s␣	s
27	se	278	se	281-␣se	␣
28	a	97	a	282-sea	se
29	-si	259	␣si	283-a␣si	a
30	c	99	c	284-␣sic	␣si
31	k	107	k	285-ck	c
32	-se	281	␣se	286-k␣se	k
33	a	97	a	287-␣sea	␣se
34	l	108	l	288-al	a
35	s	115	s	289-ls	l

Table 6.28: LZMW Example.

Step	Input	Match	Add to dictionary
	yabbadabbadabbadoo		
1	y	y	—
2	a	a	256-ya
3	b	b	257-ab
4	b	b	258-bb
5	a	a	259-ba
6	d	d	260-ad
7	ab	ab	261-da, 262-dab
8	ba	ba	263-abb, 264-abba
9	dab	dab	265-bad, 266-bada, 267-badab
10	bad	bad	268-dabb, 269-dabba, 270-dabbad
11	o	o	271-bado
12	o	o	272-oo

Table 6.29: LZAP Example.

6.17 LZJ

LZJ is an interesting LZ variant, originated by [Jakobsson 85]. It stores in its dictionary, which can be viewed either as a multiway tree or as a forest, *every* phrase found in the input. If a phrase is found n times in the input, only one copy is stored in the dictionary. Such behavior tends to fill the dictionary up very quickly, so LZJ limits the length of phrases to a preset parameter h. The developer of the method recommends h values of around 6. Thus, if $h = 4$, the alphabet includes the letters and the blank space, and the input is `Once␣upon␣a␣time...`, then the dictionary will start with all the symbols of the alphabet and the following strings will gradually be inserted in it: `on`, `onc`, `nc`, `ce`, `nce`, `Once`, `nce␣`, `ce␣u`, `e␣up`, `␣upo`, and so on.

(The value of h is critical. Small values of h cause the LZJ encoder to store only short phrases in the dictionary. This degrades the compression because every code in the compressed file will correspond to only a short string. Large values of h also degrade compression because only a few long strings occur often in the input.)

An indirect result of this principle and of the way LZJ operates is that LZJ encoding is simpler than its decoding. Thus, LZJ is an asymmetric compression method and is unusual in this respect because in other asymmetric methods decoding is simpler and faster than encoding.

Example. Given the alphabet `a`, `b`, and `c`, the value $h = 4$, and the input data `babbaaacba`, the left side of Figure 6.30 lists the 24 phrases whose length is h or shorter, together with their dictionary addresses. The right side shows the complete 3-way dictionary tree (as a forest of three trees) with all 24 phrases. Each node in the dictionary has four fields, the character itself, a pointer (shown as a solid arrow) to its leftmost child, a pointer (horizontal, in dashed) to its right sibling, if any, and a frequency count that is discussed below. If a node is a rightmost child, the sibling pointer is replaced by a pointer to the parent (dashed, pointing up).

The dictionary's size H is another parameter of LZJ and is preset in each implementation (in the mid 1980s, the developer recommended $H \approx 2^{13}$, but current imple-

a	0	bb	9	aac	17
b	1	bba	10	aacb	18
c	2	bbba	11	ac	19
ba	3	baa	12	acb	20
bab	4	baaa	13	acba	21
babb	5	aa	14	cb	22
ab	6	aaa	15	cba	23
abb	7	aaac	16		
abba	8				

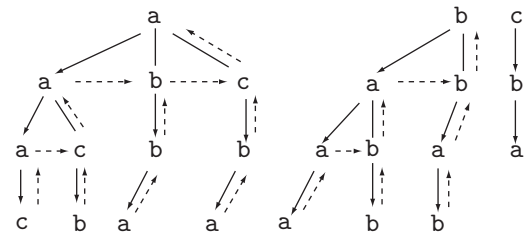


Figure 6.30: An LZJ Dictionary.

mentations can afford much larger values). Each input phrase found by the encoder in the dictionary is compressed by writing the dictionary address of the last character of the phrase on the output. This address is a number between 0 and $H-1$, written in $\lceil \log_2 H \rceil$ bits.

In order to understand the operation of the encoder, we assume $h = 4$ and the input string `Once␣upon␣a␣time...`.

The encoder initializes the dictionary to the characters of the alphabet. At a general point in the encoding it reads the next input string, say `pon␣a␣time...`, selects the dictionary tree for `p`, and tries to match as many characters as possible. If `pon` is found in the tree, but `pon␣` isn't, the encoder outputs the dictionary address of the `n` of `pon` and updates the dictionary by inserting in it the strings `e␣up`, `␣upo`, and `upon`. They are placed in the trees whose roots are `e`, `␣`, and `u`, respectively. As each string is inserted, the encoder increments the counts of all the nodes that are on the path from the root to the leaf node of the string.

When the dictionary fills up, the encoder prunes the trees by deleting any nodes with a count of 1 (where instead of 1, an implementation can opt for a user-defined parameter). The roots of the trees correspond to the characters of the alphabet and should never be removed. The decoder can mimic this operation in lockstep with the encoder.

Pruning the trees results in free nodes, but consecutive prunings result in fewer and fewer free nodes, as illustrated by Figure 6.31. The figure shows that with $H = 1{,}024$, many dictionary prunings are needed for the first 4,000 input characters. The developer of the method also shows that doubling the dictionary size to 2,048 nodes results in only three dictionary prunings for the same input data. Eventually, pruning becomes ineffective and LZJ continues (nonadaptively) without it. This behavior suggests that a large dictionary may not require any prunings at all and may therefore speed up LZJ (both encoding and decoding) considerably.

LZJ decoding is more time consuming than its encoding, because the decoder has to follow various pointers in a dictionary tree. This is best illustrated by an example (refer to Figure 6.30). Suppose that the decoder reads 17 from the compressed stream. Location 17 contains the character `c`, the last character in the phrase being decoded. The decoder has to follow the back pointers (the dashed pointers in the figure) until it arrives at the root of the tree. It then follows the forward pointers (both dashed horizontal and solid), collecting characters as it goes along, until it arrives back at location 17. This is

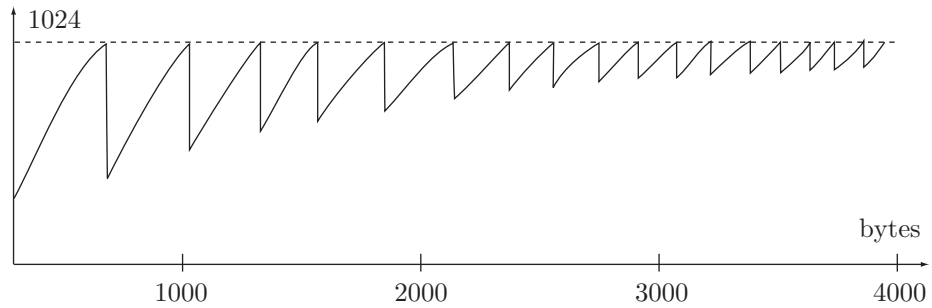


Figure 6.31: Effect of Pruning the LZJ Dictionary.

how the decoder constructs phrase `aac`. Once the phrase is decoded, the decoder uses it to update the dictionary by inserting phrases in lockstep with the encoder.

The last interesting feature of LZJ is the format of the compressed stream. If the dictionary size H is fixed, then each item in the compressed stream is a fixed-size, $\lceil \log_2 H \rceil$ bits pointer to the dictionary. In principle, however, the dictionary can be allowed to grow as needed and the algorithm may use one of the many variable-length codes for the integers to encode the dictionary pointers.

It turns out that this approach, which initially seems promising, is in fact worse than having fixed-length pointers. The reason is that each variable-length code is ideal for a certain distribution of the integers in the data, but in the case of LZJ nothing is known about this distribution. Initially, the dictionary is small, so all the pointers to it are small integers. When the dictionary grows, especially after it has been pruned, pointers to it may be small or large with the same probability. Given a set of n integers with a uniform distribution, the length of a fixed-length code for them is $\lceil \log_2 n \rceil$, but the average length of any variable-length code is generally longer (see Figure 3.11).

A good compromise is the use of phased-in codes (Section 2.9). If $H = 1{,}000$, the length of a fixed-size code is 10 bits, but only 1,000 out of the 1,024 possible 10-bit combinations are used. With phased-in codes, most codewords would be 10 bits long, but some codewords would be only nine bits long. The excellent book *Text Compression* [Bell et al. 90], refers to this compromise (on page 230) as LZJ′.

> "Heavens, Andrew!" said his wife; "what is a rustler?"
> It was not in any dictionary, and current translations of it were inconsistent. A man at Hoosic Falls said that he had passed through Cheyenne, and heard the term applied in a complimentary way to people who were alive and pushing. Another man had always supposed it meant some kind of horse. But the most alarming version of all was that a rustler was a cattle thief.
>
> —Owen Wister, *The Virginian—A Horseman of the Plains*

6.18 LZY

The LZY method is due to Dan Bernstein. The Y stands for Yabba, which came from the input string originally used to test the algorithm. The LZY dictionary is initialized to all the single symbols of the alphabet. For every symbol `C` in the input stream, the decoder looks for the longest string `P` that precedes `C` and is already included in the dictionary. If the string `PC` is not in the dictionary, it is added to it as a new phrase.

As an example, the input `yabbadabbadabbadoo` causes the phrases `ya`, `ab`, `bb`, `ba`, `ad`, `da`, `abb`, `bba`, `ada`, `dab`, `abba`, `bbad`, `bado`, `ado`, and `oo` to be added to the dictionary.

While encoding the input, the encoder keeps track of the list of matches-so-far `L`. Initially, `L` is empty. If `C` is the current input symbol, the encoder (before adding anything to the dictionary) checks, for every string `M` in `L`, whether string `MC` is in the dictionary. If it is, then `MC` becomes a new match-so-far and is added to `L`. Otherwise, the encoder outputs the number of `L` (its position in the dictionary) and adds `C`, as a new match-so-far, to `L`.

Here is a pseudo-code algorithm for constructing the LZY dictionary. The authors' personal experience suggests that implementing such an algorithm in a real programming language results in a deeper understanding of its operation.

```
Start with a dictionary containing all the symbols of the
 alphabet, each mapped to a unique integer.
M:=empty string.
Repeat
 Append the next symbol C of the input stream to M.
 If M is not in the dictionary, add it to the dictionary,
  delete the first character of M, and repeat this step.
Until end-of-input.
```

The output of LZY is not synchronized with the dictionary additions. Also, the encoder must be careful not to have the longest output match overlap itself. Because of this, the dictionary should consist of two parts, S and T, where only the former is used for the output. The algorithm is the following:

```
Start with S mapping each single character to a unique integer;
set T empty; M empty; and O empty.
Repeat
 Input the next symbol C. If OC is in S, set O:=OC;
 otherwise output S(O), set O:=C, add T to S,
  and remove everything from T.
 While MC is not in S or T, add MC to T (mapping to the next
   available integer), and chop off the first character of M.
 After M is short enough so that MC is in the dict., set M:=MC.
Until end-of-input.
Output S(O) and quit.
```

The decoder reads the compressed stream. It uses each code to find a phrase in the dictionary, it outputs the phrase as a string, then uses each symbol of the string to add a new phrase to the dictionary in the same way the encoder does. Here are the decoding steps:

```
Start with a dictionary containing all the symbols of the
 alphabet, each mapped to a unique integer.
M:=empty string.
Repeat
 Read D(O) from the input and take the inverse under D to find O.
 As long as O is not the empty string, find the first character C
  of O, and update (D,M) as above.
  Also output C and chop it off from the front of O.
Until end-of-input.
```

Notice that encoding requires two fast operations on strings in the dictionary: (1) testing whether string SC is in the dictionary if S's position is known and (2) finding S's position given CS's position. Decoding requires the same operations plus fast searching to find the first character of a string when its position in the dictionary is given.

Table 6.32 illustrates LZY for the input string abcabcabcabcabcabcx'. It shows the phrases added to the dictionary at each step, as well as the list of current matches.

The encoder starts with no matches. When it inputs a symbol, it appends it to each match-so-far; any results that are already in the dictionary become the new matches-so-far (the symbol itself becomes another match). Any results that are not in the dictionary are deleted from the list and added to the dictionary.

Before reading the fifth c, for example, the matches-so-far are bcab, cab, ab, and b. The encoder appends c to each match. bcabc doesn't match, so the encoder adds it to the dictionary. The rest are still in the dictionary, so the new list of matches-so-far is cabc, abc, bc, and c.

When the x is input, the current list of matches-so-far is abcabc, bcabc, cabc, abc, bc, and c. None of abcabcx, bcabcx, cabcx, abcx, bcx, or cx are in the dictionary, so they are all added to it, and the list of matches-so-far is reduced to just a single x.

Airman stops coed
Anagram of "data compression"

6.19 LZP

LZP is an LZ77 variant developed by Charles Bloom [Bloom 96] (the P stands for "prediction"). It is based on the principle of context prediction, which says, "if a certain string abcde has appeared in the input stream in the past and was followed by fg..., then when abcde appears again in the input stream, there is a good chance that it will be followed by the same fg...." Section 6.35 should be consulted for the relation between dictionary-based and prediction algorithms.

Figure 6.33 (part I) shows an LZ77 sliding buffer with fgh... as the current symbols (this string is denoted by S) in the look-ahead buffer, immediately preceded by abcde in the search buffer. The string abcde is called the *context* of fgh... and is denoted by C. In general, the context of a string S is the N-symbol string C immediately to the left of S. A context can be of any length N, and variants of LZP, discussed in Sections 6.19.3 and 6.19.4, use different values of N. The algorithm passes the context through a hash

Step	Input	Add to dict.	Current matches
	abcabcabcabcabcabcabcx		
1	a	—	a
2	b	256-ab	b
3	c	257-bc	c
4	a	258-ca	a
5	b	—	ab, b
6	c	259-abc	bc, c
7	a	260-bca	ca, a
8	b	261-cab	ab, b
9	c	—	abc, bc, c
10	a	262-abca	bca, ca, a
11	b	263-bcab	cab, ab, b
12	c	264-cabc	abc, bc, c
13	a	—	abca, bca, ca, a
14	b	265-abcab	bcab, cab, ab, b
15	c	266-bcabc	cabc, abc, bc, c
16	a	267-cabca	abca, bca, ca, a
17	b	—	abcab, bcab, cab, ab, b
18	c	268-abcabc	bcabc, cabc, abc, bc, c
19	a	269-bcabca	cabca, abca, bca, ca, a
20	b	270-cabcab	abcab, bcab, cab, ab, b
21	c	—	abcabc, bcabc, cabc, abc, bc, c
22	x	271-abcabcx	x
23		272-bcabcx	
24		273-cabcx	
25		274-abcx	
26		275-bcx	
27		276-cx	

Table 6.32: LZY Example.

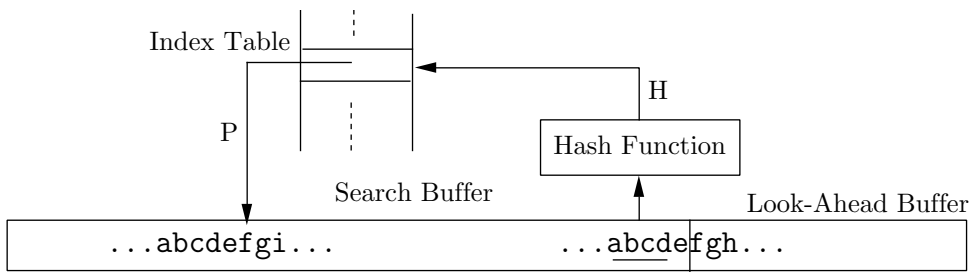


Figure 6.33: The Principle of LZP: Part I.

function and uses the result H as an index to a table of pointers called the *index table*. The index table contains pointers to various symbols in the search buffer. Index H is

used to select a pointer P. In a typical case, P points to a previously seen string whose context is also abcde (see below for atypical cases). The algorithm then performs the following steps:

Step 1: It saves P and replaces it in the index table with a fresh pointer Q pointing to fgh... in the look-ahead buffer (Figure 6.33 Part II). An integer variable L is set to zero. It is used later to indicate the match length.

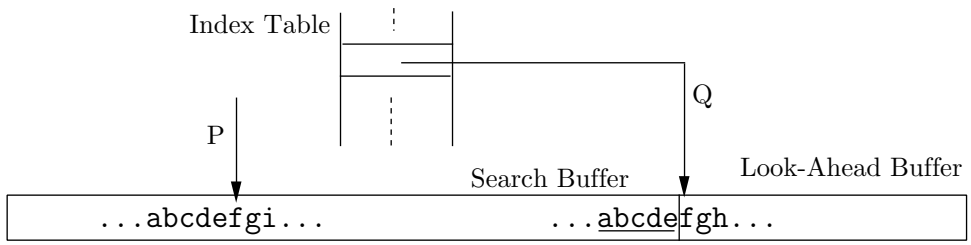


Figure 6.33: The Principle of LZP: Part II.

Step 2: If P is not a null pointer, the algorithm follows it and compares the string pointed at by P (string fgi... in the search buffer) to the string "fgh..." in the look-ahead buffer. Only two symbols match in our example, so the match length, L, is set to 2.
Step 3: If $L = 0$ (no symbols have been matched), the buffer is slid to the right (or, equivalently, the input is shifted to the left) **one position** and the first symbol of string S (the f) is written on the compressed stream as a raw ASCII code (a literal).
Step 4: If $L > 0$ (L symbols have been matched), the buffer is slid to the right L positions and the value of L is written on the compressed stream (after being suitably encoded).

In our example the single encoded value $L = 2$ is written on the compressed stream instead of the two symbols fg, and it is this step that produces compression. Clearly, the larger the value of L, the better the compression. Large values of L result when an N-symbol context C in the input stream is followed by the same long string S as a previous occurrence of C. This may happen when the input stream features high redundancy. In a random input stream each occurrence of the same context C is likely to be followed by another S, leading to $L = 0$ and therefore to no compression. An "average" input stream results in more literals than L values being written on the output stream (see also Exercise 6.18).

The decoder inputs the compressed stream item by item and creates the decompressed output in a buffer B. The steps are:

Step 1: Input the next item I from the compressed stream.
Step 2: If I is a raw ASCII code (a literal), it is appended to buffer B, and the data in B is shifted to the left one position.
Step 3: If I is an encoded match length, it is decoded, to obtain L. The present context C (the rightmost N symbols in B) is hashed to an index H, which is used to select a pointer P from the index table. The decoder copies the string of L symbols starting at B[P] and appends it to the right end of B. It also shifts the data in B to the left L positions and replaces P in the index table with a fresh pointer, to keep in lockstep with the encoder.

Two points remain to be discussed before we are ready to look at a detailed example.

1. When the encoder starts, it places the first N symbols of the input stream in the search buffer, to become the first context. It then writes these symbols, as literals, on the compressed stream. This is the only special step needed to start the compression. The decoder starts by reading the first N items off the compressed stream (they should be literals), and placing them at the rightmost end of buffer `B`, to serve as the first context.

2. It has been mentioned before that in the typical case, `P` points to a previously-seen string whose context is identical to the present context `C`. In an atypical case, `P` may be pointing to a string whose context is different. The algorithm, however, does not check the context and always behaves in the same way. It simply tries to match as many symbols as possible. At worst, zero symbols will match, leading to one literal written on the compressed stream.

6.19.1 Example

The input stream `xyabcabcabxy` is used to illustrate the operation of the LZP encoder. To simplify the example, we use $N = 2$.

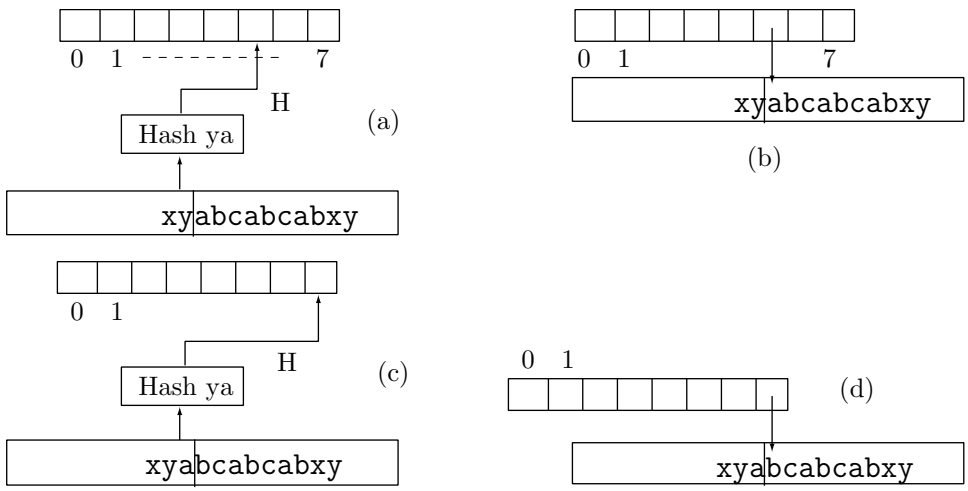


Figure 6.34: LZP Compression of xyabcabcabxy: Part I.

1. To start the operation, the encoder shifts the first two symbols `xy` to the search buffer and outputs them as literals. It also initializes all locations of the index table to the null pointer.

2. The current symbol is `a` (the first `a`), and the context is `xy`. It is hashed to, say, 5, but location 5 of the index table contains a null pointer, so `P` is null (Figure 6.34a). Location 5 is set to point to the first `a` (Figure 6.34b), which is then output as a literal. The data in the encoder's buffer is shifted to the left.

3. The current symbol is the first `b`, and the context is `ya`. It is hashed to, say, 7, but location 7 of the index table contains a null pointer, so `P` is null (Figure 6.34c). Location

7 is set to point to the first `b` (Figure 6.34d), which is then output as a literal. The data in the encoder's buffer is shifted to the left.
4. The current symbol is the first `c`, and the context is `ab`. It is hashed to, say, 2, but location 2 of the index table contains a null pointer, so `P` is null (Figure 6.34e). Location 2 is set to point to the first `c` (Figure 6.34f), which is then output as a literal. The data in the encoder's buffer is shifted to the left.
5. The same happens two more times, writing the literals `a` and `b` on the compressed stream. The current symbol is now (the second) `c`, and the context is `ab`. This context is hashed, as in step 4, to 2, so `P` points to "`cabc...`". Location 2 is set to point to the current symbol (Figure 6.34g), and the encoder tries to match strings `cabcabxy` and `cabxy`. The resulting match length is $L = 3$. The number 3 is written, encoded on the output, and the data is shifted three positions to the left.
6. The current symbol is the second `x`, and the context is `ab`. It is hashed to 2, but location 2 of the index table points to the second `c` (Figure 6.34h). Location 2 is set to point to the current symbol, and the encoder tries to match strings `cabxy` and `xy`. The resulting match length is, of course, $L = 0$, so the encoder writes `x` on the compressed stream as a literal and shifts the data one position.
7. The current symbol is the second `y`, and the context is `bx`. It is hashed to, say, 7. This is a hash collision, since context `ya` was hashed to 7 in step 3, but the algorithm does not check for collisions. It continues as usual. Location 7 of the index table points to the first `b` (or rather to the string `bcabcabxy`). It is set to point to the current symbol, and the encoder tries to match strings `bcabcabxy` and `y`, resulting in $L = 0$. The encoder writes `y` on the compressed stream as a literal and shifts the data one position.
8. The current symbol is the end-of-data, so the algorithm terminates.

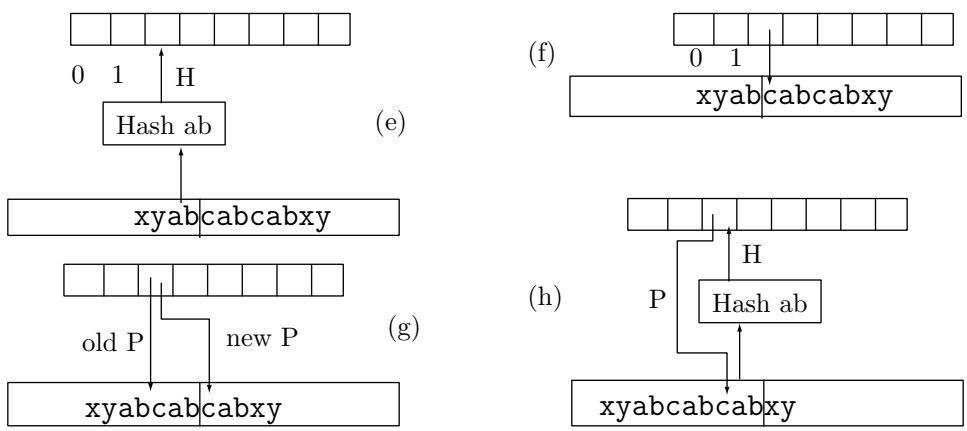


Figure 6.34 (Continued). LZP Compression of `xyabcabcabxy`: Part II.

◇ **Exercise 6.17:** Write the LZP encoding steps for the input string `xyaaaa...`.

> Structures are the weapons of the mathematician.
> —Nicholas Bourbaki

6.19.2 Practical Considerations

Shifting the data in the buffer would require updating all the pointers in the index table. An efficient implementation should therefore adopt a better solution. Two approaches are described below, but other ones may also work.

1. Reserve a buffer as large as possible, and compress the input stream in blocks. Each block is input into the buffer and is never shifted. Instead, a pointer is moved from left to right, to point at any time to the current symbol in the buffer. When the entire buffer has been encoded, the buffer is filled up again with fresh input. This approach simplifies the program but has the disadvantage that the data at the end of a block cannot be used to predict anything for the next block. Each block is encoded independently of the other ones, leading to poorer compression.

2. Reserve a large buffer, and use it as a circular queue (Section 6.3.1). The data itself does not have to be shifted, but after encoding the current symbol the data is *effectively* shifted by updating the start and end pointers, and a new symbol is input and stored in the buffer. The algorithm is somewhat more complicated, but this approach has the advantage that the entire input is encoded as one stream. Every symbol benefits from the D symbols preceding it (where D is the total length of the buffer).

Imagine a pointer P in the index table pointing to some symbol `X` in the buffer. When the movement of the two pointers in the circular queue leaves `X` outside the queue, some new symbol `Y` will be input into the position occupied by `X`, and `P` will now be pointing to `Y`. When `P` is next selected by the hashing function and is used to match strings, the match will likely result in $L = 0$. However, the algorithm always replaces the pointer that it uses, so such a case should not degrade the algorithm's performance significantly.

6.19.3 LZP1 and LZP2

There are currently four versions of LZP, called LZP1 through LZP4. This section discusses the details of the first two. The context used by LZP1 is of order 3, i.e., it is the three bytes preceding the current one. The hash function produces a 12-bit index `H` and is best described by the following C code:

```
H=((C>>11)^C)&0xFFF.
```

Since `H` is 12 bits, the index table should be $2^{12} = 4{,}096$ entries long. Each entry is two bytes (16 bits), but only 14 of the 16 bits are used. A pointer `P` selected in the index table thus points to a buffer of size $2^{14} = 16$K.

The LZP1 encoder creates a compressed stream with literals and L values mixed together. Each item must therefore be preceded by a flag indicating its nature. Since only two flags are needed, the simplest choice would be 1-bit flags. However, we have already mentioned that an "average" input stream results in more literals than L values, so it makes sense to assign a short flag (less than one bit) to indicate a literal, and a long flag (a wee bit longer than one bit) to indicate a length. The scheme used by LZP1 uses 1 to indicate two literals, 01 to indicate a literal followed by a match length, and 00 to indicate a match length.

◇ **Exercise 6.18:** Let T indicate the probability of a literal in the compressed stream. For what value of T does the above scheme produce flags that are 1-bit long on average?

A literal is written on the compressed stream as an 8-bit ASCII code. Match lengths are encoded according to Table 6.35. Initially, the codes are 2 bits. When these are all used up, 3 bits are added, for a total of 5 bits, where the first 2 bits are 1's. When these are also all used, 5 bits are added, to produce 10-bit codes where the first 5 bits are 1's. From then on another group of 8 bits is added to the code whenever all the old codes have been used up. Notice how the pattern of all 1's is never used as a code and is reserved to indicate longer and longer codes. Notice also that a unary code or a general unary code (Section 3.1) might have been a better choice.

Length	Code	Length	Code
1	00	11	11|111|00000
2	01	12	11|111|00001
3	10	⋮	
4	11|000	41	11|111|11110
5	11|001	42	11|111|11111|00000000
6	11|010	⋮	
⋮		296	11|111|11111|11111110
10	11|110	297	11|111|11111|11111111|00000000

Table 6.35: Codes Used by LZP1 and LZP2 for Match Lengths.

The compressed stream consists of a mixture of literals (bytes with ASCII codes) and control bytes containing flags and encoded lengths. This is illustrated by the output of the example of Section 6.19.1. The input of this example is the string `xyabcabcabxy`, and the output items are `x`, `y`, `a`, `b`, `c`, `a`, `b`, 3, `x`, and `y`. The actual output stream consists of the single control byte 111 01|10 1 followed by nine bytes with the ASCII codes of `x`, `y`, `a`, `b`, `c`, `a`, `b`, `x`, and `y`.

◇ **Exercise 6.19:** Explain the content of the control byte 111 01|10 1.

Another example of a compressed stream is the three literals `x`, `y`, and `a` followed by the four match lengths 12, 12, 12, and 10. We first prepare the flags

1 (`x`, `y`) 01 (`a`, 12) 00 (12) 00 (12) 00 (12) 00 (10),

then substitute the codes of 12 and 10,

$1xy01a$11|111|0000100|11|111|0000100|11|111|0000100|11|111|0000100|11|110,

and finally group together the bits that make up the control bytes. The result is

10111111 `x`, `y`, `a`, 00001001 11110000 10011111 00001001 11110000 10011110.

Notice that the first control byte is followed by the three literals.

The last point to be mentioned is the case ...0 1yyyyyyy zzzzzzzz. The first control byte ends with the 0 of a pair 01, and the second byte starts with the 1 of the same pair. This indicates a literal followed by a match length. The match length is the yyy bits (at least some of them) in the second control byte. If the code of the match length is long, then the zzz bits or some of them may be part of the code. The literal is either the zzz byte or the byte following it.

LZP2 is identical to LZP1 except that literals are coded using nonadaptive Huffman codes. Ideally, two passes should be used; the first one counting the frequency of

occurrence of the symbols in the input stream and the second pass doing the actual compression. In between the passes, the Huffman code table can be constructed.

6.19.4 LZP3 and LZP4

LZP3 is similar to both LZP1 and LZP2. It uses order-4 contexts and more sophisticated Huffman codes to encode both the match lengths and the literals. The LZP3 hash function is

```
H=((C>>15)^C)&0xFFFF,
```

so `H` is a 16-bit index, to be used with an index table of size $2^{16} = 64$ K. In addition to the pointers `P`, the index table contains also the contexts `C`. Thus if a context `C` is hashed to an index `H`, the encoder expects to find the same context `C` in location `H` of the index table. This is called *context confirmation*. If the encoder finds something else, or if it finds a null pointer, it sets `P` to null and hashes an order-3 context. If the order-3 context confirmation also fails, the algorithm hashes the order-2 context, and if that also fails, the algorithm sets `P` to null and writes a literal on the compressed stream. This method attempts to find the highest-order context seen so far.

LZP4 uses order-5 contexts and a multistep lookup process. In step 1, the rightmost four bytes `I` of the context are hashed to create a 16-bit index `H` according to the following:

```
H=((I>>15)^I)&0xFFFF.
```

Then `H` is used as an index to the index table that has 64K entries, each corresponding to one value of `H`. Each entry points to the start of a list linking nodes that have the same hash value. Suppose that the contexts `abcde`, `xbcde`, and `mnopq` hash to the same index `H` = 13 (i.e., the hash function computes the same index 13 when applied to `bcde` and `nopq`) and we are looking for context `xbcde`. Location 13 of the index table would point to a list with nodes for these contexts (and possibly others that have also been hashed to 13). The list is traversed until a node is found with `bcde`. This node points to a second list linking `a`, `x`, and perhaps other symbols that precede `bcde`. The second list is traversed until a node with `x` is found. That node contains a pointer to the most recent occurrence of context `xbcde` in the search buffer. If a node with `x` is not found, a literal is written to the compressed stream.

This complex lookup procedure is used by LZP4 because a 5-byte context does not fit comfortably in a single word in most current computers.

6.20 Repetition Finder

All the dictionary-based methods described so far have one thing in common: they use a large memory buffer as a dictionary that holds fragments of text found so far. The dictionary is used to locate strings of symbols that repeat. The method described here is different. Instead of a dictionary it uses a fixed-size array of integers to find previous occurrences of strings of text. The array size equals the square of the alphabet size, so it is not very large. The method is due to Hidetoshi Yokoo [Yokoo 91], who elected not to call it LZHY but left it nameless. The reason a name of the form LZxx was not used is that the method does not employ a traditional Ziv-Lempel dictionary. The reason it was

left nameless is that it does not compress very well and should therefore be considered the first step in a new field of research rather than a mature, practical method.

The method alternates between two modes, normal and repeat. It starts in the normal mode, where it inputs symbols and encodes them using adaptive Huffman. When it identifies a repeated string it switches to the "repeat" mode where it outputs an escape symbol, followed by the length of the repeated string.

Assume that the input stream consists of symbols $x_1x_2\ldots$ from an alphabet A. Both encoder and decoder maintain an array REP of dimensions $|A|\times|A|$ that is initialized to all zeros. For each input symbol x_i, the encoder (and decoder) compute a value y_i according to $y_i = i - \text{REP}[x_{i-1}, x_i]$, and then update $\text{REP}[x_{i-1}, x_i] := i$. The 13-symbol string

x_i:	X	A	B	C	D	E	Y	A	B	C	D	E	Z
i :	1		3		5		7		9		11		13

results in the following y values:

$i =$	1	2	3	4	5	6	7	8	9	10	11	12	13
$y_i =$	1	2	3	4	5	6	7	8	6	6	6	6	13
$x_{i-1}x_i$:	XA	AB	BC	CD	DE	EY	YA	AB	BC	CD	DE	EZ	

Table 6.36a shows the state of array REP after the eighth symbol has been input and encoded. Table 6.36b shows the state of REP after all 13 symbols have been input and encoded.

	A	B	C	D	E	...	X	Y	Z
A		3							
B			4						
C				5					
D					6				
E							7		
⋮									
X	2								
Y	8								
Z									

	A	B	C	D	E	...	X	Y	Z
A		9							
B			10						
C				11					
D					12				
E									13
⋮									
X	2								
Y	8								
Z									

Table 6.36: (a) REP at $i = 8$. (b) REP at $i = 13$.

Perhaps a better way to explain the way y is calculated is by the expression

$$y_i = \begin{cases} 1, & \text{for } i = 1, \\ i, & \text{for } i > 1 \text{ and first occurrence of } x_{i-1}x_i, \\ \min(k), & \text{for } i > 1 \text{ and } x_{i-1}x_i \text{ identical to } x_{i-k-1}x_{i-k}. \end{cases}$$

This shows that y is either i or is the distance k between the current string $x_{i-1}x_i$ and its most recent copy $x_{i-k-1}x_{i-k}$. However, recognizing a repetition of a string is done

by means of array REP and without using any dictionary (which is the main point of this method).

When a string of length l repeats itself in the input, l consecutive identical values of y are generated, and this is the signal for the encoder to switch to the "repeat" mode. As long as consecutive different values of y are generated, the encoder stays in the "normal" mode, where it encodes x_i in adaptive Huffman and outputs it. When the encoder senses $y_{i+1} = y_i$, it outputs x_i in the normal mode, and then enters the "repeat" mode. In the example above this happens for $i = 9$, so the string XABCDEYAB is output in the normal mode.

Once in the "repeat" mode, the encoder inputs more symbols and calculates y values until it finds the first value that differs from y_i. In our example this happens at $i = 13$, when the Z is input. The encoder compresses the string "CDE" (corresponding to $i = 10, 11, 12$) in the "repeat" mode by emitting an (encoded) escape symbol, followed by the (encoded) length of the repeated string (3 in our example). The encoder then switches back to the normal mode, where it saves the y value for Z as y_i and inputs the next symbol.

The escape symbol must be an extra symbol, one that's not included in the alphabet A. Notice that only two y values, y_{i-1} and y_i, need be saved at any time. Notice also that the method is not very efficient, since it senses the repeating string "too late" and encodes the first two repeating symbols in the normal mode. In our example only three of the five repeating symbols are encoded in the "repeat" mode.

The decoder inputs and decodes the first nine symbols, decoding them into the string XABCDEYAB while updating array REP and calculating y values. When the escape symbol is input, i has the value 9 and y_i has the value 6. The decoder inputs and decodes the length, 3, and now it has to figure out the repeated string of length 3 using just the data in array REP, not any of the previously decoded input. Since $i = 9$ and y_i is the distance between this string and its copy, the decoder knows that the copy started at position $i - y_i = 9 - 6 = 3$ of the input. It scans REP, looking for a 3. It finds it at position REP[A,B], so it starts looking for a 4 in row B of REP. It finds it in REP[B,C], so the first symbol of the required string is C. Looking for a 5 in row C, the decoder finds it in REP[C,D], so the second symbol is D. Looking now for a 6 in row D, the decoder finds it in REP[D,E].

This is how a repeated string can be decoded without maintaining a dictionary.

Both encoder and decoder store values of i in REP, so an entry of REP should be at least two bytes long. This way i can have values of up to $64\text{K} - 1 \approx 65{,}500$, so the input has to be encoded in blocks of size 64K. For an alphabet of 256 symbols, the size of REP should therefore be $256 \times 256 \times 2 = 128$ Kbytes, not very large. For larger alphabets REP may have to be uncomfortably large.

In the normal mode, symbols (including the escape) are encoded using adaptive Huffman (Section 5.3). In the repeat mode, lengths are encoded in a recursive prefix code denoted $Q_k(i)$, where k is a positive integer (see Section 2.2 for prefix codes). Assuming that i is an integer whose binary representation is 1α, the prefix code $Q_k(i)$ of i is defined by

$$Q_0(i) = 1^{|\alpha|}0\alpha, \qquad Q_k(i) = \begin{cases} 0, & i = 1, \\ 1Q_{k-1}(i-1), & i > 1, \end{cases}$$

where $|\alpha|$ is the length of α and $1^{|\alpha|}$ is a string of $|\alpha|$ ones. Table 6.37 shows some of the proposed codes; however, any of the prefix codes of Section 2.2 can be used instead of the $Q_k(i)$ codes proposed here.

i	α	$Q_0(i)$	$Q_1(i)$	$Q_2(i)$
1	null	0	0	0
2	0	100	10	10
3	1	101	1100	110
4	00	11000	1101	11100
5	01	11001	111000	11101
6	10	11010	111001	1111000
7	11	11011	111010	1111001
8	000	1110000	111011	1111010
9	001	1110001	11110000	1111011

Table 6.37: Proposed Prefix Code.

The developer of this method, Hidetoshi Yokoo, indicates that compression performance is not very sensitive to the precise value of k, and he proposes $k = 2$ for best overall performance.

As mentioned earlier, the method is not very efficient, which is why it should be considered the start of a new field of research where repeated strings are identified without the need for a large dictionary.

6.21 GIF Images

GIF—the graphics interchange format—was developed by Compuserve Information Services in 1987 as an efficient, compressed graphics file format, which allows for images to be sent between different computers. The original version of GIF is known as GIF 87a. The current standard is GIF 89a and, at the time of writing, can be freely obtained as the file `http://delcano.mit.edu/info/gif.txt`. GIF is not a data compression method; it is a graphics file format that uses a variant of LZW to compress the graphics data (see [Blackstock 87]). This section reviews only the data compression aspects of GIF.

In compressing data, GIF is very similar to `compress` and uses a dynamic, growing dictionary. It starts with the number of bits per pixel b as a parameter. For a monochromatic image, $b = 2$; for an image with 256 colors or shades of gray, $b = 8$. The dictionary starts with 2^{b+1} entries and is doubled in size each time it fills up, until it reaches a size of $2^{12} = 4{,}096$ entries, where it remains static. At such a point, the encoder monitors the compression ratio and may decide to discard the dictionary at any point and start with a new, empty one. When making this decision, the encoder emits the value 2^b as the *clear code*, which is the sign for the decoder to discard its dictionary.

The pointers, which get longer by 1 bit from one dictionary to the next, are accumulated and are output in blocks of 8-bit bytes. Each block is preceded by a header

that contains the block size (255 bytes maximum) and is terminated by a byte of eight zeros. The last block contains, just before the terminator, the eof value, which is $2^b + 1$. An interesting feature is that the pointers are stored with their least significant bit on the left. Consider, for example, the following 3-bit pointers 3, 7, 4, 1, 6, 2, and 5. Their binary values are 011, 111, 100, 001, 110, 010, and 101, so they are packed in 3 bytes |10101001|11000011|11110...|.

The GIF format is commonly used today by web browsers, but it is not an efficient image compressor. GIF scans the image row by row, so it discovers pixel correlations within a row, but not between rows. We can say that GIF compression is inefficient because GIF is one-dimensional while an image is two-dimensional. An illustrative example is the two simple images of Figure 7.4a,b (Section 7.3). Saving both in GIF89 has resulted in file sizes of 1053 and 1527 bytes, respectively.

Most graphics file formats use some kind of compression. For more information on those files, see [Murray and vanRyper 94].

6.22 RAR and WinRAR

The popular RAR software is the creation of Eugene Roshal, who started it as his university doctoral dissertation. RAR is an acronym that stands for Roshal ARchive (or Roshal ARchiver). The current developer is Eugene's brother Alexander Roshal. The following is a list of its most important features:

- RAR is currently available from [rarlab 06], as shareware, for Windows (WinRAR), Pocket PC, Macintosh OS X, Linux, DOS, and FreeBSD. WinRAR has a graphical user interface, whereas the other versions support only a command line interface.

- WinRAR provides complete support for RAR and ZIP archives and can decompress (but not compress) CAB, ARJ, LZH, TAR, GZ, ACE, UUE, BZ2, JAR, ISO, 7Z, and Z archives.

- In addition to compressing data, WinRAR can encrypt data with the advanced encryption standard (AES-128).

- WinRAR can compress files up to 8,589 billion Gb in size (approximately 9×10^{18} bytes).

- Files compressed in WinRAR can be self-extracting (SFX) and can come from different volumes.

- It is possible to combine many files, large and small, into a so-called "solid" archive, where they are compressed together. This may save 10–50% compared to compressing each file individually.

- The RAR software can optionally generate recovery records and recovery volumes that make it possible to reconstruct damaged archives. Redundant data, in the form of a Reed-Solomon error-correcting code, can optionally be added to the compressed file for increased reliability.

The user can specify the amount of redundancy (as a percentage of the original data size) to be built into the recovery records of a RAR archive. A large amount of redundancy makes the RAR file more resistant to data corruption, thereby allowing recovery from more serious damage. However, any redundant data reduces compression efficiency, which is why compression and reliability are opposite concepts.

- Authenticity information may be included for extra security. RAR software saves information on the last update and name of the archive.

- An intuitive wizard especially designed for novice users. The wizard makes it easy to use all of RAR's features through a simple question and answer procedure.

- Excellent drag-and-drop features. The user can drag files from WinRAR to other programs, to the desktop, or to any folder. An archive dropped on WinRAR is immediately opened to display its files. A file dropped on an archive that's open in WinRAR is immediately added to the archive. A group of files or folders dropped on WinRAR automatically becomes a new archive.

- RAR offers NTFS and Unicode support (see [ntfs 06] for NTFS).

- WinRAR is available in over 35 languages.

These features, combined with excellent compression, good compression speed, an attractive graphical user interface, and backup facilities, make RAR one of the best overall compression tools currently available.

The RAR algorithm is generally considered proprietary, and the following quote (from [donationcoder 06]) sheds some light on what this implies

> The fact that the RAR encoding algorithm is proprietary is an issue worth considering. It means that, unlike ZIP and 7z and almost all other compression algorithms, only the official WinRAR programs can create RAR archives (although other programs can decompress them). Some would argue that this is unhealthy for the software industry and that standardizing on an open format would be better for everyone in the long run. But for most users, these issues are academic; WinRAR offers great support for both ZIP and RAR formats.

Proprietary

A term that indicates that a party, or proprietor, exercises private ownership, control, or use over an item of property, usually to the exclusion of other parties.

—From `http://www.wikipedia.org/`

The following illuminating description was obtained directly from Eugene Roshal, the designer of RAR. (The source code of the RAR decoder is available at [unrarsrc 06], but only on condition that it is not used to reverse-engineer the encoder.)

RAR has two compression modes, general and special. The general mode employs an LZSS-based algorithm similar to ZIP Deflate (Section 6.25). The size of the sliding dictionary in RAR can be varied from 64 KB to 4 MB (with a 4 MB default value), and the minimum match length is 2. Literals, offsets, and match lengths are compressed further by a Huffman coder (recall that Deflate works similarly).

Starting at version 3.0, RAR also uses a special compression mode to improve the compression of text-like data. This mode is based on the PPMD algorithm (also known as PPMII) by Dmitry Shkarin.

RAR contains a set of filters to preprocess audio (in `wav` or `au` formats), true color data, tables, and executables (32-bit x86 and Itanium, see note below). A data-analyzing module selects the appropriate filter and the best algorithm (general or PPMD), depending on the data to be compressed.

(Note: The 80x86 family of processors was originally developed by Intel with a word length of 16 bits. Because of its tremendous success, its architecture was extended to 32-bit words and is currently referred to as IA-32 [Intel Architecture, 32-bit]. See [IA-32 06] for more information. The Itanium is an IA-64 microprocessor developed jointly by Hewlett-Packard and Intel.)

In addition to its use as Roshal ARchive, the acronym RAR has many other meanings, a few of which are listed here.
(1) Remote Access Router (a network device used to connect remote sites via private lines or public carriers). (2) Royal Anglian Regiment, a unit of the British Army. (3) Royal Australian Regiment, a unit of the Australian Army. (4) Resource Adapter, a specific module of the Java EE platform. (5) The airport code of Rarotonga, Cook Islands. (6) Revise As Required (editors' favorite). (7) Road Accident Rescue.
See more at [acronyms 06].

Rarissimo, by [softexperience 06], is a file utility intended to automatically compress and decompress files in WinRAR. Rarissimo by itself is useless. It can be used only if WinRAR is already installed in the computer. The Rarissimo user specifies a number of folders for Rarissimo to watch, and Rarissimo compresses or decompresses any files that have been modified in these folders. It can also move the modified files to target folders. The user also specifies how often (in multiples of 10 sec) Rarissimo should check the folders.

For each folder to be watched, the user has to specify RAR or UnRAR and a target folder. If RAR is specified, then Rarissimo employs WinRAR to compress each file that has been modified since the last check, and then moves that file to the target folder. If the target folder resides on another computer, this move amounts to an FTP transfer. If UnRAR has been specified, then Rarissimo employs WinRAR to decompress all the RAR-compressed files found in the folder and then moves them to the target folder.

An important feature of Rarissimo is that it preserves NTFS alternate streams (see [ntfs 06]). This means that Rarissimo can handle Macintosh files that happen to reside on the PC; it can compress and decompress them while preserving their data and resource forks.

6.23 The V.42bis Protocol

The V.42bis protocol is a set of rules, or a standard, published by the ITU-T (page 248) for use in fast modems. It is based on the existing V.32bis protocol and is supposed to be used for fast transmission rates, up to 57.6K baud. Thomborson [Thomborson 92] in a detailed reference for this standard. The ITU-T standards are recommendations, but they are normally followed by all major modem manufacturers. The standard contains specifications for data compression and error correction, but only the former is discussed here.

V.42bis specifies two modes: a *transparent* mode, where no compression is used, and a *compressed* mode using an LZW variant. The former is used for data streams that don't compress well and may even cause expansion. A good example is an already compressed file. Such a file looks like random data; it does not have any repetitive patterns, and trying to compress it with LZW will fill up the dictionary with short, two-symbol, phrases.

The compressed mode uses a growing dictionary, whose initial size is negotiated between the modems when they initially connect. V.42bis recommends a dictionary size of 2,048 entries. The minimum size is 512 entries. The first three entries, corresponding to pointers 0, 1, and 2, do not contain any phrases and serve as special codes. Code 0 (enter transparent mode—ETM) is sent when the encoder notices a low compression ratio, and it decides to start sending uncompressed data. (Unfortunately, V.42bis does not say how the encoder should test for low compression.) Code 1 is FLUSH, to flush data. Code 2 (STEPUP) is sent when the dictionary is almost full and the encoder decides to double its size. A dictionary is considered almost full when its size exceeds that of a special threshold (which is also negotiated by the modems).

When the dictionary is already at its maximum size and it becomes full, V.42bis recommends a *reuse* procedure. The least-recently-used phrase is located and deleted, to make room for a new phrase. This is done by searching the dictionary from entry 256 for the first phrase that is not a prefix to any other phrase. Suppose that the phrase `abcd` is found, and there are no phrases of the form `abcdx` for any `x`. This means that `abcd` has not been used since it was created, and that it is the oldest such phrase. It therefore makes sense to delete it, since it reflects an old pattern in the input stream. This way, the dictionary always reflects recent patterns in the input.

> ...there is an ever-increasing body of opinion which holds that The Ultra-Complete Maximegalon Dictionary is not worth the fleet of lorries it takes to cart its microstored edition around in. Strangely enough, the dictionary omits the word "floopily," which simply means "in the manner of something which is floopy."
>
> —Douglas Adams, *Life, the Universe, and Everything* (1982)

6.24 Various LZ Applications

ARC is a compression/archival/cataloging program developed by Robert A. Freed in the mid-1980s. It offers good compression and the ability to combine several files into one file, called an *archive*. Currently (early 2003) ARC is outdated and has been superseded by the newer PK applications.

PKArc is an improved version of ARC. It was developed by Philip Katz, who has founded the PKWare company [PKWare 03], which still markets the PKzip, PKunzip, and PKlite software. The PK programs are faster and more general than ARC and also provide for more user control. Past editions of this book have more information on these applications.

LHArc, from Haruyasu Yoshizaki, and LHA, by Haruhiko Okumura and Haruyasu Yoshizaki, use adaptive Huffman coding with features drawn from LZSS. The LZ window size may be up to 16K bytes. ARJ is another older compression tool by Robert K. Jung that uses the same algorithm, but increases the LZ window size to 26624 bytes. A similar program, called ICE (for the old MS-DOS operating system), seems to be a faked version of either LHarc or LHA. [Okumura 98] has some information about LHArc and LHA as well as a history of data compression in Japan.

Two newer applications, popular with Microsoft Windows users, are RAR/WinRAR [rarlab 06] (Section 6.22) and ACE/WinAce [WinAce 03]. They use LZ with a large search buffer (up to 4 Mb) combined with Huffman codes. They are available for several platforms and offer many useful features.

6.25 Deflate: Zip and Gzip

Deflate is a popular compression method that was originally used in the well-known Zip and Gzip software and has since been adopted by many applications, the most important of which are (1) the HTTP protocol ([RFC1945 96] and [RFC2616 99]), (2) the PPP compression control protocol ([RFC1962 96] and [RFC1979 96]), (3) the PNG (Portable Network Graphics, Section 6.27) and MNG (Multiple-Image Network Graphics) graphics file formats ([PNG 03] and [MNG 03]), and (4) Adobe's PDF (Portable Document File, Section 11.13) [PDF 01].

Phillip W. Katz was born in 1962. He received a bachelor's degree in computer science from the University of Wisconsin at Madison. Always interested in writing software, he started working in 1984 as a programmer for Allen-Bradley Co. developing programmable logic controllers for the industrial automation industry. He later worked for Graysoft, another software company, in Milwaukee, Wisconsin. At about that time he became interested in data compression and founded PKWare in 1987 to develop, implement, and market software products such as PKarc and PKzip. For a while, the company was very successful selling the programs as shareware.

Always a loner, Katz suffered from personal and legal problems, started drinking heavily, and died on April 14, 2000 from complications related to chronic alcoholism. He was 37 years old.

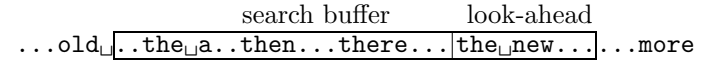
After his death, PKWare was sold, in March 2001, to a group of investors. They changed its management and the focus of its business. PKWare currently targets the corporate market, and emphasizes compression combined with encryption. Their product line runs on a wide variety of platforms.

Deflate was designed by Philip Katz as a part of the Zip file format and implemented in his PKZIP software [PKWare 03]. Both the ZIP format and the Deflate method are in the public domain, which allowed implementations such as Info-ZIP's Zip and Unzip (essentially, PKZIP and PKUNZIP clones) to appear on a number of platforms. Deflate is described in [RFC1951 96].

The most notable implementation of Deflate is zlib, a portable and free compression library ([zlib 03] and [RFC1950 96]) by Jean-Loup Gailly and Mark Adler who designed and implemented it to be free of patents and licensing requirements. This library (the source code is available at [Deflate 03]) implements the ZLIB and GZIP file formats ([RFC1950 96] and [RFC1952 96]), which are at the core of most Deflate applications, including the popular Gzip software.

Deflate is based on a variation of LZ77 combined with Huffman codes. We start with a simple overview based on [Feldspar 03] and follow with a full description based on [RFC1951 96].

The original LZ77 method (Section 6.3) tries to match the text in the look-ahead buffer to strings already in the search buffer. In the example

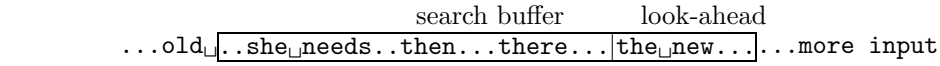


the look-ahead buffer starts with the string `the␣`, which can be matched to one of three strings in the search buffer. The longest match has a length of 4. LZ77 writes tokens on the compressed stream, where each token is a triplet (offset, length, next symbol). The third component is needed in cases where no string has been matched (imagine having `che` instead of `the` in the look-ahead buffer) but it is part of every token, which reduces the performance of LZ77. The LZ77 algorithm variation used in Deflate eliminates the third component and writes a pair (offset, length) on the compressed stream. When no match is found, the unmatched character is written on the compressed stream instead of a token. Thus, the compressed stream consists of three types of entities: literals (unmatched characters), offsets (termed "distances" in the Deflate literature), and lengths. Deflate actually writes Huffman codes on the compressed stream for these entities, and it uses two code tables—one for literals and lengths and the other for distances. This makes sense because the literals are normally bytes and are therefore in the interval $[0, 255]$, and the lengths are limited by Deflate to 258. The distances, however, can be large numbers because Deflate allows for a search buffer of up to 32Kbytes.

When a pair (length, distance) is determined, the encoder searches the table of literal/length codes to find the code for the length. This code (we later use the term "edoc" for it) is then replaced by a Huffman code that's written on the compressed stream. The encoder then searches the table of distance codes for the code of the

distance and writes that code (a special prefix code with a fixed, 5-bit prefix) on the compressed stream. The decoder knows when to expect a distance code, because it always follows a length code.

The LZ77 variant used by Deflate defers the selection of a match in the following way. Suppose that the two buffers contain



The longest match is 3. Before selecting this match, the encoder saves the `t` from the look-ahead buffer and starts a secondary match where it tries to match `he␣new...` with the search buffer. If it finds a longer match, it outputs `t` as a literal, followed by the longer match. There is also a 3-valued parameter that controls this secondary match attempt. In the "normal" mode of this parameter, if the primary match was long enough (longer than a preset parameter), the secondary match is reduced (it is up to the implementor to decide how to reduce it). In the "high-compression" mode, the encoder always performs a full secondary match, thereby improving compression but spending more time on selecting a match. In the "fast" mode, the secondary match is omitted.

Deflate compresses an input data file in blocks, where each block is compressed separately. Blocks can have different lengths and the length of a block is determined by the encoder based on the sizes of the various prefix codes used (their lengths are limited to 15 bits) and by the memory available to the encoder (except that blocks in mode 1 are limited to 65,535 bytes of uncompressed data). The Deflate decoder must be able to decode blocks of any size. Deflate offers three modes of compression, and each block can be in any mode. The modes are as follows:

1. No compression. This mode makes sense for files or parts of files that are incompressible (i.e., random) or already compressed, or for cases where the compression software is asked to segment a file without compression. A typical case is a user who wants to move an 8 Gb file to another computer but has only a DVD "burner." The user may want to segment the file into two 4 Gb segments without compression. Commercial compression software based on Deflate can use this mode of operation to segment the file. This mode uses no code tables. A block written on the compressed stream in this mode starts with a special header indicating mode 1, followed by the length `LEN` of the data, followed by `LEN` bytes of literal data. Notice that the maximum value of `LEN` is 65,535.

2. Compression with fixed code tables. Two code tables are built into the Deflate encoder and decoder and are always used. This speeds up both compression and decompression and has the added advantage that the code tables don't have to be written on the compressed stream. The compression performance, however, may suffer if the data being compressed is statistically different from the data used to set up the code tables. Literals and match lengths are located in the first table and are replaced by a code (called "edoc") that is, in turn, replaced by a prefix code that's output to the compressed stream. Distances are located in the second table and are replaced by special prefix codes that are output to the compressed stream. A block written on the compressed stream in this mode starts with a special header indicating mode 2, followed by the compressed data in the form of prefix codes for the literals and lengths, and special prefix codes for the distances. The block ends with a single prefix code for end-of-block.

3. Compression with individual code tables generated by the encoder for the particular data that's being compressed. A sophisticated Deflate encoder may gather statistics about the data as it compresses blocks, and may be able to construct improved code tables as it proceeds from block to block. There are two code tables, for literals/lengths and for distances. They again have to be written on the compressed stream, and they are written in compressed format. A block written by the encoder on the compressed stream in this mode starts with a special header, followed by (1) a compressed Huffman code table and (2) the two code tables, each compressed by the Huffman codes that preceded them. This is followed by the compressed data in the form of prefix codes for the literals, lengths, and distances, and ends with a single code for end-of-block.

6.25.1 The Details

Each block starts with a 3-bit header where the first bit is 1 for the last block in the file and 0 for all other blocks. The remaining two bits are 00, 01, or 10, indicating modes 1, 2, or 3, respectively. Notice that a block of compressed data does not always end on a byte boundary. The information in the block is sufficient for the decoder to read all the bits of the compressed block and recognize the end of the block. The 3-bit header of the next block immediately follows the current block and may therefore be located at any position in a byte on the compressed stream.

The format of a block in mode 1 is as follows:

1. The 3-bit header 000 or 100.

2. The rest of the current byte is skipped, and the next four bytes contain `LEN` and the one's complement of `LEN` (as unsigned 16-bit numbers), where `LEN` is the number of data bytes in the block. This is why the block size in this mode is limited to 65,535 bytes.

3. `LEN` data bytes.

The format of a block in mode 2 is different:

1. The 3-bit header 001 or 101.

2. This is immediately followed by the fixed prefix codes for literals/lengths and the special prefix codes of the distances.

3. Code 256 (rather, its prefix code) designating the end of the block.

Mode 2 uses two code tables: one for literals and lengths and the other for distances. The codes of the first table are not what is actually written on the compressed stream, so in order to remove ambiguity, the term "edoc" is used here to refer to them. Each edoc is converted to a prefix code that's output to the compressed stream. The first table allocates edocs 0 through 255 to the literals, edoc 256 to end-of-block, and edocs 257–285 to lengths. The latter 29 edocs are not enough to represent the 256 match lengths of 3 through 258, so extra bits are appended to some of those edocs. Table 6.38 lists the 29 edocs, the extra bits, and the lengths represented by them. What is actually written on the compressed stream is prefix codes of the edocs (Table 6.39). Notice that edocs 286 and 287 are never created, so their prefix codes are never used. We show later that Table 6.39 can be represented by the sequence of code lengths

$$\underbrace{8,8,\ldots,8}_{144},\underbrace{9,9,\ldots,9}_{112},\underbrace{7,7,\ldots,7}_{24},\underbrace{8,8,\ldots,8}_{8}, \tag{6.1}$$

Code	Extra bits	Lengths	Code	Extra bits	Lengths	Code	Extra bits	Lengths
257	0	3	267	1	15,16	277	4	67–82
258	0	4	268	1	17,18	278	4	83–98
259	0	5	269	2	19–22	279	4	99–114
260	0	6	270	2	23–26	280	4	115–130
261	0	7	271	2	27–30	281	5	131–162
262	0	8	272	2	31–34	282	5	163–194
263	0	9	273	3	35–42	283	5	195–226
264	0	10	274	3	43–50	284	5	227–257
265	1	11,12	275	3	51–58	285	0	258
266	1	13,14	276	3	59–66			

Table 6.38: Literal/Length Edocs for Mode 2.

edoc	Bits	Prefix codes
0–143	8	00110000–10111111
144–255	9	110010000–111111111
256–279	7	0000000–0010111
280–287	8	11000000–11000111

Table 6.39: Huffman Codes for Edocs in Mode 2.

but any Deflate encoder and decoder include the entire table instead of just the sequence of code lengths. There are edocs for match lengths of up to 258, so the look-ahead buffer of a Deflate encoder can have a maximum size of 258, but can also be smaller.

Examples. If a string of 10 symbols has been matched by the LZ77 algorithm, Deflate prepares a pair (length, distance) where the match length 10 becomes edoc 264, which is written as the 7-bit prefix code 0001000. A length of 12 becomes edoc 265 followed by the single bit 1. This is written as the 7-bit prefix code 0001010 followed by 1. A length of 20 is converted to edoc 269 followed by the two bits 01. This is written as the nine bits 0001101|01. A length of 256 becomes edoc 284 followed by the five bits 11110. This is written as 11000101|11110. A match length of 258 is indicated by edoc 285 whose 8-bit prefix code is 11000110. The end-of-block edoc of 256 is written as seven zero bits.

The 30 distance codes are listed in Table 6.40. They are special prefix codes with fixed-size 5-bit prefixes that are followed by extra bits in order to represent distances in the interval $[1, 32768]$. The maximum size of the search buffer is therefore 32,768, but it can be smaller. The table shows that a distance of 6 is represented by 00100|1, a distance of 21 becomes the code 01000|101, and a distance of 8195 corresponds to code 11010|000000000010.

6.25.2 Format of Mode-3 Blocks

In mode 3, the encoder generates two prefix code tables, one for the literals/lengths and

Code	Extra bits	Distance	Code	Extra bits	Distance	Code	Extra bits	Distance
0	0	1	10	4	33–48	20	9	1025–1536
1	0	2	11	4	49–64	21	9	1537–2048
2	0	3	12	5	65–96	22	10	2049–3072
3	0	4	13	5	97–128	23	10	3073–4096
4	1	5,6	14	6	129–192	24	11	4097–6144
5	1	7,8	15	6	193–256	25	11	6145–8192
6	2	9–12	16	7	257–384	26	12	8193–12288
7	2	13–16	17	7	385–512	27	12	12289–16384
8	3	17–24	18	8	513–768	28	13	16385–24576
9	3	25–32	19	8	769–1024	29	13	24577–32768

Table 6.40: Thirty Prefix Distance Codes in Mode 2.

the other for the distances. It uses the tables to encode the data that constitutes the block. The encoder can generate the tables in any way. The idea is that a sophisticated Deflate encoder may collect statistics as it inputs the data and compresses blocks. The statistics are used to construct better code tables for later blocks. A naive encoder may use code tables similar to the ones of mode 2 or may even not generate mode 3 blocks at all. The code tables have to be written on the compressed stream, and they are written in a highly-compressed format. As a result, an important part of Deflate is the way it compresses the code tables and outputs them. The main steps are (1) Each table starts as a Huffman tree. (2) The tree is rearranged to bring it to a standard format where it can be represented by a sequence of code lengths. (3) The sequence is compressed by run-length encoding to a shorter sequence. (4) The Huffman algorithm is applied to the elements of the shorter sequence to assign them Huffman codes. This creates a Huffman tree that is again rearranged to bring it to the standard format. (5) This standard tree is represented by a sequence of code lengths which are written, after being permuted and possibly truncated, on the output. These steps are described in detail because of the originality of this unusual method.

Recall that the Huffman code tree generated by the basic algorithm of Section 5.2 is not unique. The Deflate encoder applies this algorithm to generate a Huffman code tree, then rearranges the tree and reassigns the codes to bring the tree to a standard form where it can be expressed compactly by a sequence of code lengths. (The result is reminiscent of the canonical Huffman codes of Section 5.2.8.) The new tree satisfies the following two properties:

1. The shorter codes appear on the left, and the longer codes appear on the right of the Huffman code tree.

2. When several symbols have codes of the same length, the (lexicographically) smaller symbols are placed on the left.

The first example employs a set of six symbols A–F with probabilities 0.11, 0.14, 0.12, 0.13, 0.24, and 0.26, respectively. Applying the Huffman algorithm results in a tree similar to the one shown in Figure 6.41a. The Huffman codes of the six symbols

are 000, 101, 001, 100, 01, and 11. The tree is then rearranged and the codes reassigned to comply with the two requirements above, resulting in the tree of Figure 6.41b. The new codes of the symbols are 100, 101, 110, 111, 00, and 01. The latter tree has the advantage that it can be fully expressed by the sequence 3, 3, 3, 3, 2, 2 of the lengths of the codes of the six symbols. The task of the encoder in mode 3 is therefore to generate this sequence, compress it, and write it on the compressed stream.

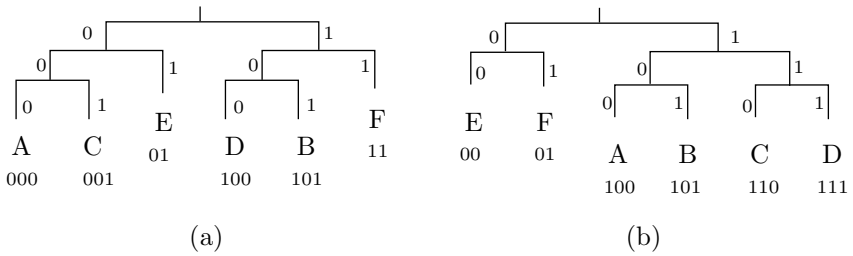


Figure 6.41: Two Huffman Trees.

The code lengths are limited to at most four bits each. Thus, they are integers in the interval $[0, 15]$, which implies that a code can be at most 15 bits long (this is one factor that affects the Deflate encoder's choice of block lengths in mode 3).

The sequence of code lengths representing a Huffman tree tends to have runs of identical values and can have several runs of the same value. For example, if we assign the probabilities 0.26, 0.11, 0.14, 0.12, 0.24, and 0.13 to the set of six symbols A–F, the Huffman algorithm produces 2-bit codes for A and E and 3-bit codes for the remaining four symbols. The sequence of these code lengths is 2, 3, 3, 3, 2, 3.

The decoder reads a compressed sequence, decompresses it, and uses it to reproduce the standard Huffman code tree for the symbols. We first show how such a sequence is used by the decoder to generate a code table, then how it is compressed by the encoder.

Given the sequence 3, 3, 3, 3, 2, 2, the Deflate decoder proceeds in three steps as follows:

1. Count the number of codes for each code length in the sequence. In our example, there are no codes of length 1, two codes of length 2, and four codes of length 3.

2. Assign a base value to each code length. There are no codes of length 1, so they are assigned a base value of 0 and don't require any bits. The two codes of length 2 therefore start with the same base value 0. The codes of length 3 are assigned a base value of 4 (twice the number of codes of length 2). The C code shown here (after [RFC1951 96]) was written by Peter Deutsch. It assumes that step 1 leaves the number of codes for each code length `n` in `bl_count[n]`.

```
code = 0;
bl_count[0] = 0;
for (bits = 1; bits <= MAX_BITS; bits++) {
  code = (code + bl_count[bits-1]) << 1;
  next code[bits] = code;
}
```

3. Use the base value of each length to assign consecutive numerical values to all the codes of that length. The two codes of length 2 start at 0 and are therefore 00 and 01. They are assigned to the fifth and sixth symbols E and F. The four codes of length 3 start at 4 and are therefore 100, 101, 110, and 111. They are assigned to the first four symbols A–D. The C code shown here (by Peter Deutsch) assumes that the code lengths are in `tree[I].Len` and it generates the codes in `tree[I].Codes`.

```
for (n = 0; n <= max code; n++) {
  len = tree[n].Len;
  if (len != 0) {
    tree[n].Code = next_code[len];
    next_code[len]++;
  }
}
```

In the next example, the sequence 3, 3, 3, 3, 3, 2, 4, 4 is given and is used to generate a table of eight prefix codes. Step 1 finds that there are no codes of length 1, one code of length 2, five codes of length 3, and two codes of length 4. The length-1 codes are assigned a base value of 0. There are zero such codes, so the next group is assigned the base value of twice 0. This group contains one code, so the next group (length-3 codes) is assigned base value 2 (twice the sum $0+1$). This group contains five codes, so the last group is assigned base value of 14 (twice the sum $2+5$). Step 3 simply generates the five 3-bit codes 010, 011, 100, 101, and 110 and assigns them to the first five symbols. It then generates the single 2-bit code 00 and assigns it to the sixth symbol. Finally, the two 4-bit codes 1110 and 1111 are generated and assigned to the last two (seventh and eighth) symbols.

Given the sequence of code lengths of Equation (6.1), we apply this method to generate its standard Huffman code tree (listed in Table 6.39).

Step 1 finds that there are no codes of lengths 1 through 6, that there are 24 codes of length 7, 152 codes of length 8, and 112 codes of length 9. The length-7 codes are assigned a base value of 0. There are 24 such codes, so the next group is assigned the base value of $2(0+24)=48$. This group contains 152 codes, so the last group (length-9 codes) is assigned base value $2(48+152)=400$. Step 3 simply generates the 24 7-bit codes 0 through 23, the 152 8-bit codes 48 through 199, and the 112 9-bit codes 400 through 511. The binary values of these codes are listed in Table 6.39.

> "Must you deflate romantic rhetoric? Besides, the Astabigans have plenty of visitors from other worlds who will be viewing her."
>
> —Roger Zelazny, *Doorways in the Sand*

It is now clear that a Huffman code table can be represented by a short sequence (termed SQ) of code lengths (herein called CLs). This sequence is special in that it tends to have runs of identical elements, so it can be highly compressed by run-length encoding. The Deflate encoder compresses this sequence in a three-step process where the first step employs run-length encoding; the second step computes Huffman codes for the run lengths and generates another sequence of code lengths (to be called CCLs) for those Huffman codes. The third step writes a permuted, possibly truncated sequence of the CCLs on the compressed stream.

Step 1. When a CL repeats more than three times, the encoder considers it a run. It appends the CL to a new sequence (termed SSQ), followed by the special flag 16 and by a 2-bit repetition factor that indicates 3–6 repetitions. A flag of 16 is therefore preceded by a CL and followed by a factor that indicates how many times to copy the CL. Thus, for example, if the sequence to be compressed contains six consecutive 7's, it is compressed to 7, 16, 10_2 (the repetition factor 10_2 indicates five consecutive occurrences of the same code length). If the sequence contains 10 consecutive code lengths of 6, it will be compressed to 6, 16, 11_2, 16, 00_2 (the repetition factors 11_2 and 00_2 indicate six and three consecutive occurrences, respectively, of the same code length).

Experience indicates that CLs of zero are very common and tend to have long runs. (Recall that the codes in question are codes of literals/lengths and distances. Any given data file to be compressed may be missing many literals, lengths, and distances.) This is why runs of zeros are assigned the two special flags 17 and 18. A flag of 17 is followed by a 3-bit repetition factor that indicates 3–10 repetitions of CL 0. Flag 18 is followed by a 7-bit repetition factor that indicates 11–138 repetitions of CL 0. Thus, six consecutive zeros in a sequence of CLs are compressed to 17, 11_2, and 12 consecutive zeros in an SQ are compressed to 18, 01_2.

The sequence of CLs is compressed in this way to a shorter sequence (to be termed SSQ) of integers in the interval $[0, 18]$. An example may be the sequence of 28 CLs

4, 4, 4, 4, 4, 3, 3, 3, 6, 6, 6, 6, 6, 6, 6, 6, 6, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2

that's compressed to the 16-number SSQ

4, 16, 01_2, 3, 3, 3, 6, 16, 11_2, 16, 00_2, 17, 11_2, 2, 16, 00_2,

or, in decimal, 4, 16, 1, 3, 3, 3, 6, 16, 3, 16, 0, 17, 3, 2, 16, 0.

Step 2. Prepare Huffman codes for the SSQ in order to compress it further. Our example SSQ contains the following numbers (with their frequencies in parentheses): 0(2), 1(1), 2(1), 3(5), 4(1), 6(1), 16(4), 17(1). Its initial and standard Huffman trees are shown in Figure 6.42a,b. The standard tree can be represented by the SSQ of eight lengths 4, 5, 5, 1, 5, 5, 2, and 4. These are the lengths of the Huffman codes assigned to the eight numbers 0, 1, 2, 3, 4, 6, 16, and 17, respectively.

Step 3. This SSQ of eight lengths is now extended to 19 numbers by inserting zeros in the positions that correspond to unused CCLs.

Position:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
CCL:	4	5	5	1	5	0	5	0	0	0	0	0	0	0	0	0	2	4	0

Next, the 19 CCLs are permuted according to

Position:	16	17	18	0	8	7	9	6	10	5	11	4	12	3	13	2	14	1	15
CCL:	2	4	0	4	0	0	0	5	0	0	0	5	0	1	0	5	0	5	0

The reason for the permutation is to end up with a sequence of 19 CCLs that's likely to have trailing zeros. The SSQ of 19 CCLs minus its trailing zeros is written on the compressed stream, preceded by its actual length, which can be between 4 and 19. Each CCL is written as a 3-bit number. In our example, there is just one trailing zero, so the 18-number sequence 2, 4, 0, 4, 0, 0, 0, 5, 0, 0, 0, 5, 0, 1, 0, 5, 0, 5 is written on the compressed stream as the final, compressed code of one prefix-code table. In mode 3,

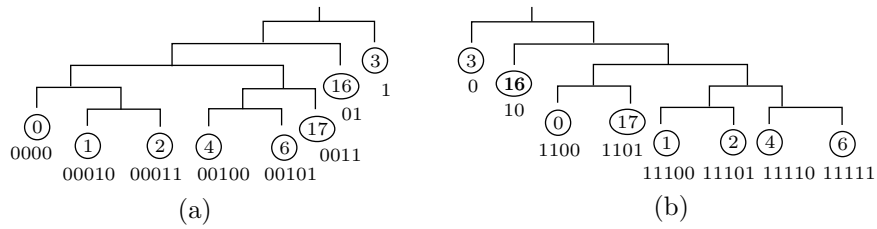


Figure 6.42: Two Huffman Trees for Code Lengths.

each block of compressed data requires two prefix-code tables, so two such sequences are written on the compressed stream.

A reader finally reaching this point (sweating profusely with such deep concentration on so many details) may respond with the single word "insane." This scheme of Phil Katz for compressing the two prefix-code tables per block is devilishly complex and hard to follow, but it works!

The format of a block in mode 3 is as follows:

1. The 3-bit header 010 or 110.

2. A 5-bit parameter `HLIT` indicating the number of codes in the literal/length code table. This table has codes 0–256 for the literals, code 256 for end-of-block, and the 30 codes 257–286 for the lengths. Some of the 30 length codes may be missing, so this parameter indicates how many of the length codes actually exist in the table.

3. A 5-bit parameter `HDIST` indicating the size of the code table for distances. There are 30 codes in this table, but some may be missing.

4. A 4-bit parameter `HCLEN` indicating the number of CCLs (there may be between 4 and 19 CCLs).

5. A sequence of `HCLEN + 4` CCLs, each a 3-bit number.

6. A sequence SQ of `HLIT + 257` CLs for the literal/length code table. This SQ is compressed as explained earlier.

7. A sequence SQ of `HDIST + 1` CLs for the distance code table. This SQ is compressed as explained earlier.

8. The compressed data, encoded with the two prefix-code tables.

9. The end-of-block code (the prefix code of edoc 256).

Each CCL is written on the output as a 3-bit number, but the CCLs are Huffman codes of up to 19 symbols. When the Huffman algorithm is applied to a set of 19 symbols, the resulting codes may be up to 18 bits long. It is the responsibility of the encoder to ensure that each CCL is a 3-bit number and none exceeds 7. The formal definition [RFC1951 96] of Deflate does not specify how this restriction on the CCLs is to be achieved.

6.25.3 The Hash Table

This short section discusses the problem of locating a match in the search buffer. The buffer is 32 Kb long, so a linear search is too slow. Searching linearly for a match to any string requires an examination of the entire search buffer. If Deflate is to be able to compress large data files in reasonable time, it should use a sophisticated search method.

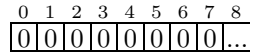
The method proposed by the Deflate standard is based on a hash table. This method is strongly recommended by the standard, but is not required. An encoder using a different search method is still compliant and can call itself a Deflate encoder. Those unfamiliar with hash tables should consult any text on data structures.

Instead of separate look-ahead and search buffers, the encoder should have one, 32 Kb buffer. The buffer is filled up with input data and initially all of it is a look-ahead buffer. In the original LZ77 method, once symbols have been examined, they are moved into the search buffer. The Deflate encoder, in contrast, does not move the data in its buffer and instead moves a pointer (or a separator) from left to right, to indicate the point where the look-ahead buffer ends and the search buffer starts. Short, 3-symbol strings from the look-ahead buffer are hashed and added to the hash table. After hashing a string, the encoder examines the hash table for matches. Assuming that a symbol occupies n bits, a string of three symbols can have values in the interval $[0, 2^{3n}-1]$. If $2^{3n}-1$ isn't too large, the hash function can return values in this interval, which tends to minimize the number of collisions. Otherwise, the hash function can return values in a smaller interval, such as 32 Kb (the size of the Deflate buffer).

We demonstrate the principles of Deflate hashing with the 17-symbol string

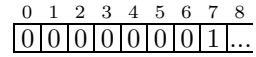
```
abbaabbaabaabaaaa
12345678901234567
```

Initially, the entire 17-location buffer is the look-ahead buffer and the hash table is empty



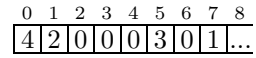
We assume that the first triplet `abb` hashes to 7. The encoder outputs the raw symbol `a`, moves this symbol to the search buffer (by moving the separator between the two buffers to the right), and sets cell 7 of the hash table to 1.

```
a|bbaabbaabaabaaaa
1 2345678901234567
```



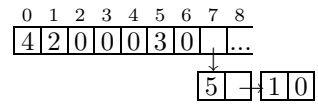
The next three steps hash the strings `bba`, `baa`, and `aab` to, say, 1, 5, and 0. The encoder outputs the three raw symbols `b`, `b`, and `a`, moves the separator, and updates the hash table as follows:

```
abba|abbaabaabaaaa
1234 5678901234567
```



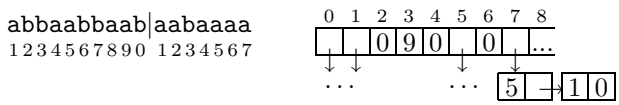
Next, the triplet `abb` is hashed, and we already know that it hashes to 7. The encoder finds 1 in cell 7 of the hash table, so it looks for a string that starts with `abb` at position 1 of its buffer. It finds a match of size 6, so it outputs the pair $(5-1, 6)$. The offset (4) is the difference between the start of the current string (5) and the start of the matching string (1). There are now two strings that start with `abb`, so cell 7 should point to both. It therefore becomes the start of a linked list (or chain) whose data items are 5 and 1. Notice that the 5 precedes the 1 in this chain, so that later searches of the chain will find the 5 first and will therefore tend to find matches with the smallest offset, because those have short Huffman codes.

```
abbaa|bbaabaabaaaa
12345 678901234567
```



Six symbols have been matched at position 5, so the next position to consider is $6+5=11$. While moving to position 11, the encoder hashes the five 3-symbol strings it

finds along the way (those that start at positions 6 through 10). They are `bba`, `baa`, `aab`, `aba`, and `baa`. They hash to 1, 5, 0, 3, and 5 (we arbitrarily assume that `aba` hashes to 3). Cell 3 of the hash table is set to 9, and cells 0, 1, and 5 become the starts of linked chains.



Continuing from position 11, string `aab` hashes to 0. Following the chain from cell 0, we find matches at positions 4 and 8. The latter match is longer and matches the 5-symbol string `aabaa`. The encoder outputs the pair $(11-8,5)$ and moves to position $11+5=16$. While doing so, it also hashes the 3-symbol strings that start at positions 12, 13, 14, and 15. Each hash value is added to the hash table. (End of example.)

It is clear that the chains can become very long. An example is an image file with large uniform areas where many 3-symbol strings will be identical, will hash to the same value, and will be added to the same cell in the hash table. Since a chain must be searched linearly, a long chain defeats the purpose of a hash table. This is why Deflate has a parameter that limits the size of a chain. If a chain exceeds this size, its oldest elements should be truncated. The Deflate standard does not specify how this should be done and leaves it to the discretion of the implementor. Limiting the size of a chain reduces the compression quality but can reduce the compression time significantly. In situations where compression time is unimportant, the user can specify long chains.

Also, selecting the longest match may not always be the best strategy; the offset should also be taken into account. A 3-symbol match with a small offset may eventually use fewer bits (once the offset is replaced with a variable-length code) than a 4-symbol match with a large offset.

6.25.4 Conclusions

Deflate is a general-purpose lossless compression algorithm that has proved valuable over the years as part of several popular compression programs. The method requires memory for the look-ahead and search buffers and for the two prefix-code tables. However, the memory size needed by the encoder and decoder is independent of the size of the data or the blocks. The implementation is not trivial, but is simpler than that of some modern methods such as JPEG 2000 or MPEG. Compression algorithms that are geared for specific types of data, such as audio or video, may perform better than Deflate on such data, but Deflate normally produces compression factors of 2.5 to 3 on text, slightly less for executable files, and somewhat more for images. Most important, even in the worst case, Deflate expands the data by only 5 bytes per 32 Kb block. Finally, free implementations that avoid patents are available. Notice that the original method, as designed by Phil Katz, has been patented (United States patent 5,051,745, September 24, 1991) and assigned to PKWARE.

6.26 LZMA and 7-Zip

LZMA is the main (as well as the default) algorithm used in the popular `7z` (or `7-Zip`) compression software [7z 06]. Both `7z` and LZMA are the creations of Igor Pavlov. The software runs on Windows and is free. Both LZMA and `7z` were designed to provide high compression, fast decompression, and low memory requirements for decompression.

The main feature of `7z` is its open architecture. The software can currently compress data in one of six algorithms, and can in principle support any new compression methods. The current algorithms are the following:

1. LZMA. This is a sophisticated extension of LZ77
2. PPMD. A variant of Dmitry Shkarin's PPMdH
3. BCJ. A converter for 32-bit x86 executables (see note)
4. BCJ2. A similar converter
5. BZip2. An implementation of the Burrows-Wheeler method (Section 11.1)
6. Deflate. An LZ77-based algorithm (Section 6.25)

(Note: The 80x86 family of processors was originally developed by Intel with a word length of 16 bits. Because of its tremendous success, its architecture had been extended to 32-bit words and is currently referred to as IA-32 [Intel Architecture, 32-bit]. See [IA-32 06] for more information.)

Other important features of `7z` are the following:

- In addition to compressing and decompressing data in LZMA, the `7z` software can compress and decompress data in ZIP, GZIP, and BZIP2, it can pack and unpack data in the TAR format, and it can decompress data originally compressed in RAR, CAB, ARJ, LZH, CHM, Z, CPIO, RPM, and DEB.
- When compressing data in the ZIP or GZIP formats, `7z` provides compression ratios that are 2–10% better than those achieved by PKZip and WinZip.
- A data file compressed in `7z` includes a decompressor; it is self-extracting.
- The `7z` software is integrated with Windows Shell [Horstmann 06].
- It constitutes a powerful file manager.
- It offers a powerful command line version.
- It has a plugin for FAR Manager.
- It includes localizations for 60 languages.
- It can encrypt the compressed data with the advanced encryption standard (AES-256) algorithm, based on a 256-bit encryption key [FIPS197 01]. (The original AES algorithm, also known as Rijndael, was based on 128-bit keys.) The user inputs a text string as a passphrase, and `7z` employs a key-derivation function to obtain the 256-bit key from the passphrase. This function is based on the SHA-256 hash algorithm [SHA256 02] and it computes the key by generating a very long string based on the passphrase and using it as the input to the hash function. The 256 bits output by the hash function become the encryption key.

To generate the string, 7z starts with the passphrase, encoded in the UTF-16 encoding scheme (two bytes per character, see [UTF16 06]). It then generates and concatenates $256\text{K} = 2^{18}$ pairs (passphrase, integer) with 64-bit integers ranging from 0 to $2^{18} - 1$. If the passphrase is p symbols long, then each pair is $2p + 8$ bytes long (the bytes are arranged in little endian), and the total length of the final string is $2^{18}(2p + 8)$ bytes; very long!

The term Little Endian means that the low-order byte (the little end) of a number or a string is stored in memory at the lowest address (it comes first). For example, given the 4-byte number $b_3b_2b_1b_0$, if the least-significant byte b_0 is stored at address A, then the most-significant byte b_3 will be stored at address $A + 3$.

LZMA (which stands for Lempel-Ziv-Markov chain-Algorithm) is the default compression algorithm of 7z. It is an LZ77 variant designed for high compression ratios and fast decompression. A free software development kit (SDK) with the LZMA source code, in C, C++, C#, and Java, is available at [7z 06] to anyone who wants to study, understand, or modify this algorithm. The main features of LZMA are the following:

- Compression speeds of about 1 Mb/s on a 2 GHz processor. Decompression speeds of about 10–20 Mb/s are typically obtained on a 2 GHz CPU
- The size of the decompressor can be as little as 2 Kb (if optimized for size of code), and it requires only 8–32Kb (plus the dictionary size) for its data.
- The dictionary size is variable and can be up to 4 Gb, but the current implementation limits it to 1 Gb.
- It supports multithreading and the Pentium 4's hyperthreading. (Hyperthreading is a technology that allows resource sharing and partitioning while providing a multiprocessor environment in a unique CPU.) The current LZMA encoder can use one or two threads, and the LZMA decoder can use only one thread.
- The LZMA decoder uses only integer operations and can easily be implemented on any 32-bit processor (implementing it on a 16-bit CPU is more involved).

The compression principle of LZMA is similar to that of Deflate (Section 6.25), but uses range encoding (Section 5.10.1) instead of Huffman coding. This complicates the encoder, but results in better compression (recall that range encoding is an integer-based version of arithmetic coding and can compress very close to the entropy of the data), while minimizing the number of renormalizations needed. Range encoding is implemented in binary such that shifts are used to divide integers, thereby avoiding the slow "divide" operation.

Recall that LZ77 searches the search buffer for the longest string that matches the look-ahead buffer, then writes on the compressed stream a triplet (distance, length, next symbol) where "distance" is the distance from the string in the look-ahead buffer to its match in the search buffer. Thus, three types of data are written on the output, literals (the next symbol, often an ASCII code), lengths (relatively small numbers), and distances (which can be large numbers if the search buffer is large).

LZMA also outputs these three types. If nothing matches the look-ahead buffer, a literal (a value in the interval $[0, 255]$) is output. If a match is found, then a pair (length,

distance) is output (after being encoded by the range coder). Because of the large size of the search buffer, a short, 4-entry, distance-history array is used that always contains the four most-recent distances that have been determined and output. If the distance of the current match equals one of the four distances in this array, then a pair (length, index) is output (after being encoded by the range coder), where "index" is a 2-bit index to the distance-history array.

Locating matches in the search buffer is done by hashing two bytes, the current byte in the look-ahead buffer and the byte immediately to its right (but see the next paragraph for more details). The output of the hash function is an index to an array (the hash-array). The size of the array is selected as the power of 2 that's closest to half the dictionary size, so the output of the hash function depends on the size of the dictionary. For example, if the dictionary size is 256 Mb (or 2^{28}), then the size of the array is 2^{27} and the hash function has to compute and output 27-bit numbers. The large size of the array minimizes hash collisions.

Experienced readers may have noticed the problem with the previous paragraph. If only two bytes are hashed, then the input to the hash function is only 16 bits, so there can be only $2^{16} = 65{,}536$ different inputs. The hash function can therefore compute only 65,536 distinct outputs (indexes to the hash-array) regardless of the size of the array. The full story is therefore more complex. The LZMA encoder can hash 2, 3, or 4 bytes and the number of items in the hash-array is selected by the encoder depending on the size of the dictionary. For example, a 1-Gb ($= 2^{30}$ bytes) dictionary results in a hash-array of size 512M $= 2^{29}$ items (where each item in the hash-array is a 32-bit integer). In order to take advantage of such a large hash-array, the encoder hashes four bytes. Four bytes constitute 32 bits, which provide the hash function with 2^{32} distinct inputs. The hash function converts each input to a 29-bit index. (Thus, many inputs are converted to the same index.)

Table 6.44 lists several user options and shows how the user can control the encoder by setting the Match Finder parameter to certain values. The value `bt4`, for example, specifies the binary tree mode with 2-3-4 bytes hashing. For simplicity, the remainder of this description talks about hashing two bytes (see Table 6.44 for various hashing options).

The output of the hash function is used as an index to a hash-array and the user can choose one of two search methods, hash-chain (the fast method) or binary tree (the efficient method).

In the fast method, the output of the hash function is an index to a hash-array of lists. Each array location is the start of a list of distances. Thus, if the two bytes hashed are `XY` and the result of the hash is index 123, then location 123 of the hash-array is a list of distances to pairs `XY` in the search buffer. These lists can be very long, so LZMA checks only the first 24 distances. These correspond to the 24 most-recent occurrences of `XY` (the number 24 is a user-controlled parameter). The best of the 24 matches is selected, encoded, and output. The distance of this match is then added to the start of the list, to become the first match found when array location 123 is checked again. This method is reminiscent of match searching in LZRW4 (Section 6.12).

In the binary tree method, the output of the hash function is an index to a hash-array of binary search trees (Section 6.4). Initially, the hash-array is empty and there are no trees. As data is read and encoded, trees are generated and grow, and each

data byte becomes a node in one of the trees. A binary tree is created for every pair of consecutive bytes in the original data. Thus, if the data contains `good␣day`, then trees are generated for the pairs `go`, `oo`, `od`, `d␣`, `␣d`, `da`, and `ay`. The total number of nodes in those trees is eight (the size of the data). Notice that the two occurrences of `o` end up as nodes in different trees. The first resides in the tree for `oo`, and the second ends up in the tree of `od`.

The following example illustrates how this method employs the binary trees to find matches and how the trees are unpdated. The example assumes that the match-finder parameter (Table 6.44) is set to `bt2`, indicating 2-byte hashing.

We assume that the data to be encoded is already fully stored in a long buffer (the dictionary). Five strings that start with the pair `ab` are located at various points as shown

```
...abm...abcd2...abcx...abcd1...aby...
   1       2       3       5       7
   1       4       0       7       8
```

We denote by `p(n)` the index (location) of string `n` in the dictionary. Thus, `p(abm...)` is 11 and `p(abcd2)` is 24. Each time a binary tree T is searched and a match is selected, T is rearranged and is updated according to the following two rules:

1. The tree must always remain a binary search tree.

2. If $\texttt{p(n1)} < \texttt{p(n2)}$, then `n2` cannot be in any subtree of `n1`. This implies that (a) the latest string (the one with the greatest index) is always the root of the tree and (b) indexes decrease as we slide down the tree. The result is a tree where recent strings are located near the root.

We also assume that the pair `ab` of bytes is hashed to 62. Figure 6.43 illustrates how the binary tree for the pair `ab` is created, kept up to date, and searched. The following numbered items refer to the six parts of this figure.

1. When the LZMA encoder gets to location 11 and finds `a`, it hashes this byte and the `b` that follows, to obtain 62. It examines location 62 of the hash-array and finds it empty. It then generates a new binary tree (for the pair `ab`) with one node containing the pointer 11, and sets location 62 of the hash-array to point to this tree. There is no match, the byte `a` at 11 is output as a literal, and the encoder proceeds to hash the next pair `bm`.

2. The next pair `ab` is found by the LZMA encoder when it gets to location 24. Hashing produces the same 62, and location 62 of the hash-array is found to point to a binary tree whose root (which is so far its only node) contains 11. The encoder places 24 as the new root with 11 as its right subtree, because `abcd2...` is smaller than `abm...` (strings are compared lexicographically). The encoder matches `abcd2...` with `abm...` and the match length is 2. The encoder continues with `cd2...`, but before it does that, it hashes the pair `bc` and either appends it to the binary tree for `bc` (if such a tree exists) or generates a new tree.

3. The next pair `ab` is found at location 30. Hashing produces the same 62. The encoder places 30 (`abcx...`) as the new root with 24 (`abcd2...`) as its left subtree and 11 (`abm..`) as its right subtree. The better match is `abcd2...`, and the match length is 3. The next two pairs `bc` and `cx` are appended to their respective trees (if such trees exist), and the encoder continues with the pair `x.`.

4. The next occurrence of `ab` is found at location 57 and is hashed to 62. It becomes the root of the tree, with 30 (`abcx...`) as its right subtree. The best match is with `abcd2...` where the match length is 4. The encoder continues with the string `1...` found at location 61.

5. In the last step of this example, the encoder finds a pair `ab` at location 78. This string (`aby...`) becomes the new root, with 57 as its left subtree. The match length is 2.

6. Now assume that location 78 contains the string `abk..` instead of `aby...` Since `abm..` is greater than `abk..`, it must be moved to the right subtree of `abk..`, resulting in a completely different tree.

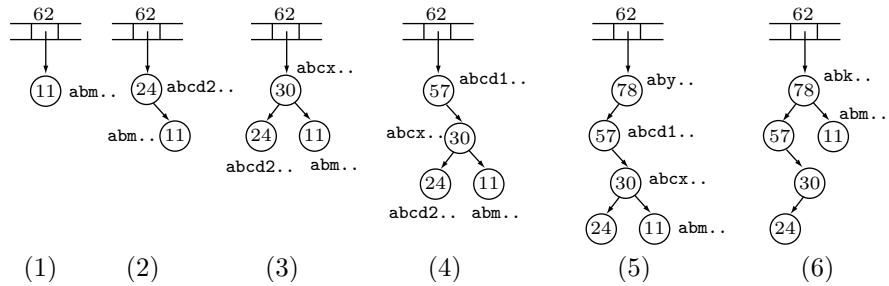


Figure 6.43: Binary Search Trees in LZMA.

Past versions of LZMA used a data structure called a Patricia trie (see page 357 for the definition of a trie), but this structure proved inefficient and has been eliminated.

We end this description with some of the options that can be specified by the user when LZMA is invoked.

- `-a{N}`. The compression mode. 0, 1, and 2 specify the fast, normal, and max modes, with 2 as the default. (The latest version, currently in its beta stage, does not support the max mode.)

- `-d{N}`. The Dictionary size. N is an integer between 0 and 30, and the dictionary size is set to 2^N. The default is $N = 23$ (an 8-Mb dictionary), and the current maximum is $N = 30$ (a 1-Gb dictionary).

- `-mf{MF_ID}`. The Match Finder. It specifies the method for finding matches and limits the number of bytes to be hashed. The memory requirements depend on this choice and on the dictionary size d. Table 6.44 lists the choices. The default value of `MF_ID` in the normal, max, and ultra modes is `bt4` and in the fast and fastest modes it is `hc4`.

We would like to thank Igor Pavlov for contributing important information and details to the material of this section.

MF_ID	Memory	Description
bt2	$d \times 9.5 + 1$ Mb	Binary Tree with 2 bytes hashing
bt3	$d \times 11.5 + 4$ Mb	Binary Tree with 3 bytes hashing
bt4	$d \times 11.5 + 4$ Mb	Binary Tree with 4 bytes hashing
hc4	$d \times 7.5 + 4$ Mb	Hash Chain with 4 bytes hashing

Table 6.44: LZMA Match Finder Options.

Notes for Table 6.44:

1. bt4 uses three hash tables with 2^{10} items for hashing two bytes, with 2^{16} items for hashing three bytes, and with a variable size to hash four bytes. Only the latter table points to binary trees. The other tables point to positions in the input buffer.

2. bt3 uses two hash tables, one with 2^{10} items for hashing two bytes and the other with a variable size to hash three bytes

3. bt2 uses only one hash table with 2^{16} items.

4. bt2 and bt3 also can find almost all the matches, but bt4 is faster.

6.27 PNG

The portable network graphics (PNG) file format has been developed in the mid-1990s by a group (the PNG development group [PNG 03]) headed by Thomas Boutell. The project was started in response to the legal issues surrounding the GIF file format (Section 6.21). The aim of this project was to develop a sophisticated graphics file format that will be flexible, will support many different types of images, will be easy to transmit over the Internet, and will be unencumbered by patents. The design was finalized in October 1996, and its main features are as follows:

1. It supports images with 1, 2, 4, 8, and 16 bitplanes.
2. Sophisticated color matching.
3. A transparency feature with very fine control provided by an alpha channel.
4. Lossless compression by means of Deflate combined with pixel prediction.
5. Extensibility: New types of meta-information can be added to an image file without creating incompatibility with existing applications.

Currently, PNG is supported by many image viewers and web browsers on various platforms. This subsection is a general description of the PNG format, followed by the details of the compression method it uses.

A PNG file consists of chunks that can be of various types and sizes. Some chunks contain information that's essential for displaying the image, and decoders must be able to recognize and process them. Such chunks are referred to as "critical chunks." Other chunks are ancillary. They may enhance the display of the image or may contain meta-data such as the image title, author's name, creation and modification dates and times, etc. (but notice that decoders may choose not to process such chunks). New, useful types of chunks can also be registered with the PNG development group.

A chunk consists of the following parts: (1) size of the data field, (2) chunk name, (3) data field, and (4) a 32-bit cyclical redundancy code (CRC, Section 6.32). Each chunk has a 4-letter name of which (1) the first letter is uppercase for a critical chunk and lowercase for an ancillary chunk, (2) the second letter is uppercase for standard

chunks (those defined by or registered with the PNG group) and lowercase for a private chunk (an extension of PNG), (3) the third letter is always uppercase, and (4) the fourth letter is uppercase if the chunk is "unsafe to copy" and lowercase if it is "safe to copy."

Any PNG-aware application will process all the chunks it recognizes. It can safely ignore any ancillary chunk it doesn't recognize, but if it finds a critical chunk it cannot recognize, it has to stop and issue an error message. If a chunk cannot be recognized but its name indicates that it is safe to copy, the application may safely read and rewrite it even if it has altered the image. However, if the application cannot recognize an "unsafe to copy" chunk, it must discard it. Such a chunk should not be rewritten on the new PNG file. Examples of "safe to copy" are chunks with text comments or those indicating the physical size of a pixel. Examples of "unsafe to copy" are chunks with gamma/color correction data or palette histograms.

The four critical chunks defined by the PNG standard are IHDR (the image header), PLTE (the color palette), IDAT (the image data, as a compressed sequence of filtered samples), and IEND (the image trailer). The standard also defines several ancillary chunks that are deemed to be of general interest. Anyone with a new chunk that may also be of general interest may register it with the PNG development group and have it assigned a public name (a second letter in uppercase).

The PNG file format uses a 32-bit CRC (Section 6.32) as defined by the ISO standard 3309 [ISO 84] or ITU-T V.42 [ITU-T 94]. The CRC polynomial is

$$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1.$$

The particular calculation proposed in the PNG standard employs a precalculated table that speeds up the computation significantly.

A PNG file starts with an 8-byte signature that helps software to identify it as PNG. This is immediately followed by an IHDR chunk with the image dimensions, number of bitplanes, color type, and data filtering and interlacing. The remaining chunks must include a PLTE chunk if the color type is palette, and one or more adjacent IDAT chunks with the compressed pixels. The file must end with an IEND chunk. The PNG standard defines the order of the public chunks, whereas private chunks may have their own ordering constraints.

An image in PNG format may have one of the following five color types: RGB with 8 or 16 bitplanes, palette with 1, 2, 4, or 8 bitplanes, grayscale with 1, 2, 4, 8, or 16 bitplanes, RGB with alpha channel (with 8 or 16 bitplanes), and grayscale with alpha channel (also with 8 or 16 bitplanes). An alpha channel implements the concept of transparent color. One color can be designated transparent, and pixels of that color are not drawn on the screen (or printed). Instead of seeing those pixels, a viewer sees the background behind the image. The alpha channel is a number in the interval $[0, 2^{bp}-1]$, where bp is the number of bitplanes. Assuming that the background color is B, a pixel in the transparent color P is painted in color $(1-\alpha)B+\alpha P$. This is a mixture of $\alpha\%$ background color and $(1-\alpha)\%$ pixel color.

Perhaps the most intriguing feature of the PNG format is the way it handles interlacing. Interlacing makes it possible to display a rough version of the image on the screen, then improve it gradually until it reaches its final, high-resolution form. The special interlacing used in PNG is called Adam 7 after its developer, Adam M. Costello.

PNG divides the image into blocks of 8×8 pixels each, and displays each block in seven steps. In step 1, the entire block is filled up with copies of the top-left pixel (the one marked "1" in Figure 6.45a). In each subsequent step, the block's resolution is doubled by modifying half its pixels according to the next number in Figure 6.45a. This process is easiest to understand with an example, such as the one shown in Figure 6.45b.

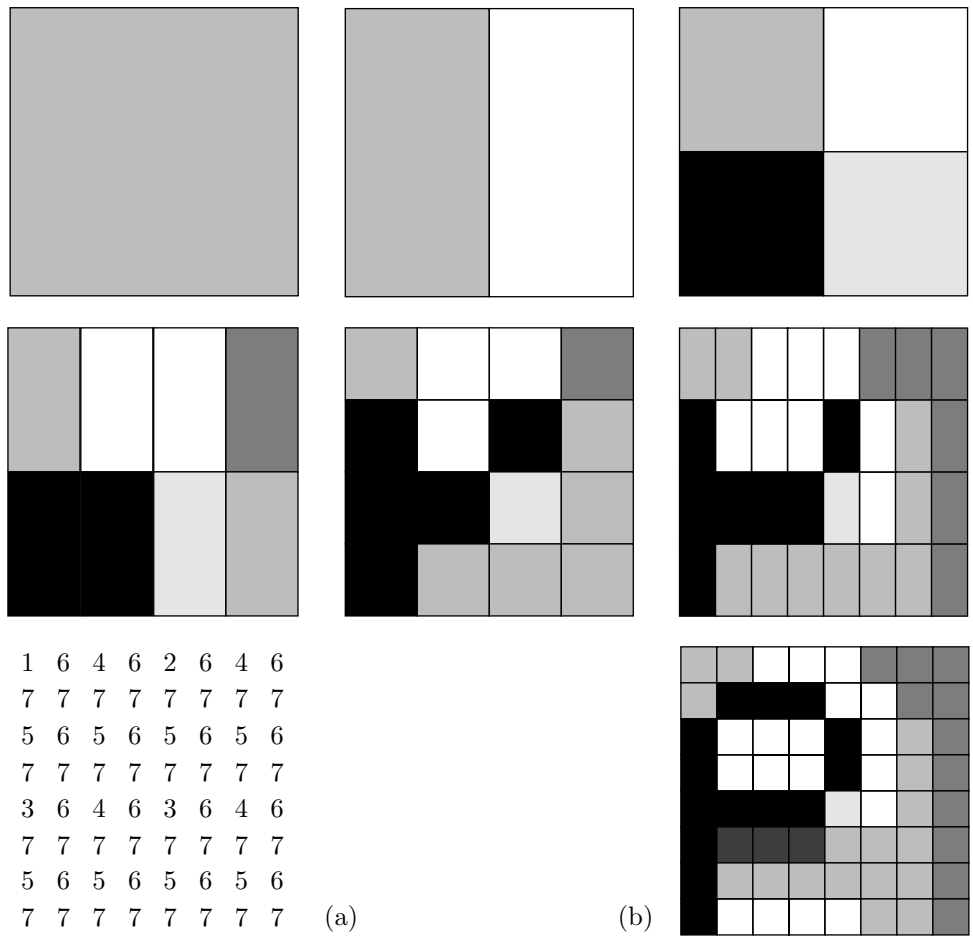


Figure 6.45: Interlacing in PNG.

PNG compression is lossless and is performed in two steps. The first step, termed *delta filtering* (or just *filtering*), converts pixel values to numbers by a process similar to the prediction used in the lossless mode of JPEG (Section 7.10.5). The filtering step calculates a "predicted" value for each pixel and replaces the pixel with the difference between the pixel and its predicted value. The second step employs Deflate to encode the differences. Deflate is the topic of Section 6.25, so only filtering needs be described here.

Filtering does not compress the data. It only transforms the pixel data to a format where it is more compressible. Filtering is done separately on each image row, so an intelligent encoder can switch filters from one image row to the next (this is called *adaptive filtering*). PNG specifies five filtering methods (the first one is simply no filtering) and recommends a simple heuristic for choosing a filtering method for each image row. Each row starts with a byte indicating the filtering method used in it. Filtering is done on bytes, not on complete pixel values. This is easy to understand in cases where a pixel consists of three bytes, specifying the three color components of the pixel. Denoting the three bytes by a, b, and c, we can expect a_i and a_{i+1} to be correlated (and also b_i and b_{i+1}, and c_i and c_{i+1}), but there is no correlation between a_i and b_i. Also, in a grayscale image with 16-bit pixels, it makes sense to compare the most-significant bytes of two adjacent pixels and then the least-significant bytes. Experiments suggest that filtering is ineffective for images with fewer than eight bitplanes, and also for palette images, so PNG recommends no filtering in such cases.

The heuristic recommended by PNG for adaptive filtering is to apply all five filtering types to the row of pixels and select the type that produces the smallest sum of absolute values of outputs. (For the purposes of this test, the filtered bytes should be considered signed differences.)

The five filtering types are described next. The first type (type 0) is no filtering. Filtering type 1 (sub) sets byte $B_{i,j}$ in row i and column j to the difference $B_{i,j} - B_{i-t,j}$, where t is the interval between a byte and its correlated predecessor (the number of bytes in a pixel). Values of t for the various image types and bitplanes are listed in Table 6.46. If $i - t$ is negative, then nothing is subtracted, which is equivalent to having a zero pixel on the left of the row. The subtraction is done modulo 256, and the bytes subtracted are considered unsigned.

Image type	Bit planes	Interval t
Grayscale	1, 2, 4, 8	1
Grayscale	16	2
Grayscale with alpha	8	2
Grayscale with alpha	16	4
Palette	1, 2, 4, 8	1
RGB	8	3
RGB	16	6
RGB with alpha	8	4
RGB with alpha	16	8

Figure 6.46: Interval Between Bytes.

Filtering type 2 (up) sets byte $B_{i,j}$ to the difference $B_{i,j} - B_{i,j-1}$. The subtraction is done as in type 1, but if j is the top image row, no subtraction is done.

Filtering type 3 (average) sets byte $B_{i,j}$ to the difference $B_{i,j} - \left[B_{i-t,j} + B_{i,j-1}\right] \div 2$. The average of the left neighbor and the top neighbor is computed and subtracted from the byte. Any missing neighbor to the left or above is considered zero. Notice that the sum $B_{i-t,j} + B_{i,j-1}$ may be up to 9 bits long. To guarantee that the filtering is lossless

and can be reversed by the decoder, the sum must be computed exactly. The division by 2 is equivalent to a right shift and brings the 9-bit sum down to 8 bits. Following that, the 8-bit average is subtracted from the current byte $B_{i,j}$ modulo 256 and unsigned.

Example. Assume that the current byte $B_{i,j} = 112$, its left neighbor $B_{i-t,j} = 182$, and its top neighbor $B_{i,j-1} = 195$. The average is $(182 + 195) \div 2 = 188$. Subtracting $(112 - 188) \bmod 256$ yields $-76 \bmod 256$ or $256 - 76 = 180$. Thus, the encoder sets $B_{i,j}$ to 180. The decoder inputs the value 180 for the current byte, computes the average in the same way as the encoder to end up with 188, and adds $(180 + 188) \bmod 256$ to obtain 112.

Filtering type 4 (Paeth) sets byte $B_{i,j}$ to $B_{i,j} - \text{PaethPredict}\big[B_{i-t,j}, B_{i,j-1}, B_{i-t,j-1}\big]$. PaethPredict is a function that uses simple rules to select one of its three parameters, then returns that parameter. Those parameters are the left, top, and top-left neighbors. The selected neighbor is then subtracted from the current byte, modulo 256 unsigned. The PaethPredictor function is defined by the following pseudocode:

```
function PaethPredictor (a, b, c)
begin
; a=left, b=above, c=upper left
p:=a+b-c ;initial estimate
pa := abs(p-a) ; compute distances
pb := abs(p-b) ; to a, b, c
pc := abs(p-c)
; return nearest of a,b,c,
; breaking ties in order a,b,c.
if pa<=pb AND pa<=pc then return a
else if pb<=pc then return b
else return c
end
```

PaethPredictor must perform its computations exactly, without overflow. The order in which PaethPredictor breaks ties is important and should not be altered. This order (that's different from the one given in [Paeth 91]) is left neighbor, neighbor above, upper-left neighbor.

PNG is a single-image format, but the PNG development group has also designed an animated companion format named MNG (multiple-image network format), which is a proper superset of PNG.

We are indebted to Cosmin Truţa for reviewing and correcting this subsection.

> Does the world really need yet another graphics format? We believe so. GIF is no longer freely usable,... it would not be all that much easier to implement than a whole new file format. (PNG is designed to be simple to implement, with the exception of the compression engine, which would be needed in any case.) We feel that this is an excellent opportunity to design a new format that fixes some of the known limitations of GIF.
>
> From the PNG standard, RFC 2083, 1999

6.28 XML Compression: XMill

XMill is a special-purpose, efficient software application for the compression of XML (Extensible Markup Language) documents. Its description in this short section is based on [Liefke and Suciu 99], but more details and a free implementation are available from [XMill 03]. First, a few words about XML.

XML is a markup language for documents containing structured information. A markup language is a mechanism to identify structures in a document. A short story or a novel may have very little structure. It may be divided into chapters and may also include footnotes, an introduction, and an epilogue. A cooking recipe has more structure. It starts with the name of the dish and its class (such as salads, soups, etc.). This is followed by the two main structural items: the ingredients and the preparation. The recipe may then describe the proper way to serve the dish and may end with notes, reviews, and comments. A business card is similarly divided into a number of short items.

The XML standard [XML 03] defines a way to add markup to documents, and has proven very popular and successful. An XML file contains data represented as text and also includes tags that identify the types of (or that assign meaning to) various data items. HTML is also a markup language, familiar to many, but it is restrictive in that it defines only certain tags and their meanings. The `<H3>` tag, for example, is defined in HTML and has a certain meaning (a certain header), but anyone wanting to use a tag of, say, `<blah>`, has first to convince the WWW consortium to define it and assign it a meaning, then wait for the various web browsers to support it. XML, in contrast, does not specify a set of tags but provides a facility to define tags and structural relationships between them. The meaning of the tags (their semantics) is later defined by applications that process XML documents or by style sheets.

Here is a simple example of a business card in XML. Most tags specify the start and end of a certain data item; they are wrappers. A tag such as `<red_backgrnd/>` that has a trailing "/" is considered empty. It specifies something to the application that reads and processes the XML file, but that something does not require any extra data.

```
<card xmlns="http://businesscard.org">
   <name>Melvin Schwartzkopf</name>
   <title>Chief person, Monster Inc.</title>
   <email>mschwa@monster.com</email>
   <phone>(212)555-1414</phone>
   <logo url="widget.gif"/>
   <red_backgrnd/>
</card>
```

In summary, an XML file consists of markup and content. There are six types of markup that can occur in an XML document: elements, entity references, comments, processing instructions, marked sections, and document-type declarations. The contents can be any digital data.

The main aim of the developers of XMill was to design and implement a special-purpose XML encoder that will compress an XML file better than a typical compressor will compress just the data of that file. Given a data file A, suppose that a typical

compressor, such as gzip, compresses it to X. Now add XML tags to A, converting it to B and increasing its size in the process. When B is compressed by XMill, the resulting file, Y, should be smaller than X. As an example, the developers had tested XMill on a 98-Mb data file taken from SwissProt, a data base for protein structure. The file was initially compressed by gzip down to 16 Mb. The original file was then converted to XML which increased its size to 165 Mb and was compressed by gzip (to 19 Mb) and by XMill (to 8.6 Mb, albeit after fine-tuning XMill for the specific data in that file). However, since XMill is a special-purpose encoder, it is not expected to perform well on arbitrary data files. The design of XMill is based on the following principles:

1. By itself, XMill is not a compressor. Rather, it is an extensible tool for specifying and applying existing compressors to XML data items. XMill analyzes the XML file, then invokes different compressors to compress different parts of the file. The main compressor used by XMill is gzip, but XMill includes other compressors and can also be linked by the user to any existing compressor.

2. Separate the structure from the raw data. The XML tags and attributes are separated by XMill from the data in the input file and are compressed separately. The data is the contents of XML elements and the values of attributes.

3. Group together items that are related. XMill uses the concept of a *container*. In the above example of business cards, all the URLs are grouped in one container, all the names are grouped in a second container, and so on. Also, all the XML tags and attributes are grouped in the structure container. The user can control the contents of the container by providing *container expressions* on the XMill command line.

4. Use semantic compressors. A container may include data items of a certain type, such as telephone numbers or airport codes. A sophisticated user may have an encoder that compresses such items efficiently. The user may therefore direct XMill to use certain encoders for certain containers, thereby achieving excellent compression for the entire XML input file. XMill comes with several built-in encoders, but any encoder available to the user may be linked to XMill and will be used by it to compress any specified container.

An important part of XMill is a concise language, termed the *container expressions*, that's used to group data items in containers and to specify the proper encoders for the various containers.

XMill was designed to prepare XML files for storage or transmission. Sometimes, an XML file is used in connection with a query processor, where the file has to be searched often. XMill is not a good choice for such an application. Another limitation of XMill is that it performs well only on large files. Any XML file smaller than about 20 Kb will be poorly compressed by XMill because XMill adds overhead in the form of bookkeeping to the compressed file. Small XML files, however, are common in situations where messages in XML format are exchanged between users or between applications.

> I do not know it—it is without name—it is a word unsaid, It is not in any dictionary, utterance, symbol.
>
> —Walt Whitman, *Leaves of Grass*, (1900)

6.29 EXE Compressors

The LZEXE program is freeware originally written in the late 1980s by Fabrice Bellard as a special-purpose utility to compress EXE files (PC executable files). The idea is that an EXE file compressed by LZEXE can be decompressed **and** executed with one command. The decompressor does not write the decompressed file on the disk but loads it in memory, relocates addresses, and executes it! The decompressor uses memory that's eventually used by the program being decompressed, so it does not require any extra RAM. In addition, the decompressor is very small compared with decompressors in self-extracting archives.

The algorithm is based on LZ. It uses a circular queue and a dictionary tree for finding string matches. The position and size of the match are encoded by an auxiliary algorithm based on the Huffman method. Uncompressed bytes are kept unchanged, since trying to compress them any further would have entailed a much more complex and larger decompressor. The decompressor is located at the end of the compressed EXE file and is 330 bytes long (in version 0.91). The main steps of the decoder are as follows:

1. Check the CRC (Section 6.32) to ensure data reliability.
2. Locate itself in high RAM; then move the compressed code in order to leave sufficient room for the EXE file.
3. Decompress the code, check that it is correct, and adjust the segments if bigger than 64K.
4. Decompress the relocation table and update the relocatable addresses of the EXE file.
5. Run the program, updating the CS, IP, SS, and SP registers.

The idea of EXE compressors, introduced by LZEXE, was attractive to both users and software developers, so a few more have been developed:

1. PKlite, from PKWare, is a similar EXE compressor that can also compress .COM files.
2. DIET, by Teddy Matsumoto, is a more general EXE compressor that can compress data files. DIET can act as a monitor, permanently residing in RAM, watching for applications trying to read files from the disk. When an application tries to read a DIET-compressed data file, DIET senses it and does the reading and decompressing in a process that's transparent to the application.

UPX is an EXE compressor started in 1996 by Markus Oberhumer and László Molnár. The current version (as of June 2006) is 2.01. Here is a short quotation from [UPX 03].

UPX is a free, portable, extendable, high-performance executable packer for several different executable formats. It achieves an excellent compression ratio and offers very fast decompression. Your executables suffer no memory overhead or other drawbacks.

6.30 Off-Line Dictionary-Based Compression

The dictionary-based compression methods described in this chapter are distinct, but have one thing in common; they generate the dictionary as they go along, reading data and compressing it. The dictionary is not included in the compressed file and is generated by the decoder in lockstep with the encoder. Thus, such methods can be termed "online." In contrast, the methods described in this section are also based on a dictionary, but can be considered "offline" because they include the dictionary in the compressed file.

BPE. The first method is byte pair encoding (BPE). This is a simple compression method, due to [Gage 94], that often features only mediocre performance. It is described here because (1) it is an example of a multipass method (two-pass compression algorithms are common, but multipass methods are generally considered too slow) and (2) it eliminates only certain types of redundancy and should therefore be applied only to data files that feature this redundancy. (The next method, by [Larsson and Moffat 00], does not suffer from these drawbacks and is much more efficient.) BPE is both an example of an offline dictionary-based compression algorithm and a simple example (perhaps the simplest) of a grammar-based compression method. In addition, the BPE decoder is very small, which makes it ideal for applications where memory size is restricted.

The BPE method is easy to describe. We assume that the data symbols are bytes and we use the term *bigram* for a pair of consecutive bytes. Each pass locates the most-common bigram and replaces it with an unused byte value. Thus, the method performs best on files that have many unused byte values, and one aim of this discussion is to point out the types of data that feature this kind of redundancy. First, however, a small example. Given the character set `A`, `B`, `C`, `D`, `X`, and `Y` and the data file `ABABCABCD` (where `X` and `Y` are unused bytes), the first pass identifies the pair `AB` as the most-common bigram and replaces each of its three occurrences with the single byte `X`. The result is `XXCXCD`. The second pass identifies the pair `XC` as the most-common bigram and replaces each of its two occurrences with the single byte `Y`. The result is `XYYD`, where no bigram occurs more than once. Bigrams that occur just once can also be replaced, if more unused byte values are available. However, each replacement rule must be appended to the dictionary and thus ends up being included in the compressed file. As a result, the BPE encoder stops when no bigram occurs more than once.

What types of data tend to have many unused byte values? The first type that comes to mind is text. Currently (in late 2008), most text files use the well-known ASCII codes to encode text. An ASCII code occupies a byte, but only seven bits constitue the actual code. The eighth bit can be used for a parity check, but is often simply set to zero. Thus, we can expect 128 byte values out of the 256 possible ones to be unused in a typical ASCII text file. A quick glance at an ASCII code table shows that codes 0 through 32 (as well as code 127) are control codes. They indicate commands such as backspace, carriage return, escape, delete, and blank space. It therefore makes sense to expect only a few of those to appear in any given text file.

The validity of these arguments can be checked by a simple test. The following Mathematica code prints the unused byte values in a given text file.

```
scn = ReadList["Hobbit.txt", Byte];
btc = Table[0, {256}];
```

```
Do[btc[[scn[[i]]]] = btc[[scn[[i]]]] + 1, {i, 1, Length[scn]}];
btc
dis = Table[0, {256}]; j = 0;
Do[If[btc[[i]] == 0, {j = j + 1, dis[[j]] = i - 1}], {i, 1, 256}];
Take[dis, j]
```

It was executed on three chapters from the book *Data Compression: The Complete Reference* (4th edition) and on Tolkien's *The Hobbit*. The following results were obtained:

Dcomp3: 0–7, 9–11, 13–30, 126–255.
Dcomp4.1: 0–11, 13–30, 126–255.
Dcomp5: 0–7, 9–11, 13–30, 126–255.
Hobbit: 0–7, 9–11, 13–30, 34–36, 42, 46, 60, 63, 87, 89–92, 95, 122–123, 125–255.

The results of this experiment are clear. More than half the byte values are unused. The 128 values 128 through 255 are unused and the only ASCII control characters used are BS, FF, US, and Space.

Today, more and more text files are encoded in Unicode, but even such files should have many unused byte values. (Notice that most Unicodes are 16 bits, but there are also 8-bit and 32-bit Unicodes.) A typical Unicode text file in an alphabetic language consists of letters, digits, punctuation marks, and accented letters, so the total number of codes is around 100–150, leaving many unused byte values. As an example, the 128 Unicodes for Greek and Coptic [Greek-Unicode 07] are `0370` through `03FF`, and these do not use byte values `00`, `01`, and `04` through `6F`. Naturally, text files in an ideograph-based language, such as Hangul or Cuneiforms, easily use all 256 byte values.

Grayscale images constitute another example of data where many byte values may be unused. A typical image in grayscale may have millions of pixels, each in one of 256 shades of gray, but many shades may be unused. An experiment with the Mathematica code above indicates 41 unused byte values in the well-known `lena` image (raw, 128×128, 1-byte pixels), and 35 unused shades of gray (out of 256) in the familiar, raw format, `baboon` image of the same resolution.

On the other hand, the same images in color (in raw format, with three bytes per pixels) use all the 256 byte values and it is easy to see why. A typical color image may consist of 6–7 million pixels (this is typical for today's digital cameras) and may use only a few thousand colors. However, each color occupies three bytes, so even if a certain color, say (r, g, b)=(108, 56, 213), is unused, there is a good chance that some pixels have color components 108, 56, or 213.

Thus, Gage's method makes sense for compressing text and grayscale images. If it is applied to other types of data files, a large file should be segmented into small sections with unused byte values in each.

At any given time, the method looks only at one pair of bytes, but this algorithm also indirectly takes advantage of longer repeating patterns. Thus, an input file of the form `abcdeabcdfabcdg` compresses quite efficiently. The most-common byte pairs are `ab`, `bc`, and `cd`. If the algorithm selects the first pair and replaces it with the unused byte `x`, the file becomes `xcdexcdfxcdg`. If `xc` is next selected and is replaced by `y`, the result is `ydeydfydg`. Now the pair `yd` is replaced by `z`, to produce `zezfzg`. The compression factor is $15/6 = 2.5$, comparable to (or even exceeding) more efficient methods.

The remainder of this section describes a possible implementation of this method (reference [Gage 94] includes C source code). To compress a file, the entire file must be input into a buffer in memory (if the file is too large, it should be compressed in segments). As an example, we consider an alphabet of the eight symbols `a` through `h` (coded as 0 through 7) and the input file `ababcabcd` (with symbols `e` through `h` unused).

The program constructs the following dictionary (also referred to as a pair-table or a phrase-table), where each zero in the bottom row indicates an unused character.

```
0 1 2 3 4 5 6 7
---------------
a b c d e f g h
1 1 1 1 0 0 0 0
```

The first step is to locate the most-common bigram. The simplest approach is to set up a table of size 256×256, initialize it to all zeros, use the two bytes of the next input bigram as row and column indexes to the table, and increment the particular entry pointed to by this bigram. The implementation described in [Gage 94] is based on a hash table that hashes each pair of bytes into a 12-bit number. That number is used as an index to an array of size $2^{12} = 4{,}096$ and the array location pointed to by the index is incremented. This works if the number of bigrams in the data file is less than 4,096 (notice that it can be up to $256 \times 256 = 65{,}536$).

Once the most-common bigram is located (`ab` in our example), it is replaced by an unused character (`h`). The file in the buffer becomes `hhchcd` and the pair-table is updated to

```
0 1 2 3 4 5 6 7
---------------
a b c d e f g a
1 1 1 1 0 0 0 b
```

The last entry indicates that byte-pair `ab` has been replaced by `h` (code 7).

The next (and last) pass identifies pair `hc` as the most common bigram and replaces it with unused symbol `g`. The file in the buffer becomes `hggd` and the pair-table is updated to

```
0 1 2 3 4 5 6 7
---------------
a b c d e f h a
1 1 1 1 0 0 c b
```

The 7th entry indicates that bigram `hc` has been replaced by `g` (code 6).

The pair-table consists of two types of entries. One type (type 1) has a binary flag in the bottom row indicating used or unused symbols (1 or 0 flags). This type is easy to identify because the element in the top row is identical to its index (thus, the first element `a` had code 0 and is in column 0, while the last element, also `a`, has code 0 but is in column 7). The other type (type 2) indicates pair substitutions, and entries of this type should be written on the compressed file. Notice that the two types of entries may be mixed and do not have to be contiguous. In our example, the pair-table consists of six type-1 entries followed by two type-2 entries and is written on the compressed file as the six bytes -6, `h`, `c`, `1`, `a`, `b`, where the first three bytes indicate six irrelevant type-1 entries (corresponding to codes 0 through 5) followed by one type-2 entry `hc` (corresponding to code 6). The next three bytes indicate one type-2 entry `ab` (which corresponds to code 7). The encoding rule for this table is therefore the following: Each contiguous segment of n type-1 entries followed by a type-2 entry is encoded as $-n$ followed by the two bytes

of the type-2 entry. Each segment of m consecutive type-2 entries is encoded as the byte m followed by the $2m$ bytes of the entries.

The last feature of the encoder has to do with replacing pairs of bytes in the buffer. When a pair of bytes `xy` is replaced by a single byte `p`, it (the pair) becomes `p-`, where the `-` indicates an empty byte. There may be several ways to handle empty bytes. The straightforward way is to move bytes and compact the buffer each time a pair is replaced by a single byte. This is extremely slow because it may involve many thousands of byte movements for each replacement. A better approach is to have an auxiliary array of flags that indicate which byte positions in the buffer are empty. If the size of the buffer is n bytes, the size of the auxiliary array should be n bits, one-eighth (or 12.5%) the buffer size. If the buffer contains the 16 bytes `the␣la-t␣fea-ure`, then the auxiliary array should have the two bytes 00000010|00001000, where the two 1's indicate empty bytes. When the compressed file is written from the buffer to the output, only those bytes that correspond to zero bits in the auxiliary array should be written. Another way to handle empty bytes is to organize the buffer as a linked list and establish another list (initially empty) of empty bytes. When a byte becomes empty, pointers are updated to take this byte out of the buffer and append it to the list of empty bytes.

It is now clear that encoding may not be very efficient, but on the other hand the method never expands the data (except for the overhead from the pair-table). If no byte values are unused, the data is simply written on the compressed file as is, with the addition of the pair-table. This should be compared to other, more efficient methods where the "wrong" type of data may cause significant expansion.

Encoding is slow, requiring multiple passes and a large buffer. Decoding, on the other hand, is fast. The decoder first reads and expands the pair-table, where only the type-2 entries are relevant. Bytes are then read from the compressed file. If a byte is literal (i.e., if it does not appear in the type-2 entries, such as byte `d` in our example), it is written to the final output as is. Otherwise, the byte is one of the type-2 entries and it represents a pair. The pair is constructed and is pushed into a stack. If the stack is not empty, the next byte is popped from the stack and handled as described above (which may cause another byte pair to be pushed into the stack). If the stack is empty, the next byte is read from the compressed file. Being so simple, the decoder is also very small, which is an advantage (as has been mentioned earlier).

Re-Pair, an efficient offline dictionary algorithm

The performance of BPE depends on the number of unused byte values in the original data. The next offline compression method (recursive pairing, or Re-Pair, by [Larsson and Moffat 00]) is much more sophisticated. It features high compression factors, a fast, small decoder, and it does not depend on the existence of unused byte values. It also employs sophisticated data structures that make it possible to select the most-common bigram in each pass without having to scan the entire input each time.

No unused byte values are required. In each pass, the most-common bigram is identified and is replaced by a *new* symbol. We first illustrate this idea symbolically, using the well-known line from the television film *Yabba-Dabba Do!* (based on the 1960–66 animated sitcom *The Flintstones*). The original data is shown in lowercase and the new symbols are in uppercase.

Pair	String
	yabba␣dabba,␣yabba␣dabba␣dabba␣do
A → ba	yabA␣dabA,␣yabA␣dabA␣dabA␣do
B → ab	yBA␣dBA,␣yBA␣dBA␣dBA␣do
C → BA	yC␣dC,␣yC␣dC␣dC␣do
D → C␣	yDdC,␣yDdDdDdo
E → Dd	yEC,␣yEEEo
F → yE	FC,␣FEEo

The compressed file consists of the final string FC,␣FEEo and the six-entry dictionary (or phrase-table). Notice that the final string contains one bigram, but replacing it with a new symbol would require an extra dictionary entry and would therefore contribute nothing to the compression (and may even cause expansion).

The main question is how to add new symbols. Assuming that the original data consists of bytes and that all 256 byte values are used, how can we add new symbols? We can start by placing each 8-bit byte in a pair of bytes (16 bits). Two-byte symbols can have values from 0 to $2^{16} - 1 = 65{,}535$, so there is room for the original 256 byte values and 65,280 new symbols. The bigram replacement algorithm is then executed and a phrase-table is constructed. Once this is done, the algorithm knows how many new symbols are needed, and the symbols (the 256 original ones and the new ones) are replaced with variable-length codes.

One way to assign reasonable variable-length codes to the symbols is to count the number of occurrences of each symbol in the final string and in the phrase-table and use this information to construct a set of Huffman codes. The table of Huffman codes becomes overhead that must be included in the compressed file for the decoder's use.

An alternative is to sort the list of symbols according to their frequencies of occurrence and assign them one of the many variable-length codes for the integers, such as the Golomb, Rice, or Elias codes. The most-common symbols are assigned the shortest codes and the overhead in this case is the sorting information. This information—which is a permutation of the symbols, each represented by its variable-length code—must be included in the compressed file for the decoder's use.

In either case, the compressed file consists of three parts, the final string (where each symbol is represented by its variable-length code), the phrase-table (which must also be compressed), and the overhead.

The Nakamura Murashima Method

The Nakamura Murashima method ([Nakamura and Murashima 91] and [Nakamura and Murashima 96]) also employs new symbols, but encodes the phrase-table differently. The authors propose two variants, dubbed SCT and SED. The former is straightforward. Given the 4-symbol alphabet a, b, c, and d and the source data string

acabbadcaddbaaddcaaddb, SCT encodes it as follows:

Pair	String
	acabbadcaddbaaddcaaddb
A → ad	acabbAcAdbaAdcaAdb
B → Ad	acabbAcBbaBcaBb
C → ca	aCbbAcBbaBCBb
D → Bb	aCbbAcDaBCD

SCT encodes the phrase-table in a simple way and prepends it to the output. It generates a string that includes two symbols for each table entry. The names of the new symbols are not included in this string and are assumed to be A, B, C, and so on. Thus, our phrase-table is encoded as the 8-symbol string adAdcaBb and the complete encoder output is the string adAdcaBb|aCbbAcDaBCD, where the vertical bar is a special separator symbol. The decoder reconstructs the phrase-table easily. It reads pairs of symbols, assigns the first pair to the new symbol A, the second pair to new symbol B, and so on until it reaches the separator.

The SED method is more complex. It constructs a different phrase-table and its output is a string of (original and new) symbols where flags distinguish between symbols and phrase-table entries. Each symbol in the output string is preceded by such a flag (a bit). A single symbol (either from the original alphabet or a new symbol) is preceded by a zero, while a phrase-table entry is preceded by a 1. Thus, the already-familiar input string acabbadcaddbaaddcaaddb becomes 0a10c0a0b0b10a0d0c110β0d0b0a0γ0α0δ (where Greek letters stand for the new symbols). Here is why. The first input symbol a becomes 0a, but the following pair ca appears several times, so it becomes 10c0a, where the "1" indicates a new symbol, α. The next pair bb appears only once, so it becomes 0b0b, but the following pair ad repeats several times, so it becomes 10a0d, where the "1" indicates the next new symbol β. The single c that follows becomes 0c. Notice that the encoder does not process the pair ca, because the c is followed by the pair ad, which has already been replaced by β. Thus, the next substring addb becomes βdb, and then γb (where the new symbol γ is add), and finally the new symbol $\delta = \gamma$b = βdb = addb. This is encoded as the string 1(10β0d)0b (without the paretheses). The rest of the encoding is easy to follow.

There remains the question of how to write a hybrid string (with bits and symbols mixed up) such as 0a10c0a0b0b10a0d0c110β0d0b0a0γ0α0δ on the output. The authors of this algorithm mention three different methods that employ adaptive arithmetic coding and complete binary trees, but no details are given.

I would like to acknowledge the help provided by Hirofumi Nakamura in the preparation of this section.

6.31 DCA, Compression with Antidictionaries

A dictionary-based compression method is based on a dictionary; a collection of bits and pieces of data found in the input. If the encoder locates the next input string a in the dictionary, it compresses a by emitting a pointer to its location in the dictionary. An antidictionary method is based on an inverse kind of knowledge. Such a method maintains an antidictionary with strings that *do not* appear in the input. Using the antidictionary, the encoder can often predict the next data symbol a with certainty, so that a doesn't have to be output. This is how such a method results in compression. The decoder can mimic the steps of the encoder and can therefore determine many data symbols and place them in the output. This section is based on [Crochemore et al. 00], who dubbed their approach DCA (data compression, antidictionaries). Reference [Davidson and Ilie 04] takes the basic idea further and describes algorithms for fast encoding and decoding.

> Antidictionary: This book would explain the disadvantages and hazards of using the word you looked up, and try to persuade you not to have anything to do with it. For example, under "solution," it could say, "A term widely used by apish technology-floggers to avoid the need to think of actual descriptive words for products and services." Imagine what it could say under "country music."
>
> http://www.halfbakery.com/idea/Anti-Dictionary#1096876321

An antidictionary method employs a binary alphabet where the basic data symbols are 0 and 1. We denote the input data (a bitstring) by w and assume that there exists an antidictionary AD with bitstrings that are not found in w (forbidden bitstrings). Notice that AD does not have to contain *every* forbidden bitstring. The encoder scans the input w bit by bit from left to right. We denote the part that has been scanned so far (the prefix of the input) by v. Assume that at a certain point, v is the bitstring 10110 and the next (still unknown) bit to be scanned is b. The encoder examines all the suffixes s of v which are 0, 10, 110, 0110, and 10110. If any of the strings $s0$ (i.e., a suffix s followed by a zero) is found in the antidictionary, then it is a forbidden string and the encoder knows that b must be 1. Similarly, if any of the strings $s1$ is in the AD, then b must be 0. In either case, b can be omitted from the output because the decoder can determine its value by searching the antidictionary in lockstep with the encoder. If neither $s0$ nor $s1$ is located in AD, then b is read by the encoder, is appended to v, and is also output because the decoder will not be able to determine it.

Given the input string $w = 10110101$, it is easy to construct the set $F(w)$ of substrings in w (we denote the empty string by ε)

$$F(w) = (\varepsilon, 0, 1, 01, 10, 11, 011, 101, 110, 0101, 0110, 1010, 1011, 1101, \ldots, 10110101).$$

An antidictionary AD for w is any set of bitstrings that are not in $F(w)$. Such bitstrings are forbidden in w. For example, $(00, 000, 001, 010, 100, 111, 0000, 0001)$ is such a set. Assuming that such an antidictionary exists, both encoder and decoder can easily be described and understood. Figure 6.47 is a pseudocode listing of the encoder.

```
ADC_Encoder(w, AD, γ)
1. v ← ε; γ ← ε;
2. for a ← first to last bit of w
3.   if for every suffix s of v, neither s0 nor s1 is in AD
4.       then γ ← γ.a; /* no compression */
5.   endif;
6.   v ← v.a;
7. endfor
8. return γ;
```

Figure 6.47: A Simple DCA Encoder.

As an example, we assume the input string $w = 10110101$ and the antidictionary $AD = (00, 000, 111, 11011)$. Table 6.48 lists the eight encoding steps, the values of γ and v at the end of each step, and the suffixes that have to be checked against AD at each step. Notice that the suffixes checked in each step are those of v of the previous step, because v is updated (in step 6) after the suffix check. Each suffix found in the AD (i.e., each forbidden string) is underlined, and it is obvious that each step with an underlined suffix leaves γ unchanged and thus increases compression (the difference in length between γ and v) by one bit. At the end of the loop, variable v equals the original input w, which suggests that there is really no need to have v as an independent variable. v is always a prefix of w and it is enough to maintain a pointer to w that will point to the current bit (the rightmost bit of v) at each step.

	a	γ	v	Suffix pairs $s0$, $s1$ to examine
Initial		ε	ε	
1.	1	1	1	$\varepsilon 0$, $\varepsilon 1$
2.	0	10	10	10, 11
3.	1	10	101	00, 01, 100, 101
4.	1	101	1011	10, 11, 010, 011, 1010, 1011
5.	0	101	10110	10, 11, 110, 111, 0110, 0111, 10110, 10111
6.	1	101	101101	00, 01, 100, 101, 1100, 1101, 01100, 01101, 101100, ...
7.	0	101	1011010	10, 11, 010, 011, 1010, 1011, 11010, 11011, 011010, ...
8.	1	101	10110101	00, 01, 100, 101, 0100, 0101, 10100, 10101, 110100, ...

Table 6.48: DCA Encoding of $w = 10110101$.

The compressed stream that is output by the encoder must consist of (1) the AD, (2) the length n of w, and (3) bitstring γ. Notice that n is needed by the decoder because different input strings w may produce the same γ when processed by the encoder of Figure 6.47 (this is illustrated by the decoding listing of Table 6.50). The antidictionary must also be included in the encoder's output, because it is static and has to be used by the decoder. (Notice that the entire input string w must be input in order to construct the antidictionary, which implies that DCA is a two-pass method.) Naturally, including

AD in the compressed stream reduces the effectiveness of this method, and suggests that an algorithm to construct an effective AD should be a crucial part of any practical DCA implementation. Alternatively, a dynamic approach is also possible, where the AD starts empty and is populated by forbidden strings as the encoder (and in lockstep, also the decoder) scans more and more input bits. At the very least, the AD should be compressed before it is written on the output. Notice that an AD is specific to the input data, which is why a general AD, based on training documents, cannot be used.

The DCA decoder is simple. It receives an AD, the length n of the original data w, and the compressed string γ. The decoder loops n times. In each iteration, it has a prefix v of w from previous iterations. It examines all the suffixes s of v. If a bitstring $s0$ is in the AD, then the next reconstructed bit must be a 1. Similarly, if $s1$ is found in the antidictionary, then the next bit must be 0. The next bit is then appended to v. (The notation $|v|$ denotes the length of string v, while $\overline{b}$ denotes the complement of bit b.) If neither $s0$ nor $s1$ is in the antidictionary, the next bit of γ is appended to v. In either case, v grows by one bit in the iteration. Figure 6.49 is a pseudocode listing of this simple decoder.

```
ADC_Decoder(AD, n, γ, v)
1. v ← ε;
2. while |v| < n
3.    if for a suffix s of v and a bit b, string s.b is in AD
4.      then v ← v.b̄;
5.      else v ← v.(next bit of γ);
6.    endif;
7. endwhile;
8. return v;
```

Figure 6.49: A Simple DCA Decoder.

Table 6.50 lists the steps performed by this decoder when it is given the same antidictionary and is asked to perform eight iterations with $\gamma = 101$ (when a bit of γ is used, it is underlined). Notice that the last (least-significant) bit of γ is used in iteration 4, which is why the decoder has to be given the length of the final, reconstructed bitstring.

Here are some observations regarding antidictionary design. If a substring s of the input w is forbidden, but a proper substring u of s is also forbidden, then u but not s should be included in the antidictionary. In general, only *minimal* forbidden strings (strings where no substring is also forbidden) should appear in an antidictionary. In our examples above, the antidictionary contains 000 but this string is never used because it is not minimal. Also, most matches between suffixes $s0$, $s1$ and antidictionary strings occur for short suffixes s. Thus, it is more important to include short forbidden strings in an AD, which keeps it short. In practice, the length of an input string may be many thousands of bits, whereas the length of the antidictionary may be a few hundred bits. Still, it is important to construct an antidictionary by an efficient algorithm and also to look for ways to compress the AD.

	v	$\lvert v\rvert$	γ	Suffix pairs $s0$, $s1$ to examine
1.	$\varepsilon \to 1$	0	101	none
2.	$1 \to 10$	1	101	10, 11
3.	$10 \to 101$	2		00, 01, 100, 101
4.	$101 \to 1011$	3	101	10, 11, 010, 011, 1010, 1011
5.	$1011 \to 10110$	4		10, 11, 110, 111, 0110, 0111, 10110, 10111
6.	$10110 \to 101101$	5		00, 01, 100, 101, 1100, 1101, 01100, 01101, ...
7.	$101101 \to 1011010$	6		10, 11, 010, 011, 1010, 1011, 11010, 11011, ...
8.	$1011010 \to 10110101$	7		00, 01, 100, 101, 0100, 0101, 10100, 10101, ...

Table 6.50: DCA Decoding of $\gamma = 101$ and $n = 8$.

The main reference of DCA is [Crochemore et al. 00] and it discusses ways to prune the antidictionary and to compress it. The developers observe the following. Given an antidictionary AD with only minimal forbidden strings, if we remove a string v from AD, the remaining strings constitute an antidictionary for v. Thus, an antidictionary can be compressed by removing each of its strings in turn and then using the remaining antidictionary to compress the string; a process that can be termed self compression.

The developers of this method also describe an algorithm to construct an AD. Unfortunately, this algorithm is complex, it is based on finite-state automata, and its construction is beyond the scope of this book. Instead, we show an example (after [Davidson and Ilie 04]) illustrating how the automaton that corresponds to an antidictionary for a given bitstring w can be used to create two state diagrams (also referred to as transducers) that simplify the encoding and decoding of w.

Given the input string $w = 11010001$, the AD-creation algorithm (not described here) produces the automaton of Figure 6.51a. Each state in such an automaton has two outgoing transitions, for 0 and 1. If one of the transitions of state A leads to a non-final state (a square), then A is referred to as a predict state and is shown in gray in the figure. Once this automaton has been obtained, it is used to create the antidictionary $AD = (0000, 111, 011, 0101, 1100)$ in the following way. We start at state 0 and work our way to one of the non-final states (shown as squares), collecting bits off the edges on our way and concatenating them to form a bitstring. When we get to a non-final state, the bitstring is added to AD as the next forbidden string. Once we have visited all the non-final states, the AD is complete.

With the AD in hand, the same automaton is used to construct a state diagram (a transducer) for encoding $w = 11010001$ (Figure 6.51b). This transducer has the following property: For each outgoing transition from a non-predict state (a white circle), the output equals the input. Thus, in our example, we enter state 0 with an input of 1 (because w starts with 1), so we have to leave it with the same output, which takes us to state 2. The input bit at this point is the second 1 of w, so we leave state 2 with an identical output and find ourselves in state 5. The next input bit is 0, so the output is ε and the transducer sends us to state 9. The entire encoding sequence can be summarized by

$$0 \overset{1/1}{\to} 2 \overset{1/1}{\to} 5 \overset{0/\varepsilon}{\to} 9 \overset{1/\varepsilon}{\to} 4 \overset{0/\varepsilon}{\to} 7 \overset{0/\varepsilon}{\to} 3 \overset{0/0}{\to} 6 \overset{1/\varepsilon}{\to} 4$$

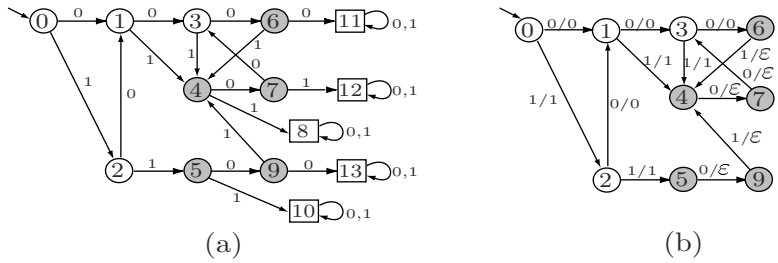


Figure 6.51: Automaton and Transducer for Fast Encoding and Decoding.

and the encoding produces (in states 0, 2, and 3) the output string 110.

Notice that state 5 has only one output transition (corresponding to an input of 0). The reason is that we can arrive at this state only after scanning the substring 11 from the input, and the AD tells us that this substring cannot be followed by a 0.

The same transducer is used for decoding. The only change needed is to swap the input and output bits on each transition edge. Thus, for example, the transition from state 5 to state 9 will now be labeled $\varepsilon/0$. Decoding string 111 is done in eight steps (notice that the decoder requires the length of w) as follows

$$0 \stackrel{1/1}{\to} 2 \stackrel{1/1}{\to} 5 \stackrel{\varepsilon/0}{\to} 9 \stackrel{\varepsilon/1}{\to} 4 \stackrel{\varepsilon/0}{\to} 7 \stackrel{\varepsilon/0}{\to} 3 \stackrel{0/0}{\to} 6 \stackrel{\varepsilon/1}{\to} 4.$$

It seems that the idea of using antidictionaries for compression has caught the attention of many researchers. Here are just two interesting contributions to this area. Reference [Crochemore and Navarro 02] introduces the notion of almost antifactors, which are strings that rarely appear in the text. More precisely, almost antifactors are strings that improve compression if they are considered forbidden and are added to the antidictionary. In [Ota and Morita 07], the authors present an algorithm (linear in space and time) to construct an antidictionary from a suffix tree.

6.32 CRC

The idea of a parity bit is simple, old, and familiar to most computer practitioners. A parity bit is the simplest type of error detecting code. It adds reliability to a group of bits by making it possible for hardware to detect certain errors that may occur when the group is stored in memory, is written on a disk, or is transmitted over communication lines between computers. A single parity bit does not make the group completely reliable. There are certain errors that cannot be detected with a parity bit, but experience shows that even a single parity bit can make data transmission reliable in most practical cases.

The parity bit is computed from a group of $n-1$ bits, then added to the group, making it n bits long. A common example is a 7-bit ASCII code that becomes 8 bits long after a parity bit is added. The parity bit p is computed by counting the number of 1's in the original group, and setting p to complete that number to either odd or even. The former is called odd parity, and the latter is called even parity.

Examples: Given the group of 7 bits 1010111, the number of 1's is five, which is odd. Assuming odd parity, the value of p should be 0, leaving the total number of 1's odd. Similarly, the group 1010101 has four 1's, so its odd parity bit should also be a 1, bringing the total number of 1's to five.

Imagine a block of data where the most significant bit (MSB) of each byte is an odd parity bit, and the bytes are written vertically (Table 6.52a).

(a)	(b)	(c)	(d)
1 01101001	1 01101001	1 01101001	1 01101001
0 00001011	0 00001011	0 00001011	0 00001011
0 11110010	0 11**0**10010	0 11**0**10**1**10	0 11**0**10**1**10
0 01101110	0 01101110	0 01101110	0 01101110
1 11101101	1 11101101	1 11101101	1 11101101
1 01001110	1 01001110	1 01001110	1 01001110
0 11101001	0 11101001	0 11101001	0 11101001
1 11010111	1 11010111	1 11010111	1 11010111
			0 00011100

Table 6.52: Horizontal and Vertical Parities.

When this block is read from a disk or is received by a computer, it may contain transmission errors, errors that have been caused by imperfect hardware or by electrical interference during transmission. We can think of the parity bits as *horizontal reliability*. When the block is read, the hardware can check every byte, verifying the parity. This is done by simply counting the number of 1's in the byte. If this number is odd, the hardware assumes that the byte is good. This assumption is not always correct, since two bits may get corrupted during transmission (Table 6.52c). A single parity bit is therefore useful (Table 6.52b) but does not provide full error detection capability.

A simple way to increase the reliability of a block of data is to compute vertical parities. The block is considered to be eight vertical columns, and an odd parity bit is computed for each column (Table 6.52d). If two bits in a byte go bad, the horizontal parity will not catch it, but two of the vertical ones will. Even the vertical bits do not provide complete error detection capability, but they are a simple way to significantly improve data reliability.

A CRC is a glorified vertical parity. CRC stands for Cyclical Redundancy Check (or Cyclical Redundancy Code) and is a rule that shows how to compute the vertical check bits (they are now called check bits, not just simple parity bits) from all the bits of the data. Here is how CRC-32 (one of the many standards developed by the CCITT) is computed. The block of data is written as one long binary number. In our example this will be the 64-bit number

101101001|000001011|011110010|001101110|111101101|101001110|011101001|111010111.

The individual bits are considered the coefficients of a *polynomial* (see below for definition). In our example, this will be the degree-63 polynomial

$$\begin{aligned} P(x) &= 1\times x^{63}+0\times x^{62}+1\times x^{61}+1\times x^{60}+\cdots+1\times x^2+1\times x^1+1\times x^0 \\ &= x^{63}+x^{61}+x^{60}+\cdots+x^2+x+1. \end{aligned}$$

This polynomial is then divided by the standard CRC-32 *generating polynomial*

$$\mathrm{CRC}_{32}(x)=x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x^1+1.$$

When an integer M is divided by an integer N, the result is a quotient Q (which we will ignore) and a remainder R, which is in the interval $[0, N-1]$. Similarly, when a polynomial $P(x)$ is divided by a degree-32 polynomial, the result is two polynomials, a quotient and a remainder. The remainder is a degree-31 polynomial, which means that it has 32 coefficients, each a single bit. Those 32 bits are the CRC-32 code, which is appended to the block of data as four bytes. As an example, the CRC-32 of a recent version of the file with the text of this chapter is $\texttt{586DE4FE}_{16}$.

Selecting a generating polynomial is more an art than science. Page 196 of [Tanenbaum 02] is one of a few places where the interested reader can find a clear discussion of this topic.

The CRC is sometimes called the "fingerprint" of the file. Of course, since it is a 32-bit number, there are only 2^{32} different CRCs. This number equals approximately 4.3 billion, so, in theory, there may be different files with the same CRC, but in practice this is rare. The CRC is useful as an error detecting-code because it has the following properties:

1. Every bit in the data block is used to compute the CRC. This means that changing even one bit may produce a different CRC.
2. Even small changes in the data normally produce very different CRCs. Experience with CRC-32 shows that it is very rare that introducing errors in the data does not modify the CRC.
3. Any histogram of CRC-32 values for different data blocks is flat (or very close to flat). For a given data block, the probability of any of the 2^{32} possible CRCs being produced is practically the same.

Other common generating polynomials are $\mathrm{CRC}_{12}(x)=x^{12}+x^3+x+1$ and $\mathrm{CRC}_{16}(x)=x^{16}+x^{15}+x^2+1$. They generate the common CRC-12 and CRC-16 codes, which are 12 and 16 bits long, respectively.

Definition: A polynomial of degree n in x is the function

$$P_n(x)=\sum_{i=0}^{n}a_ix^i=a_0+a_1x+a_2x^2+\cdots+a_nx^n,$$

where a_i are the $n+1$ coefficients (in our case, real numbers).

Two simple, readable references on CRC are [Ramabadran and Gaitonde 88] and [Williams 93].

6.33 Summary

The dictionary-based methods presented here are different but are based on the same principle. They read the input stream symbol by symbol and add phrases to the dictionary. The phrases are symbols or strings of symbols from the input. The main difference between the methods is in deciding what phrases to add to the dictionary. When a string in the input stream matches a dictionary phrase, the encoder outputs the position of the match in the dictionary. If that position requires fewer bits than the matched string, compression results.

In general, dictionary-based methods, when carefully implemented, give better compression than statistical methods. This is why many popular compression programs are dictionary based or employ a dictionary as one of several compression steps.

6.34 Data Compression Patents

It is generally agreed that an invention or a process is patentable but a mathematical concept, calculation, or proof is not. An algorithm seems to be an abstract mathematical concept that should not be patentable. However, once the algorithm is implemented in software (or in firmware) it may not be possible to separate the algorithm from its implementation. Once the implementation is used in a new product (i.e., an invention), that product—including the implementation (software or firmware) and the algorithm behind it—may be patentable. [Zalta 88] is a general discussion of algorithm patentability. Several common data compression algorithms, most notably LZW, have been patented; and the LZW patent is discussed here in some detail.

The Sperry Corporation was granted a patent (4,558,302) on LZW in December 1985 (even though the inventor, Terry Welch, left Sperry prior to that date). When Unisys acquired Sperry in 1986 it became the owner of this patent and required users to obtain (and pay for) a license to use it.

When CompuServe designed the GIF format in 1987 it decided to use LZW as the compression method for GIF files. It seems that CompuServe was not aware at that point that LZW was patented, nor was Unisys aware that the GIF format uses LZW. After 1987 many software developers became attracted to GIF and published programs to create and display images in this format. GIF became widely accepted and was commonly used on the World-Wide Web, where it was one of the prime image formats for Web pages and browsers.

It was not until GIF had become a world-wide de facto standard that Unisys contacted CompuServe for a license. Naturally, CompuServe and other LZW users tried to challenge the patent. They applied to the United States Patent Office for a reexamination of the LZW patent, with the (perhaps surprising) result that on January 4, 1994, the patent was reconfirmed and CompuServe had to obtain a license (for an undisclosed sum) from Unisys later that year. Other important licensees of LZW (see [Rodriguez 95]) were Aldus (in 1991, for the TIFF graphics file format), Adobe (in 1990, for PostScript level II), and America Online and Prodigy (in 1995).

The Unisys LZW patent had significant implications for the World-Wide Web, where use of GIF format images was currently widespread. Similarly, the Unix `compress`

utility uses LZW and therefore required a license. In the United States, the patent expired on 20 June 2003 (20 years from the date of first filing). In Europe (patent EP0129439) expired on 18 June 2004. In Japan, patents 2,123,602 and 2,610,084 expired on 20 June 2004, and in Canada, patent CA1223965 expired on 7 July 2004.

Unisys graciously exempted old software products (those written or modified before January 1, 1995) from a patent license. Also exempt was any noncommercial and nonprofit software, old and new. Commercial software (even shareware) or firmware created after December 31, 1994, had to be licensed if it supported the GIF format or implemented LZW. A similar policy was enforced with regard to TIFF, where the cutoff date is July 1, 1995. Notice that computer users could legally keep and transfer GIF and any other files compressed with LZW; only the compression/decompression software required a license.

For more information on the Unisys LZW patent and license see [unisys 03].

An alternative to GIF is the Portable Network Graphics, PNG (pronounced "ping," Section 6.27) graphics file format [Crocker 95], which was developed expressly to replace GIF, and avoid patent claims. PNG is simple, portable, with source code freely available, and is unencumbered by patent licenses. It has potential and promise in replacing GIF. However, any GIF-to-PNG conversion software required a Unisys license.

The GNU `gzip` compression software (Section 6.14) should also be mentioned here as a popular substitute for `compress`, since it is free from patent claims, is faster, and provides superior compression.

The LZW U.S. patent number is 4,558,302, issued on Dec. 10, 1985. Here is the abstract filed as part of it (the entire filing constitutes 50 pages).

> A data compressor compresses an input stream of data character signals by storing in a string table strings of data character signals encountered in the input stream. The compressor searches the input stream to determine the longest match to a stored string. Each stored string comprises a prefix string and an extension character where the extension character is the last character in the string and the prefix string comprises all but the extension character. Each string has a code signal associated therewith and a string is stored in the string table by, at least implicitly, storing the code signal for the string, the code signal for the string prefix and the extension character. When the longest match between the input data character stream and the stored strings is determined, the code signal for the longest match is transmitted as the compressed code signal for the encountered string of characters and an extension string is stored in the string table. The prefix of the extended string is the longest match and the extension character of the extended string is the next input data character signal following the longest match. Searching through the string table and entering extended strings therein is effected by a limited search hashing procedure. Decompression is effected by a decompressor that receives the compressed code signals and generates a string table similar to that constructed by the compressor to effect lookup of received code signals so as to recover the data character signals comprising a stored string. The decompressor string table is updated by storing a string having a prefix in accordance with a prior received code signal and an extension character in accordance with the first character of the currently recovered string.

Here are a few other patented compression methods, some of them mentioned elsewhere in this book:
1. "Textual Substitution Data Compression with Finite Length Search Windows," U.S. Patent 4,906,991, issued March 6, 1990 (the LZFG method).
2. "Search Tree Data Structure Encoding for Textual Substitution Data Compression Systems," U.S. Patent 5,058,144, issued Oct. 15, 1991.

The two patents above were issued to Edward Fiala and Daniel Greene.
3. "Apparatus and Method for Compressing Data Signals and Restoring the Compressed Data Signals." This is the LZ78 patent, assigned to Sperry Corporation by the inventors Willard L. Eastman, Abraham Lempel, Jacob Ziv, and Martin Cohn. U.S. Patent 4,464,650, issued August, 1984.

The following, from Ross Williams `http://www.ross.net/compression/` illustrates how thorny this issue of patents is.

> Then, just when I thought all hope was gone, along came some software patents that drove a stake through the heart of the LZRW algorithms by rendering them unusable. At last I was cured! I gave up compression and embarked on a new life, leaving behind the world of data compression forever.

Appropriately, Dr Williams maintains a list [patents 06] of data compression-related patents.

> Your patent application will be denied. Your permits will be delayed. Something will force you to see reason-and to sell your drug at a lower cost.
> Wu had heard the argument before. And he knew Hammond was right, some new bioengineered pharmaceuticals had indeed suffered inexplicable delays and patent problems.
> You don't even know exactly what you have done, but already you have reported it, patented it, and sold it.
>
> —Michael Crichton, *Jurassic Park* (1991)

6.35 A Unification

Dictionary-based methods and methods based on prediction approach the problem of data compression from two different directions. Any method based on prediction predicts (i.e., assigns probability to) the current symbol based on its order-N context (the N symbols preceding it). Such a method normally stores many contexts of different sizes in a data structure and has to deal with frequency counts, probabilities, and probability ranges. It then uses arithmetic coding to encode the entire input stream as one large number. A dictionary-based method, on the other hand, works differently. It identifies the next phrase in the input stream, stores it in its dictionary, assigns it a code, and continues with the next phrase. Both approaches can be used to compress data because each obeys the general law of data compression, namely, to assign short codes to common events (symbols or phrases) and long codes to rare events.

On the surface, the two approaches are completely different. A predictor deals with probabilities, so it can be highly efficient. At the same time, it can be expected to

be slow, since it deals with individual symbols. A dictionary-based method deals with strings of symbols (phrases), so it gobbles up the input stream faster, but it ignores correlations between phrases, typically resulting in poorer compression.

The two approaches are similar because a dictionary-based method *does use* contexts and probabilities (although implicitly) just by storing phrases in its dictionary and searching it. The following discussion uses the LZW trie to illustrate this concept, but the argument is valid for any dictionary-based method, no matter what the details of its algorithm and its dictionary data structure.

Imagine the phrase `abcdef...` stored in an LZW trie (Figure 6.53a). We can think of the substring `abcd` as the order-4 context of `e`. When the encoder finds another occurrence of `abcde...` in the input stream, it will locate our phrase in the dictionary, parse it symbol by symbol starting at the root, get to node `e`, and continue from there, trying to match more symbols. Eventually, the encoder will get to a leaf, where it will add another symbol and allocate another code. We can think of this process as adding a new leaf to the subtree whose root is the `e` of `abcde...`. Every time the string `abcde` becomes the prefix of a parse, both its subtree and its code space (the number of codes associated with it) get bigger by 1. It therefore makes sense to assign node `e` a probability depending on the size of its code space, and the above discussion shows that the size of the code space of node `e` (or, equivalently, string `abcde`) can be measured by counting the number of nodes of the subtree whose root is `e`. This is how probabilities can be assigned to nodes in any dictionary tree.

The ideas of Glen Langdon in the early 1980s (see [Langdon 83] but notice that his equation (8) is wrong; it should read $P(y|s) = c(s)/c(s \cdot y)$; [Langdon 84] is perhaps more useful) led to a simple way of associating probabilities not just to nodes but also to edges in a dictionary tree. Assigning probabilities to edges is more useful, since the edge from node `e` to node `f`, for example, signifies an `f` whose context is `abcde`. The probability of this edge is thus the probability that an `f` will follow `abcde` in the input stream. The fact that these probabilities can be calculated in a dictionary tree shows that every dictionary-based data compression algorithm can be "simulated" by a prediction algorithm (but notice that the converse is not true). Algorithms based on prediction are, in this sense, more general, but the important fact is that these two seemingly different classes of compression methods can be unified by the observations listed here.

The process whereby a dictionary encoder slides down from the root of its dictionary tree, parsing a string of symbols, can now be given a different interpretation. We can visualize it as a sequence of making predictions for individual symbols, computing codes for them, and combining the codes into one longer code, which is eventually written on the compressed stream. It is as if the code generated by a dictionary encoder for a phrase is actually made up of small chunks, each a code for one symbol.

The rule for calculating the probability of the edge $e \to f$ is to count the number of nodes in the subtree whose root is `f` (including node `f` itself) and divide by the number of nodes in the subtree of `e`. Figure 6.53b shows a typical dictionary tree with the strings `aab`, `baba`, `babc`, and `cac`. The probabilities associated with every edge are also shown and should be easy for the reader to verify. Note that the probabilities of sibling subtrees don't add up to 1. The probabilities of the three subtrees of the root, for example, add up to 11/12. The remaining 1/12 is assigned to the root itself and

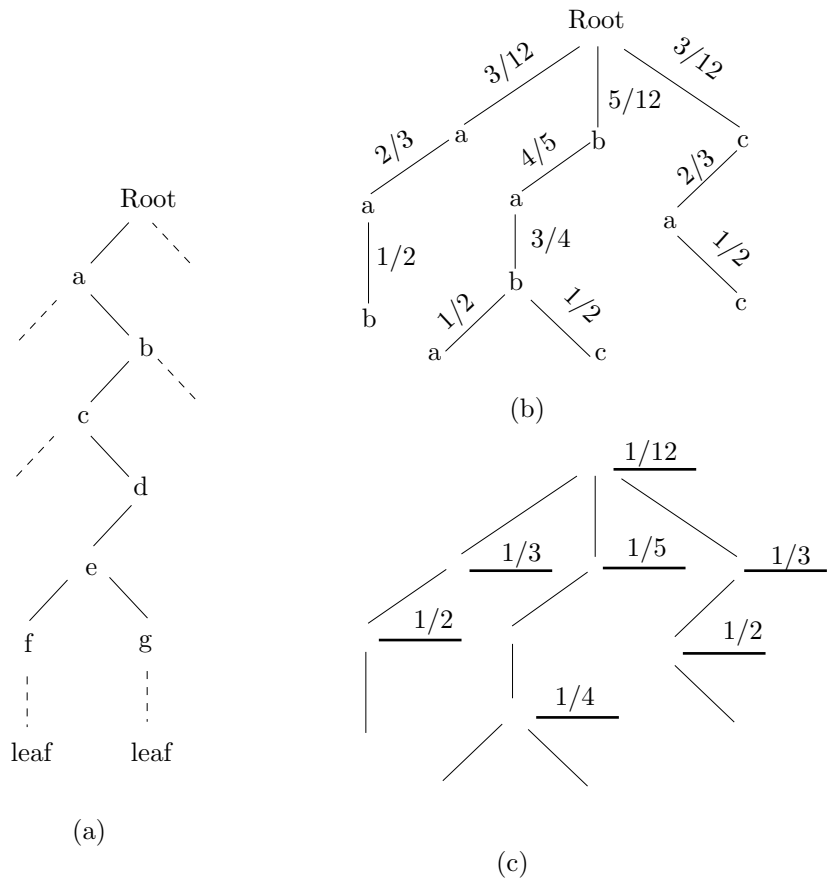


Figure 6.53: Defining Probabilities in a Dictionary Tree.

represents the probability that a fourth string will eventually start at the root. These "missing probabilities" are shown as horizontal lines in Figure 6.53c.

The two approaches, dictionary and prediction, can be combined in a single compression method. The LZP method of Section 6.19 is one example; the LZRW4 method (Section 6.12) is another. These methods work by considering the context of a symbol before searching the dictionary.

The only place where housework comes
before needlework is in the dictionary.
—Mary Kurtz