Application of Source Coding

■**Extended Huffman Codes**

■**Lempel-Ziv Codes**

Extended Huffman Codes

In applications where the size of the alphabet is large, P_{max} is generally quite small, and the amount of deviation from the entropy, especially in terms of a percentage of the rate, is quite small.

However, in cases where the alphabet is small and the probability of occurrence of the different letters is skewed, the value of P_{max} can be quite large and the Huffman code can become rather inefficient when compared to the entropy.

- Example:
- Consider a source that puts out iid letters from the alphabet A $= \{a_1, a_2, a_3\}$ with the probability model $P(a_1) = 0.8$, $P(a_2) = 0.02$, and $P(a_3) = 0.18$. The entropy for this source is 0.816 bits/symbol. A Huffman code for this source is shown in the following table.

Letters	**Codeword**
a_1	0
a_2	11
a_3	10

The average length for this code is 1.2 bits/symbol. The difference between the average code length and the entropy, or the redundancy, for this code is 0.384 bits/symbol, which is 47% of the entropy. This means that to code this sequence we would need 47% more bits than the minimum required.

We can sometimes reduce the coding rate by blocking more than one symbol together. To see how this can happen, consider a source S that emits a sequence of letters from an alphabet $A = \{a_1, a_2, \dots, a_m\}$. Each element of the sequence is generated independently of the other elements in the sequence. The entropy for this source is given by

$$H(S) = -\sum_{i=1}^{m} P(a_i) \log_2 P(a_i).$$

We know that we can generate a Huffman code for this source with rate R such that

$$H(S) \leq R < H(S) + 1.$$

Suppose we now encode the sequence by generating one codeword for every n symbols. As there are m^n combinations of n symbols, we will need m^n codewords in our Huffman code. We could generate this code by viewing the m^n symbols as letters of an *extended alphabet*

$$\mathcal{A}^{(n)} = \{\overbrace{a_1 a_1 \ldots a_1}^{n \text{ times}}, a_1 a_1 \ldots a_2, \ldots, a_1 a_1 \ldots a_m, a_1 a_1 \ldots a_2 a_1, \ldots, a_m a_m \ldots a_m\}$$

from a source $S^{(n)}$Let us denote the rate for the new source as $R^{(n)}$. Then we know that

$$H(S^{(n)}) \leq R^{(n)} < H(S^{(n)}) + 1.$$

$R^{(n)}$ is the number of bits required to code n symbols. Therefore, the number of bits required per symbol, R, is given by

$$R = \frac{1}{n} R^{(n)}.$$

The number of bits per symbol can be bounded as

$$\frac{H(S^{(n)})}{n} \leq R < \frac{H(S^{(n)})}{n} + \frac{1}{n}.$$

Example:

For the source described in the previous example, instead of generating a codeword for **every** symbol, we will generate a codeword for every ***two*** symbols. If we look at the source sequence two at a time, the number of possible symbol pairs, or size of the extended alphabet, is $3^2 = 9$. The extended alphabet, probability model, and Huffman code for this example are shown in following tables.

Letters	Probability	Codeword
a_1a_1	0.64	0
a_1a_2	0.016	10101
a_1a_3	0.144	11
a_2a_1	0.016	101000
a_2a_2	0.0004	10100101
a_2a_3	0.0036	1010011
a_3a_1	0.1440	100
a_3a_2	0.0036	10100100
a_3a_3	0.0324	1011

The average codeword length for this extended code is 1.7228 bits/symbol. However, each symbol in the extended alphabet corresponds to two symbols from the original alphabet. Therefore, in terms of the original alphabet, the average codeword length is $1.7228/2 \ = 0.8614$ bits/symbol. This redundancy is about 0.045 bits/symbol, which is only about 5.5% of the entropy.

Lempel-Ziv Codes

In the LZ77 approach, the dictionary is simply a portion of the previously encoded sequence. The encoder examines the input sequence through a sliding window. The window consists of two parts, a ***search buffer*** that contains a portion of the recently encoded sequence, and a ***look-ahead buffer*** that contains the next portion of the sequence to be encoded. In the following figure, the search buffer contains eight symbols, while the look-ahead buffer contains seven symbols.

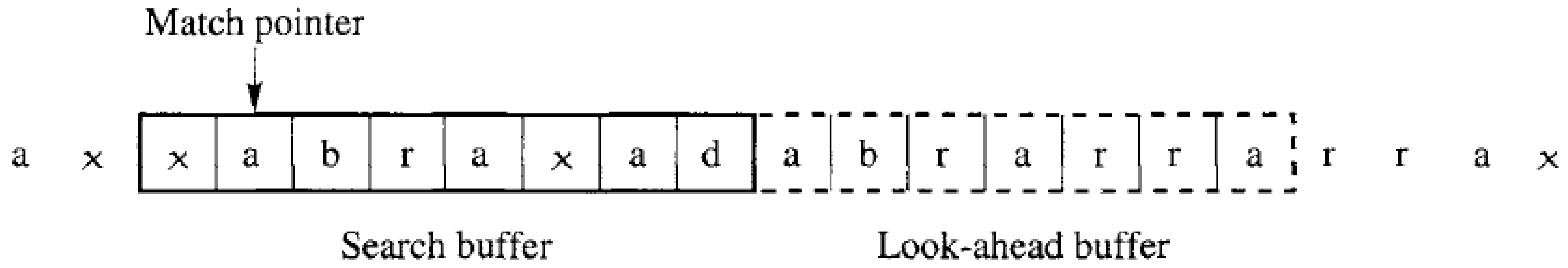


Figure 5.1

The encoder searches the search buffer for the longest match. Once the longest match has been found, the encoder encodes it with a triple (o, l, c), where $\boldsymbol{o}$ is the offset, $\boldsymbol{l}$ is the length of the match, and $\boldsymbol{c}$ is the codeword corresponding to the symbol in the look-ahead buffer that follows the match.

For example, in Figure 5.1 the pointer is pointing to the beginning of the longest match. The offset $\boldsymbol{o}$ in this case is 7, the length of the match $\boldsymbol{l}$ is 4, and the symbol in the look-ahead buffer following the match is $\boldsymbol{r}$.

Example: The LZ77 Approach

Suppose the sequence to be encoded is

$$\ldots cabracadabrarrarrad \ldots$$

Suppose the length of the window is 13, the size of the lookahead buffer is 6, and the current condition is

$$\boxed{cabraca\,|\,dabrar}$$

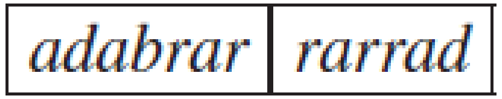
- with $\boldsymbol{dabrar}$ in the lookahead buffer. We look back in the already encoded portion of the window to find a match for $\boldsymbol{d}$. As we can see, there is no match, so we transmit the triple $(0, 0, C(d))$. The first two elements of the triple show that there is no match to $\boldsymbol{d}$ in the search buffer, while $C(d)$ is the code for the character $\boldsymbol{d}$.

For now, let’s continue with the encoding process. As we have encoded a single character, we move the window by one character. Now the contents of the buffer are

$$\boxed{abracad}\boxed{abrarr}$$

with $\boldsymbol{abrarr}$ in the lookahead buffer. Looking back from the current location, we find a match to $\boldsymbol{a}$ at an offset of two. The length of this match is one. Looking further back, we have another match for $\boldsymbol{a}$ at an offset of four; again the length of the match is one. Looking back even further in the window, we have a third match for $\boldsymbol{a}$ at an offset of seven. However, this time the length of the match is four. So we encode the string $\boldsymbol{abra}$ with the triple $(7, 4, C(r))$, and move the window forward by five characters.

The window now contains the following characters:



adabrar	*rarrad*

Now the lookahead buffer contains the string $\boldsymbol{rarrad}$. Looking back in the window, we find a match for $\boldsymbol{r}$ at an offset of one and a match length of one, and a second match at an offset of three with a match length of what at first appears to be three. It turns out we can use a match length of five instead of three.

Why this is so will become clearer when we decode the sequence. To see how the decoding works, let us assume that we have decoded the sequence $cabraca$ and we receive the triples $(0, 0, C(d))$, $(7, 4, C(r))$, and $(3, 5, C(d))$.

The first triple is easy to decode; there was no match within the previously decoded string, and the next symbol is $\boldsymbol{d}$. The decoded string is now $\boldsymbol{cabracad}$. The first element of the next triple tells the decoder to move the copy pointer back seven characters, and copy four characters from that point. The decoding process works as shown in the following figure 5.2.

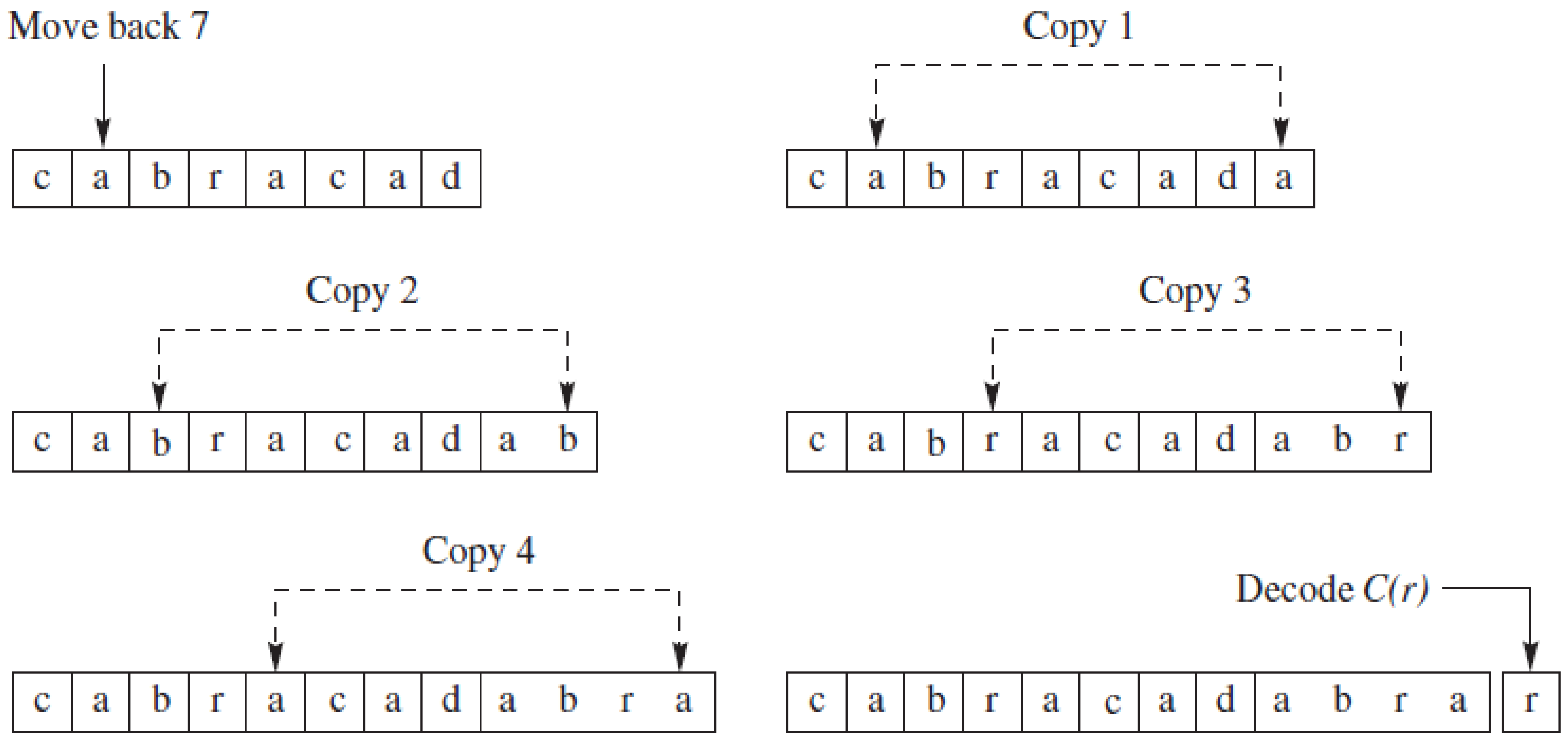
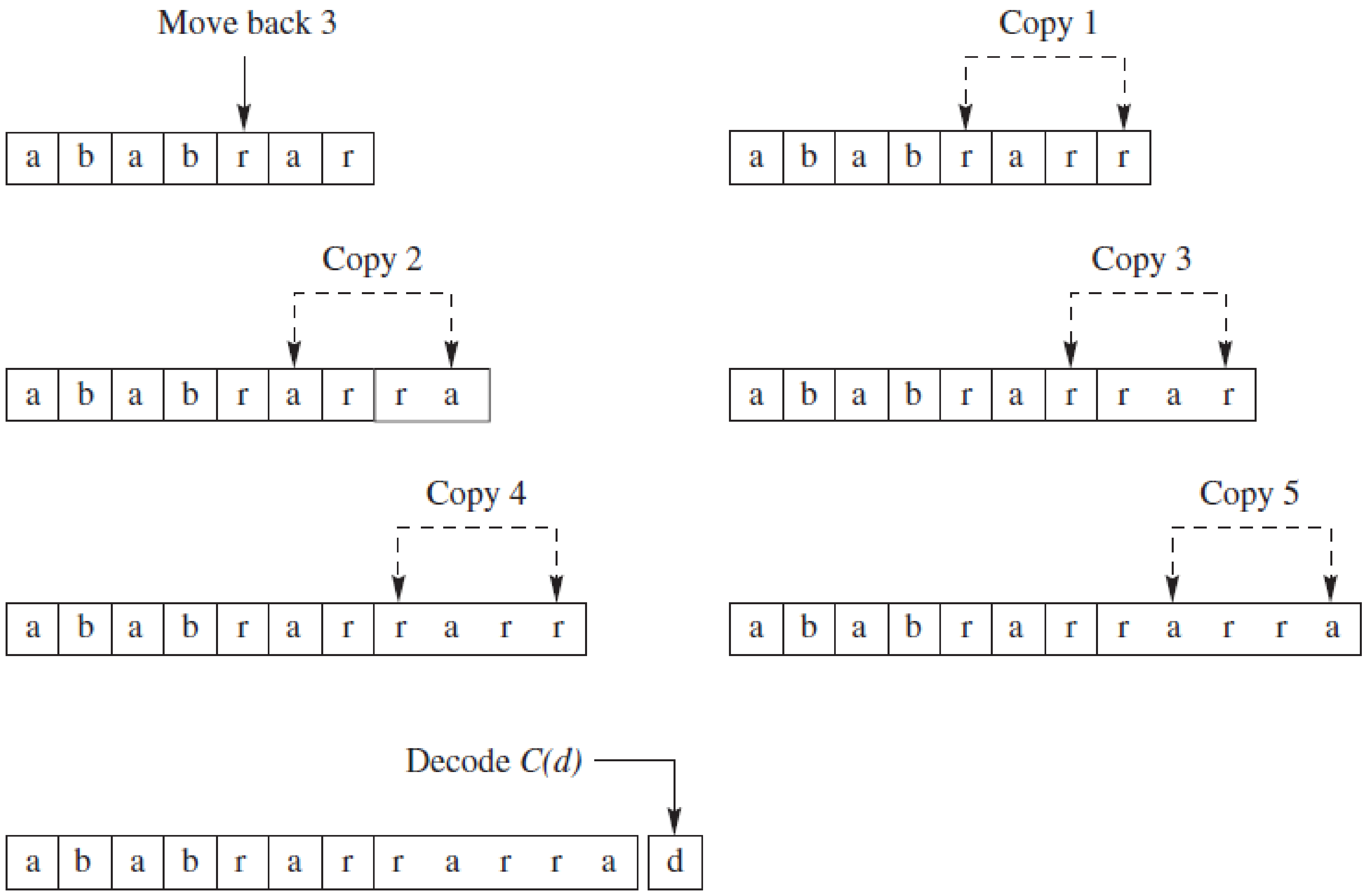

Move back 7
c a b r a c a d
Copy 1
c a b r a c a d a
Copy 2
c a b r a c a d a b
Copy 3
c a b r a c a d a b r
Copy 4
c a b r a c a d a b r a
Decode $C(r)$
c a b r a c a d a b r a r

Figure 5.2

Finally, let's see how the triple $(3, 5, C(d))$ gets decoded. We move back three characters and start copying. The first three characters we copy are $\boldsymbol{rar}$. The copy pointer moves once again, as shown in Figure 5.3, to copy the recently copied character $\boldsymbol{r}$.

Similarly, we copy the next character $\boldsymbol{a}$. Even though we started copying only three characters back, we end up decoding five characters. Notice that the match only has to *start* in the search buffer; it can extend into the lookahead buffer. In fact, if the last character in the lookahead buffer had been $\boldsymbol{r}$ instead of $\boldsymbol{d}$, followed by several more repetitions of $\boldsymbol{rar}$, the entire sequence of repeated $\boldsymbol{rar}$**s** could have been encoded with a single triple.



Move back 3
a b a b r a r
Copy 1
a b a b r a r r
Copy 2
a b a b r a r r a
Copy 3
a b a b r a r r a r
Copy 4
a b a b r a r r a r r
Copy 5
a b a b r a r r a r r a
Decode $C(d)$
a b a b r a r r a r r a d

Figure 5.3

The LZ77 approach implicitly assumes that like patterns will occur close together. It makes use of this structure by using the recent past of the sequence as the dictionary for encoding. However, this means that any pattern that recurs over a period longer than that covered by the coder window will not be captured. Consider Figure 5.4.

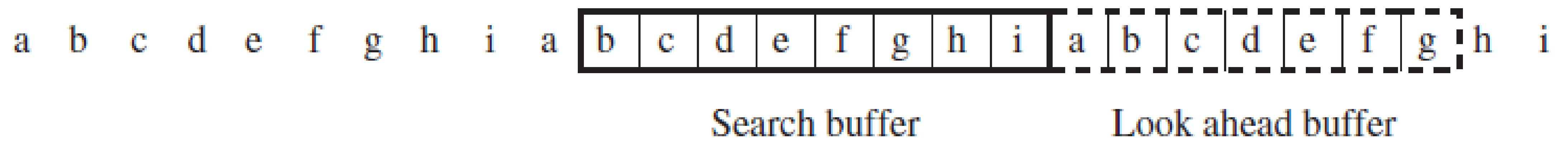


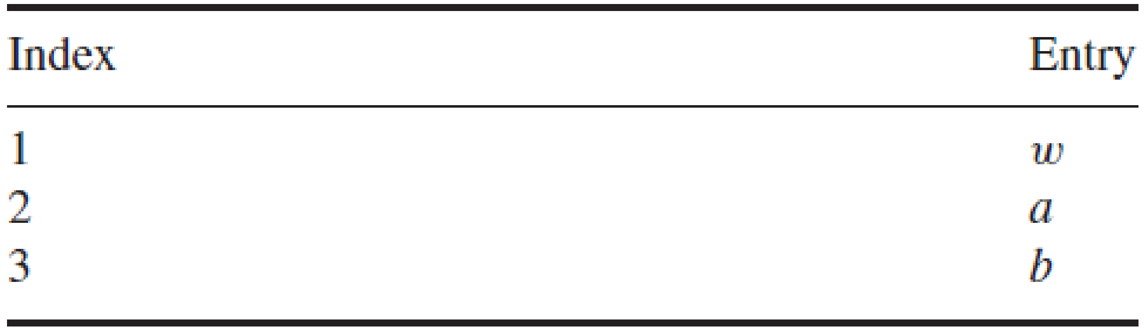
Figure 5.4

Example: The LZ78 Approach

Let us encode the following sequence using the LZ78 approach:

$$wabba\not{b}wabba\not{b}wabba\not{b}wabba\not{b}woo\not{b}woo\not{b}woo.$$

Where $\not{b}$ stands for space. Initially, the dictionary is empty, so the first few symbols encountered are encoded with the index value set to 0. The first three encoder outputs are $(0, C(w))$, $(0, C(a))$, and $(0, C(b))$, and the dictionary looks like the following table.



Index	Entry
1	w
2	a
3	b

The fourth symbol is **a** $\boldsymbol{b}$, which is the third entry in the dictionary. If we append the next symbol, we would get the pattern $\boldsymbol{ba}$, which is not in the dictionary, so we encode these two symbols as $(3, C(a))$ and add the pattern $\boldsymbol{ba}$ as the fourth entry in the dictionary.

Continuing in this fashion, the encoder output and the dictionary develop as in the following table. Notice that the entries in the dictionary generally keep getting longer, and if this particular sentence was repeated often, as it is in the song, after a while the entire sentence would be an entry in the dictionary.

	Dictionary	
Encoder Output	Index	Entry
$\langle 0, C(w)\rangle$	01	w
$\langle 0, C(a)\rangle$	02	a
$\langle 0, C(b)\rangle$	03	b
$\langle 3, C(a)\rangle$	04	ba
$\langle 0, C(\not{b})\rangle$	05	$\not{b}$
$\langle 1, C(a)\rangle$	06	wa
$\langle 3, C(b)\rangle$	07	bb
$\langle 2, C(\not{b})\rangle$	08	$a\not{b}$
$\langle 6, C(b)\rangle$	09	wab
$\langle 4, C(\not{b})\rangle$	10	$ba\not{b}$
$\langle 9, C(b)\rangle$	11	$wabb$
$\langle 8, C(w)\rangle$	12	$a\not{b}w$
$\langle 0, C(o)\rangle$	13	o
$\langle 13, C(\not{b})\rangle$	14	$o\not{b}$
$\langle 1, C(o)\rangle$	15	wo
$\langle 14, C(w)\rangle$	16	$o\not{b}w$
$\langle 13, C(o)\rangle$	17	oo