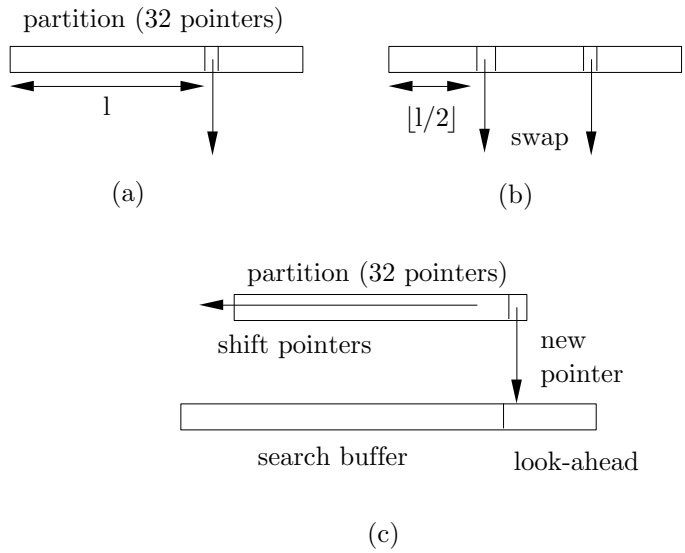


Figure 6.22: Updating an LZRW4 Partition.

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.

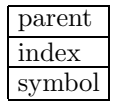
Suppose that string `abc` has already been input, symbol by symbol, and has been stored in the tree in the three nodes at locations 97, 266, and 284. Following that, the encoder has just input the next symbol `d`. The encoder now searches for string `abcd`, or, more specifically, for a node containing 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

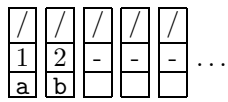
2. The node contains a parent pointer of 284 and the ASCII code of `d`. This means that string `abcd` is already in the tree. The encoder inputs the next symbol, say `e`, and searches the dictionary tree for string `abcde`.
3. The node contains something else. This means that another hashing of a pointer 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.

characters AC at positions 19–20 are a repeat of the string at positions 8–9, so they will be encoded as a string of length 2 at offset $20 - 9 = 11$.

6.6: The decoder interprets the first 1 of the end marker as the start of a token. The second 1 is interpreted as the prefix of a 7-bit offset. The next 7 bits are 0, and they identify the end marker as such, since a "normal" offset cannot be zero.

6.7: This is straightforward. The remaining steps are shown in Table Ans.21

Dictionary		Token	Dictionary		Token
15	␣t	(4, t)	21	␣si	(19,i)
16	e	(0, e)	22	c	(0, c)
17	as	(8, s)	23	k	(0, k)
18	es	(16,s)	24	␣se	(19,e)
19	␣s	(4, s)	25	al	(8, l)
20	ea	(4, a)	26	s(eof)	(1, (eof))

Table Ans.21: Next 12 Encoding Steps in the LZ78 Example.

6.8: Table Ans.22 shows the last three steps.

p_src	3 chars	Hash index	P	Output	Binary output
11	h t	7	any→11	h	01101000
12	␣th	5	5→12	4,7	0000|0011|00000111
16	ws			ws	01110111|01110011

Table Ans.22: Last Steps of Encoding that thatch thaws.

The final compressed stream consists of 1 control word followed by 11 items (9 literals and 2 copy items)
0000010010000000|01110100|01101000|01100001|01110100|00100000|0000|0011
|00000101|01100011|01101000|0000|0011|00000111|01110111|01110011.

6.9: An example is a compression utility for a personal computer that maintains all the files (or groups of files) on the hard disk in compressed form, to save space. Such a utility should be transparent to the user; it should automatically decompress a file every time it is opened and automatically compress it when it is being closed. In order to be transparent, such a utility should be fast, with compression ratio being only a secondary feature.

6.10: Table Ans.23 summarizes the steps. The output emitted by the encoder is
97 (a), 108 (l), 102 (f), 32 (␣), 101 (e), 97 (a), 116 (t), 115 (s), 32 (␣), 256 (al), 102 (f), 265 (alf), 97 (a),
and the following new entries are added to the dictionary
(256: al), (257: lf), (258: f␣), (259: ␣e), (260: ea), (261: at), (262: ts),
(263: s␣), (264: ␣a), (265: alf), (266: fa), (267: alfa).

I	in dict?	new entry	output	I	in dict?	new entry	output
a	Y			s␣	N	263-s␣	115 (s)
al	N	256-al	97 (a)	␣	Y		
l	Y			␣a	N	264-␣a	32 (␣)
lf	N	257-lf	108 (l)	a	Y		
f	Y			al	Y		
f␣	N	258-f␣	102 (f)	alf	N	265-alf	256 (al)
␣	Y			f	Y		
␣e	N	259-␣e	32 (w)	fa	N	266-fa	102 (f)
e	Y			a	Y		
ea	N	260-ea	101 (e)	al	Y		
a	Y			alf	Y		
at	N	261-at	97 (a)	alfa	N	267-alfa	265 (alf)
t	Y			a	Y		
ts	N	262-ts	116 (t)	a,eof	N		97 (a)
s	Y						

Table Ans.23: LZW Encoding of "alf eats alfalfa".

6.11: The encoder inputs the first a into I, searches and finds a in the dictionary. It inputs the next a but finds that Ix, which is now aa, is not in the dictionary. The encoder thus adds string aa to the dictionary as entry 256 and outputs the token 97 (a). Variable I is initialized to the second a. The third a is input, so Ix is the string aa, which is now in the dictionary. I becomes this string, and the fourth a is input. Ix is now aaa which is not in the dictionary. The encoder thus adds string aaa to the dictionary as entry 257 and outputs 256 (aa). I is initialized to the fourth a. Continuing this process is straightforward.

The result is that strings aa, aaa, aaaa,... are added to the dictionary as entries 256, 257, 258,..., and the output is

$$97\ (a),\ 256\ (aa),\ 257\ (aaa),\ 258\ (aaaa),\ldots$$

The output consists of pointers pointing to longer and longer strings of as. The first k pointers thus point at strings whose total length is $1+2+\cdots+k=(k+k^2)/2$.

Assuming an input stream that consists of one million as, we can find the size of the compressed output stream by solving the quadratic equation $(k+k^2)/2=1000000$ for the unknown k. The solution is $k\approx 1414$. The original, 8-million bit input is thus compressed into 1414 pointers, each at least 9-bit (and in practice, probably 16-bit) long. The compression factor is thus either $8\text{M}/(1414\times 9)\approx 628.6$ or $8\text{M}/(1414\times 16)\approx 353.6$.

This is an impressive result but such input streams are rare (notice that this particular input can best be compressed by generating an output stream containing just "1000000 a", and without using LZW).

6.12: We simply follow the decoding steps described in the text. The results are:
1. Input 97. This is in the dictionary so set I=a and output a. String ax needs to be saved in the dictionary but x is still unknown.

2. Input 108. This is in the dictionary so set J=l and output l. Save al in entry 256. Set I=l.
3. Input 102. This is in the dictionary so set J=f and output f. Save lf in entry 257. Set I=f.
4. Input 32. This is in the dictionary so set J=␣ and output ␣. Save f␣ in entry 258. Set I=␣.
5. Input 101. This is in the dictionary so set J=e and output e. Save ␣e in entry 259. Set I=e.
6. Input 97. This is in the dictionary so set J=a and output a. Save ea in entry 260. Set I=a.
7. Input 116. This is in the dictionary so set J=t and output t. Save at in entry 261. Set I=t.
8. Input 115. This is in the dictionary so set J=s and output s. Save ts in entry 262. Set I=t.
9. Input 32. This is in the dictionary so set J=␣ and output ␣. Save s␣ in entry 263. Set I=␣.
10. Input 256. This is in the dictionary so set J=al and output al. Save ␣a in entry 264. Set I=al.
11. Input 102. This is in the dictionary so set J=f and output f. Save alf in entry 265. Set I=f.
12. Input 265. This has just been saved in the dictionary so set J=alf and output alf. Save fa in dictionary entry 266. Set I=alf.
13. Input 97. This is in the dictionary so set J=a and output a. Save alfa in entry 267 (even though it will never be used). Set I=a.
14. Read eof. Stop.

6.13: We assume that the dictionary is initialized to just the two entries (1: a) and (2:␣b). The encoder outputs

1 (a), 2 (b), 3 (ab), 5(aba), 4(ba), 7 (bab), 6 (abab), 9 (ababa), 8 (baba),...

and adds the new entries (3: ab), (4: ba), (5: aba), (6: abab), (7: bab), (8: baba), (9: ababa), (10: ababab), (11: babab),... to the dictionary. This regular behavior can be analyzed and the kth output pointer and dictionary entry predicted, but the effort is probably not worth it.

6.14: The answer to exercise 6.11 shows the relation between the size of the compressed file and the size of the largest dictionary string for the "worst case" situation (input that creates the longest strings). For a 1 Mbyte input stream, there will be 1,414 strings in the dictionary, the largest of which is 1,414 symbols long.

6.15: This is straightforward (Table Ans.24) but not very efficient since only one two-symbol dictionary phrase is used.

6.16: Table Ans.25 shows all the steps. In spite of the short input, the result is quite good (13 codes to compress 18-symbols) because the input contains concentrations of as and bs.