3
Dictionary Methods

Prelude

The Huffman algorithm is based on the probabilities of the individual data symbols. These probabilities become a statistical model of the data. As a result, the compression produced by this method depends on how good that model is. Dictionary-based compression methods are different. They do not use a statistical model of the data, nor do they employ variable-length codes. Instead they select strings of symbols from the input and employ a dictionary to encode each string as a *token*. The dictionary holds strings of symbols, and it may be static or dynamic (adaptive). The former is permanent, sometimes allowing the addition of strings but no deletions, whereas the latter holds strings previously found in the input, 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 and the dictionary is searched. If a match is found, an index to the dictionary is written on the output. Otherwise, the uncompressed word itself is written.

As a result, the output contains indexes and raw words, and it is important to distinguish between them. This can be done by reserving an extra bit in each item

written on the output. 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 on the output, followed by the size of the unmatched word, followed by the word itself.

Example: Assuming that the word `bet` is found in dictionary entry 1025, it is encoded as the 20-bit number 0|00000000010000000001. 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. Compressing 48 bits into 20 bits is excellent if 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 will be $N[20P + 48(1 - P)] = N[48 - 28P]$ bits. The size of the input 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 3.1:** (1) What compression factor do we get with $P = 0.9$? (2) What is the maximum compression possible with this method?

As long as the input consists of English text, most words will be found in a 500,000-word dictionary. Other types of data, however, may not do as well. A file with 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 (Figure 3.1), so very few matches may be found, resulting in considerable expansion instead of compression.

Figure 3.1: An Image and Corresponding Text.

Thus, a static dictionary is not a good choice for a general-purpose compressor. It may, however, be a good choice for a special-purpose dictionary-based encoder. Consider, for example, a chain of hardware stores. 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, 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 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. Otherwise, the uncompressed word is output 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). It reads the next input item and determines whether it is a token or raw data. A token is used to obtain data from the dictionary and write it on the output. Raw data is output as is. The decoder does not have to parse the input in a complex way, nor does it have to search the dictionary to find matches. Many programmers like that, too.

> 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)

3.1 LZ78

The LZ78 method (sometimes also referred to as LZ2) [Ziv and Lempel 78] does not employ 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, because 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 file. 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 rapidly and to fill up the entire available memory).

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, the dictionary is searched for an entry with the one-symbol string `x`. If none is 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 is 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 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 is 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 3.2 lists the first 14 steps in encoding the string `sir␣sid␣eastman␣easily␣teases␣sea␣sick␣seals`.

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 3.2: First 14 Encoding Steps in LZ78.

◇ **Exercise 3.2:** Complete Table 3.2.

In each step, the string added to the dictionary is the one that is being encoded, minus 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 (which implies longer 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 example above 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 3.3. For example, all the strings that start with `s` (the four strings `si`, `sil`, `st`, and `s(eof)`) constitute the subtree of node `s`.

Assuming an alphabet of 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 therefore 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. Recall that no nodes are ever deleted, so 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

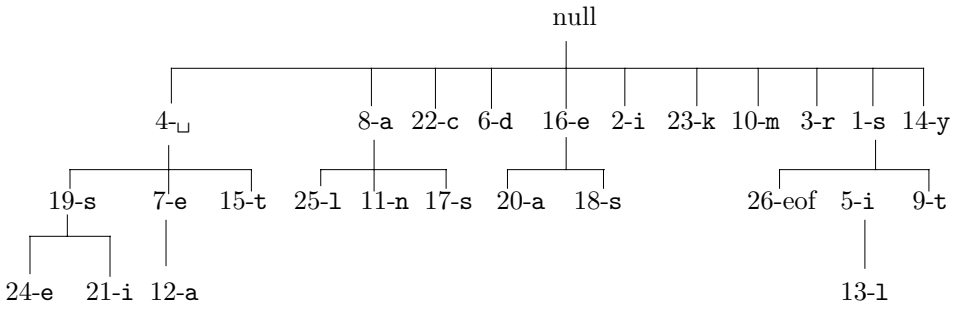


Figure 3.3: An LZ78 Dictionary Tree.

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 occurs. 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* (pronounced try). 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. In the case of LZ78, each string added to the tree effectively adds just one symbol, and does that by adding a branch.

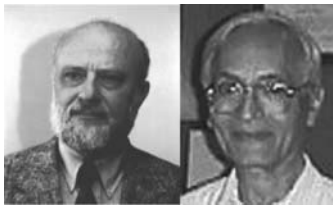
Since the total size of the tree is limited, it may fill up during compression. This, in fact, happens all the time except when the input is unusually small. The original LZ78 method does not specify what to do in such a case, so we list a few possible solutions.

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 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 simple, fast algorithm to decide which entries to delete, and how many.

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.

Intermezzo

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 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. More than a dozen such methods are described in detail in [Salomon 07]. Of special interest is the popular LZW algorithm, partly devised by Terry Welch (Section 3.2), which has extended LZ78 and made it useful in practical applications.



Abraham Lempel and Jacob Ziv.

Gamblers like the phrase "heads I win, tails I lose" (if you hear this, make sure you did not hear "heads I win, tails you lose"). Mr G. Ambler, a veteran gambler, decided to try a simple scheme, one where even he could easily figure out his winnings and losses. The principle is to always gamble half the money he has in his pocket. Thus, if he starts with an amount a and wins, he ends up with $1.5a$. If next he loses, he pays out half that and is left with $0.75a$. Assuming that he plays g games and wins half the time, what is his chance of making a net profit?

3.2 LZW

LZW 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 was patented and for many years its use required a license. This issue is discussed in [Salomon 07] as well as in many places on the Internet.)

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, appending 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 3.4 summarizes all the steps of this process. Table 3.5 shows some of the original 256 entries in the LZW dictionary plus the entries added during encoding of the string above. The complete output file 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 3.6 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 64 K-entry dictionary (where $64\,\mathrm{K} = 2^{16} = 65{,}536$). Naturally, such a dictionary will 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 with a chance to achieve really good compression. We say that LZW adapts slowly to its input data.

◇ **Exercise 3.3:** Use LZW to encode the string `alf␣eats␣alfalfa`. Show the encoder output and the new entries appended to the dictionary.

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 3.4: 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 3.5: 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 3.6: The LZW Algorithm.

◇ **Exercise 3.4:** Analyze the LZW compression of the string `aaaa...`.

`A dirty icon` (anagram of "dictionary")

3.2.1 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. 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 (which consists of pointers to the dictionary) and uses each pointer to retrieve uncompressed symbols from its dictionary and write them on its output. It also builds its dictionary in 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 3.5:** Decode the string `alf␣eats␣alfalfa` by using the encoding results from Exercise 3.3.

◇ **Exercise 3.6:** Assume a two-symbol alphabet with the symbols `a` and `b`. Show the first few steps for encoding and decoding the string "`ababab...`".

3.2.2 LZW Dictionary Structure

Up until now, we have assumed that the LZW dictionary is an array of variable-size strings. It turns out that a trie is an ideal data structure for a practical implementation of such a dictionary. The first step in understanding such an implementation is 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 receives one child. Later, when, say, the string `ae` is added, node 97 receives 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:

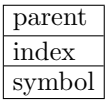
Node				
Address :	97	266	284	299
Contents :	(-:a)	(97:b)	(266:c)	(284:d)
Represents:	a	ab	abc	abcd

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`.

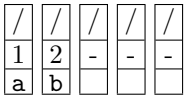
3. The node contains something else. This means that another hashing of a pointer and an ASCII code has resulted in 299, and node 299 already contains information from another string. This is called a *collision*, and it can be dealt with in several ways. The simplest way to deal with a collision is 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 (nor-

mally ASCII) of the symbol contained in the node. The second field is necessary because of collisions. A node can therefore be illustrated by



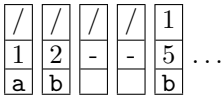
We illustrate this data structure using string `ababab...` of Exercise 3.6. The dictionary is an array `dict` where each entry is a structure with the three fields `parent`, `index`, and `symbol`. We refer to a field by, for example, `dict[pointer].parent`, where `pointer` is an index to the array. The 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;
```

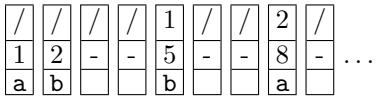
with `pointer` = 5. `J` is moved into `I`, so `I` = 2.



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.



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 3.3. 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 3.7.
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 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.

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 3.7: Growing an LZW Trie for alf␣eats␣alfalfa.

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. 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.

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 file. When this file 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 common 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 3.7:** What is the longest string that can be retrieved from the LZW dictionary during decoding?

 Big Amy lives in London and works in a store on Oxford Street. In order to justify her name, she is married to two men, neither of whom knows about the other. Thus, she has to juggle her marital life carefully. Every day after work, she walks to the Oxford

Circus underground station and takes either the Victoria line (cyan) south, to Brixton, where one husband lives, or the Bakerloo line (brown) north, to Maida Vale, where the other husband lives. Being a loving, impartial, and also a careful wife, she tries to balance her visits so as not to prefer any husband over the other. To this end she gets off work and arrives at the underground station at random times. In spite of this, she finds herself at Maida Vale much more often than at Brixton. What could be a reason for such imbalance?

3.3 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) and MNG (Multiple-Image Network Graphics) graphics file formats ([PNG 03] and [MNG 03]), and (4) Adobe's PDF (Portable Document File) [PDF 01].

Deflate was developed 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].

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.

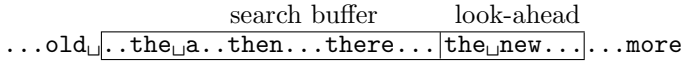
Always a loner, Katz suffered from personal and legal problems, started drinking heavily, and died on April 14, 2000 from complications related to chronic alcoholism. He was 37 years old.

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.

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 1.3.1) tries to match the text in the look-ahead buffer to strings already in the search buffer. In the example

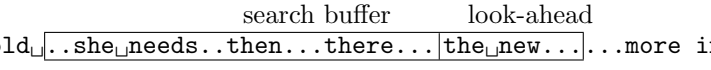


the look-ahead buffer starts with the string `the␣`, which can be matched to one of three strings in the search buffer. The longest match has a length of 4. LZ77 writes tokens on the compressed file, where each token is a triplet (offset, length, next symbol). The third component is needed in cases where 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 file. When no match is found, the unmatched character is written on the compressed file instead of a token. Thus, the compressed data 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 file 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 32 Kbytes.

◇ **Exercise 3.8:** When no match is found, Deflate writes the unmatched character (in raw format) on the compressed file instead of a token. Suggest an alternative.

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 file. 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 file. 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 archive an 8 Gb file 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 file 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 file. 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 file. Distances are located in the second table and are replaced by special prefix codes that are output to the compressed file. A block written on the compressed file 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 output, and they are written in compressed format. A block output by the encoder 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.

 What is the next integer in the sequence (12, 6), (6, 3), (10, ?)?

3.3.1 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 file.

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.

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 3.8: 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 3.9: Huffman Codes for Edocs in Mode 2.

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 file, 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. 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 3.8 lists the 29 edocs, the extra bits, and the lengths that they represent. What is actually written on the output is prefix codes of the edocs (Table 3.9). Notice that edocs 286 and 287 are never created, so their prefix codes are never used. We show later that Table 3.9 can be represented by the sequence of code lengths

$$\underbrace{8,8,\ldots,8}_{144},\underbrace{9,9,\ldots,9}_{112},\underbrace{7,7,\ldots,7}_{24},\underbrace{8,8,\ldots,8}_{8}, \tag{3.1}$$

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 3.10. 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.

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 3.10: Thirty Prefix Distance Codes in Mode 2.

3.3.2 Format of Mode-3 Blocks

In mode 3, the encoder generates two prefix code tables, one for the literals/lengths and 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 output, 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 2.1 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 2.2.6.) 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 3.11a. 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 3.11b. 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 output.

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.

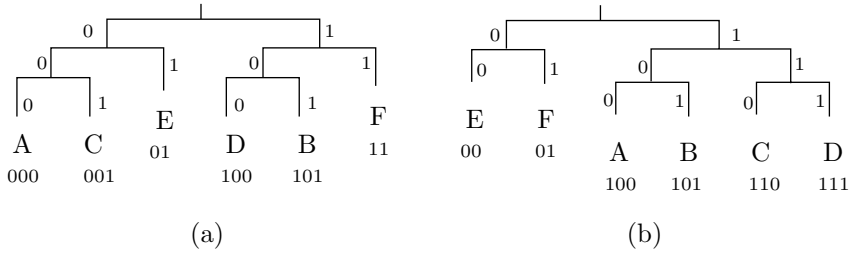


Figure 3.11: Two Huffman Trees.

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 also assigned the base value of 0 (more accurately, twice 0, twice the number of codes of the previous group). 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 (3.1), we apply this method to generate its standard Huffman code tree (listed in Table 3.9).

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 3.9.

> How many a dispute could have been deflated into a single paragraph if the disputants had dared to define their terms.
>
> —Aristotle

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 output.

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 3.12a,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 output, 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 output as the final, compressed code of one prefix-code table. In mode 3, each block of compressed data requires two prefix-code tables, so two such sequences are written on the output.

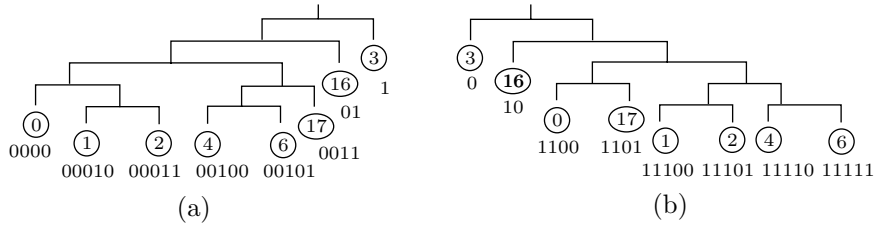


Figure 3.12: Two Huffman Trees for Code Lengths.

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.

3.3.3 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.

> If it wasn't for faith, there would be no living in this world; we couldn't even eat hash with any safety.
>
> —Josh Billings

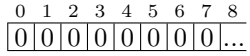
Instead of separate look-ahead and search buffers, the encoder should have a single, 32 Kb buffer. The buffer is filled up with input data and initially all of it is a look-ahead buffer. In the original LZ77 method, once symbols have been examined, they are moved into the search buffer. The Deflate encoder, 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 boundary between the look-ahead and search buffers. 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

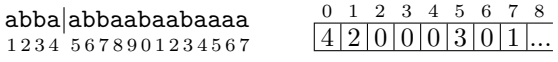
```
abbaabbaabaabaaaa
1234567890123456 7
```

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

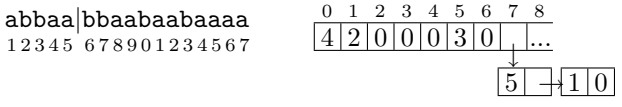


We assume that the first triplet `abb` hashes to 7. The encoder outputs the raw symbol `a`, moves this symbol to the search buffer (by moving the separator between the two buffers to the right), and sets cell 7 of the hash table to 1.

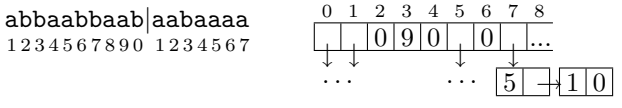
The next three steps hash the strings `bba`, `baa`, and `aab` to, say, 1, 5, and 0. The encoder outputs the three raw symbols `b`, `b`, and `a`, moves the separator, and updates the hash table as follows:



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.



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.

◇ **Exercise 3.9:** Hashing 3-byte sequences prevents the encoder from finding matches of length 1 and 2 bytes. Is this a serious limitation?

3.3.4 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 smaller for executable files, and somewhat bigger 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.

Chapter Summary

The Huffman algorithm is based on the probabilities of the individual data symbols, which is why it is considered a statistical compression method. Dictionary-based compression methods are different. They do not compute or estimate symbol probabilities and they do not use a statistical model of the data. They are based on the fact that the data files that are of interest to us, the files we want to compress and keep for later use, are not random. A typical data file features redundancies in the form of patterns and repetitions of data symbols.

A dictionary-based compression method selects strings of symbols from the input and employs a dictionary to encode each string as a *token*. The dictionary consists of

strings of symbols, and it may be static or dynamic (adaptive). The former type is permanent, sometimes allowing the addition of strings but no deletions, whereas the latter type holds strings previously found in the input, thereby allowing for additions and deletions of strings as new input is being read.

If the data features many repetitions, then many input strings will match strings in the dictionary. A matched string is replaced by a token, and compression is achieved if the token is shorter than the matched string. If the next input symbols is not found in the dictionary, then it is output in raw form and is also added to the dictionary. The following points are especially important: (1) Any dictionary-based method must write the raw items and tokens on the output such that the decoder will be able to distinguish them. (2) Also, the capacity of the dictionary is finite and any particular algorithm must have explicit rules specifying what to do when the (adaptive) dictionary fills up. Many dictionary-based methods have been developed over the years, and these two points constitute the main differences between them.

This book describes the following dictionary-based compression methods. The LZ77 algorithm (Section 1.3.1) is simple but not very efficient because its output tokens are triplets and are therefore large. The LZ78 method (Section 3.1) generates tokens that are pairs, and the LZW algorithm (Section 3.2) output single-item tokens. The Deflate algorithm (Section 3.3), which lies at the heart of the various zip implementations, is more sophisticated. It employs several types of blocks and a hash table, for a very effective compression.

Self-Assessment Questions

1. Redo Exercise 3.1 for various values of P (the probability of a match).

2. Study the topic of patents in data compression. A good starting point is [patents 07].

3. Test your knowledge of the LZW algorithm by manually encoding several short strings, similar to Exercise 3.3.

Words—so innocent and powerless as they are, as standing in a dictionary, how potent for good and evil they become in the hands of one who knows how to combine them.

—Nathaniel Hawthorne