7.3 RUN-LENGTH CODING

Instead of assuming a memoryless source, *run-length coding* (RLC) exploits memory present in the information source. It is one of the simplest forms of data compression. The basic idea is that if the information source we wish to compress has the property that symbols tend to form continuous groups, instead of coding each symbol in the group individually, we can code one such symbol and the length of the group.

As an example, consider a bilevel image (one with only 1-bit black and white pixels) with monotone regions. This information source can be efficiently coded using run-length coding. In fact, since there are only two symbols, we do not even need to code any symbol at the start of each run. Instead, we can assume that the starting run is always of a particular color (either black or white) and simply code the length of each run.

The above description is the one-dimensional run-length coding algorithm. A two-dimensional variant of it is usually used to code bilevel images. This algorithm uses the coded run information in the previous row of the image to code the run in the current row. A full description of this algorithm can be found in [5].

7.4 VARIABLE-LENGTH CODING (VLC)

Since the entropy indicates the information content in an information source S, it leads to a family of coding methods commonly known as *entropy coding* methods. As described earlier, *variable-length coding* (VLC) is one of the best-known such methods. Here, we will study the Shannon–Fano algorithm, Huffman coding, and adaptive Huffman coding.

7.4.1 Shannon–Fano Algorithm

The Shannon–Fano algorithm was independently developed by Shannon at Bell Labs and Robert Fano at MIT [6]. To illustrate the algorithm, let's suppose the symbols to be coded are the characters in the word HELLO. The frequency count of the symbols is

Symbol	H	E	L	O
Count	1	1	2	1

The encoding steps of the Shannon–Fano algorithm can be presented in the following *top-down* manner:

1. Sort the symbols according to the frequency count of their occurrences.
2. Recursively divide the symbols into two parts, each with approximately the same number of counts, until all parts contain only one symbol.

A natural way of implementing the above procedure is to build a binary tree. As a convention, let's assign bit 0 to its left branches and 1 to the right branches.

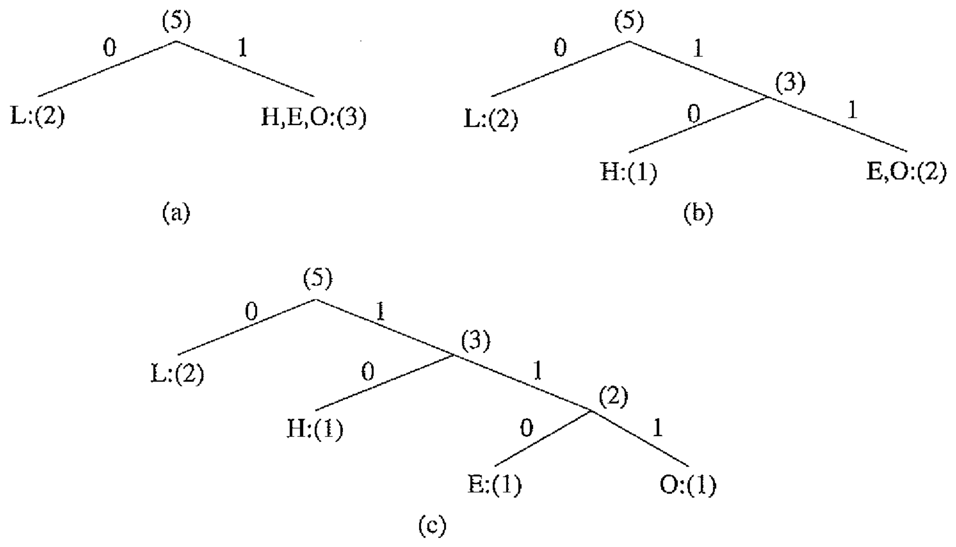


FIGURE 7.3: Coding tree for HELLO by the Shannon–Fano algorithm.

Initially, the symbols are sorted as LHEO. As Figure 7.3 shows, the first division yields two parts: (a) L with a count of 2, denoted as L:(2); and (b) H, E and O with a total count of 3, denoted as H,E,O:(3). The second division yields H:(1) and E,O:(2). The last division is E:(1) and O:(1).

Table 7.1 summarizes the result, showing each symbol, its frequency count, information content $\left(\log_2 \frac{1}{p_i}\right)$, resulting codeword, and the number of bits needed to encode each symbol in the word HELLO. The total number of bits used is shown at the bottom.

To revisit the previous discussion on entropy, in this case,

$$\begin{aligned}\eta &= p_L \cdot \log_2 \frac{1}{p_L} + p_H \cdot \log_2 \frac{1}{p_H} + p_E \cdot \log_2 \frac{1}{p_E} + p_O \cdot \log_2 \frac{1}{p_O} \\ &= 0.4 \times 1.32 + 0.2 \times 2.32 + 0.2 \times 2.32 + 0.2 \times 2.32 = 1.92\end{aligned}$$

TABLE 7.1: One result of performing the Shannon–Fano algorithm on HELLO.

Symbol	Count	$\log_2 \frac{1}{p_i}$	Code	Number of bits used
L	2	1.32	0	2
H	1	2.32	10	2
E	1	2.32	110	3
O	1	2.32	111	3
TOTAL number of bits:				10

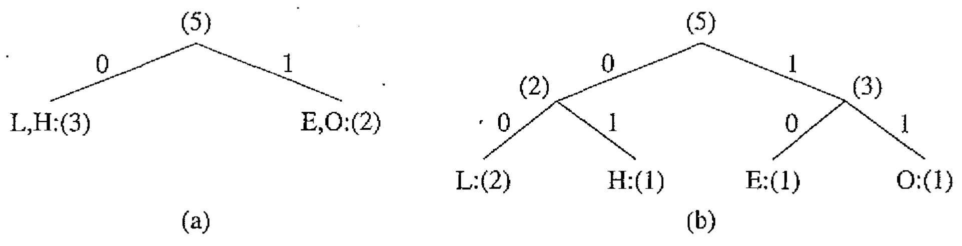


FIGURE 7.4: Another coding tree for HELLO by the Shannon–Fano algorithm.

This suggests that the minimum average number of bits to code each character in the word HELLO would be at least 1.92. In this example,, the Shannon–Fano algorithm uses an average of $10/5 = 2$ bits to code each symbol, which is fairly close to the lower bound of 1.92. Apparently, the result is satisfactory.

It should be pointed out that the outcome of the Shannon–Fano algorithm is not necessarily unique. For instance, at the first division in the above example, it would be equally valid to divide into the two parts L,H:(3) and E,O:(2). This would result in the coding in Figure 7.4. Table 7.2 shows the codewords are different now. Also, these two sets of codewords may behave differently when errors are present. Coincidentally, the total number of bits required to encode the world HELLO remains at 10.

The Shannon–Fano algorithm delivers satisfactory coding results for data compression, but it was soon outperformed and overtaken by the Huffman coding method.

7.4.2 Huffman Coding

First presented by David A. Huffman in a 1952 paper [7], this method attracted an overwhelming amount of research and has been adopted in many important and/or commercial applications, such as fax machines, JPEG, and MPEG.

In contradistinction to Shannon–Fano, which is top-down, the encoding steps of the Huffman algorithm are described in the following *bottom-up* manner. Let's use the same example word, HELLO. A similar binary coding tree will be used as above, in which the left branches are coded 0 and right branches 1. A simple list data structure is also used.

TABLE 7.2: Another result of performing the Shannon–Fano algorithm on HELLO.

Symbol	Count	$\log_2 \frac{1}{p_i}$	Code	Number of bits used
L	2	1.32	00	4
H	1	2.32	01	2
E	1	2.32	10	2
O	1	2.32	11	2
TOTAL number of bits:				10

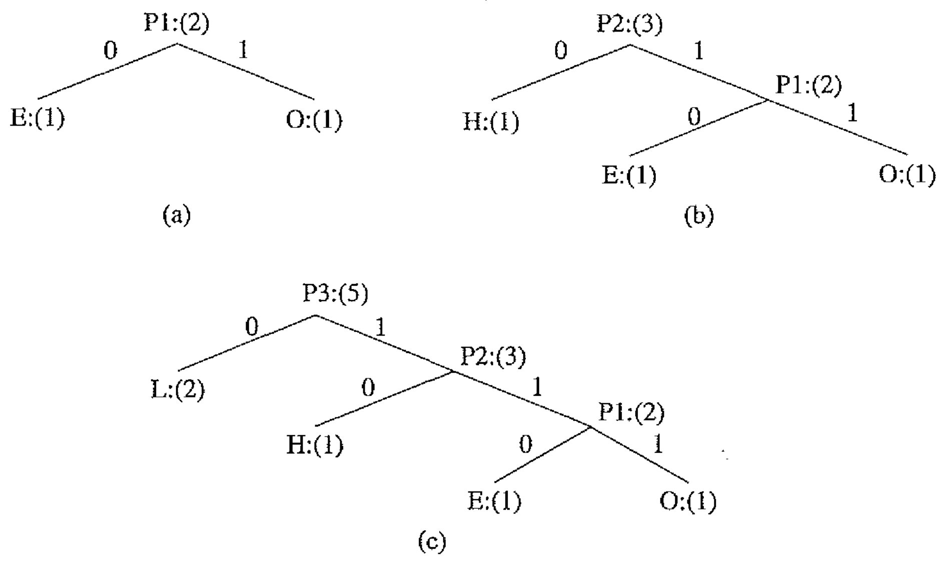


FIGURE 7.5: Coding tree for HELLO using the Huffman algorithm.

ALGORITHM 7.1 HUFFMAN CODING

1. Initialization: put all symbols on the list sorted according to their frequency counts.
2. Repeat until the list has only one symbol left.
   (a) From the list, pick two symbols with the lowest frequency counts. Form a Huffman subtree that has these two symbols as child nodes and create a parent node for them.
   (b) Assign the sum of the children's frequency counts to the parent and insert it into the list, such that the order is maintained.
   (c) Delete the children from the list.
3. Assign a codeword for each leaf based on the path from the root.

In the above figure, new symbols P1, P2, P3 are created to refer to the parent nodes in the Huffman coding tree. The contents in the list are illustrated below:

After initialization:	L H E O
After iteration (a):	L P1 H
After iteration (b):	L P2
After iteration (c):	P3

For this simple example, the Huffman algorithm apparently generated the same coding result as one of the Shannon–Fano results shown in Figure 7.3, although the results are usually better. The average number of bits used to code each character is also 2, (i.e., $(1 + 1 + 2 + 3 + 3)/5 = 2$). As another simple example, consider a text string containing a set of characters and their frequency counts as follows: A:(15), B:(7), C:(6), D:(6) and E:(5). It is easy to show that the Shannon–Fano algorithm needs a total of 89 bits to encode this string, whereas the Huffman algorithm needs only 87.

As shown above, if correct probabilities ("prior statistics") are available and accurate, the Huffman coding method produces good compression results. Decoding for the Huffman coding is trivial as long as the statistics and/or coding tree are sent before the data to be compressed (in the file header, say). This overhead becomes negligible if the data file is sufficiently large.

The following are important properties of Huffman coding:

- **Unique prefix property**. No Huffman code is a prefix of any other Huffman code. For instance, the code 0 assigned to L in Figure 7.5(c) is not a prefix of the code 10 for H or 110 for E or 111 for O; nor is the code 10 for H a prefix of the code 110 for E or 111 for O. It turns out that the unique prefix property is guaranteed by the above Huffman algorithm, since it always places all input symbols at the leaf nodes of the Huffman tree. The Huffman code is one of the *prefix codes* for which the unique prefix property holds. The code generated by the Shannon–Fano algorithm is another such example.

  This property is essential and also makes for an efficient decoder, since it precludes any ambiguity in decoding. In the above example, if a bit 0 is received, the decoder can immediately produce a symbol L without waiting for any more bits to be transmitted.

- **Optimality**. The Huffman code is a *minimum-redundancy code*, as shown in Huffman's 1952 paper [7]. It has been proven [8, 2] that the Huffman code is *optimal* for a given data model (i.e., a given, accurate, probability distribution):

  - The two least frequent symbols will have the same length for their Huffman codes, differing only at the last bit. This should be obvious from the above algorithm.
  - Symbols that occur more frequently will have shorter Huffman codes than symbols that occur less frequently. Namely, for symbols s_i and s_j, if $p_i \geq p_j$ then $l_i \leq l_j$, where l_i is the number of bits in the codeword for s_i.
  - It has been shown (see [2]) that the average code length for an information source S is strictly less than $\eta + 1$. Combined with Eq.(7.5), we have

$$\eta \leq \bar{l} < \eta + 1 \tag{7.6}$$

Extended Huffman Coding. The discussion of Huffman coding so far assigns each symbol a codeword that has an *integer* bit length. As stated earlier, $\log_2 \frac{1}{p_i}$ indicates the amount of information contained in the information source s_i, which corresponds to the number of bits needed to represent it. When a particular symbol s_i has a large probability (close to 1.0), $\log_2 \frac{1}{p_i}$ will be close to 0, and assigning one bit to represent that symbol will be costly. Only when the probabilities of all symbols can be expressed as 2^{-k}, where k is a positive integer, would the average length of codewords be truly optimal — that is, $\bar{l} \equiv \eta$. Clearly, $\bar{l} > \eta$ in most cases.

One way to address the problem of integral codeword length is to group several symbols and assign a single codeword to the group. Huffman coding of this type is called *Extended Huffman Coding* [2]. Assume an information source has alphabet $S = \{s_1, s_2, \ldots, s_n\}$. If k symbols are grouped together, then the *extended alphabet* is

$$S^{(k)} = \{\overbrace{s_1 s_1 \ldots s_1}^{k \text{ symbols}}, s_1 s_1 \ldots s_2, \ldots, s_1 s_1 \ldots s_n, s_1 s_1 \ldots s_2 s_1, \ldots, s_n s_n \ldots s_n\}$$

Note that the size of the new alphabet $S^{(k)}$ is n^k. If k is relatively large (e.g., $k \geq 3$), then for most practical applications where $n \gg 1$, n^k would be a very large number, implying a huge symbol table. This overhead makes Extended Huffman Coding impractical.

As shown in [2], if the entropy of S is η, then the average number of bits needed for each symbol in S is now

$$\eta \leq \bar{l} < \eta + \frac{1}{k} \tag{7.7}$$

so we have shaved quite a bit from the coding schemes' bracketing of the theoretical best limit. Nevertheless, this is not as much of an improvement over the original Huffman coding (where group size is 1) as one might have hoped for.

7.4.3 Adaptive Huffman Coding

The Huffman algorithm requires prior statistical knowledge about the information source, and such information is often not available. This is particularly true in multimedia applications, where future data is unknown before its arrival, as for example in live (or streaming) audio and video. Even when the statistics are available, the transmission of the symbol table could represent heavy overhead.

For the non-extended version of Huffman coding, the above discussion assumes a so-called *order-0* model — that is, symbols/characters were treated singly, without any context or history maintained. One possible way to include contextual information is to examine k preceding (or succeeding) symbols each time; this is known as an *order-k* model. For example, an order-1 model can incorporate such statistics as the probability of "qu" in addition to the individual probabilities of "q" and "u". Nevertheless, this again implies that much more statistical data has to be stored and sent for the order-k model when $k \geq 1$.

The solution is to use *adaptive* compression algorithms, in which statistics are gathered and updated dynamically as the datastream arrives. The probabilities are no longer based on prior knowledge but on the actual data received so far. The new coding methods are "adaptive" because, as the probability distribution of the received symbols changes, symbols will be given new (longer or shorter) codes. This is especially desirable for multimedia data, when the content (the music or the color of the scene) and hence the statistics can change rapidly.

As an example, we introduce the *Adaptive Huffman Coding* algorithm in this section. Many ideas, however, are also applicable to other adaptive compression algorithms.

PROCEDURE 7.1 Procedures for Adaptive Huffman Coding

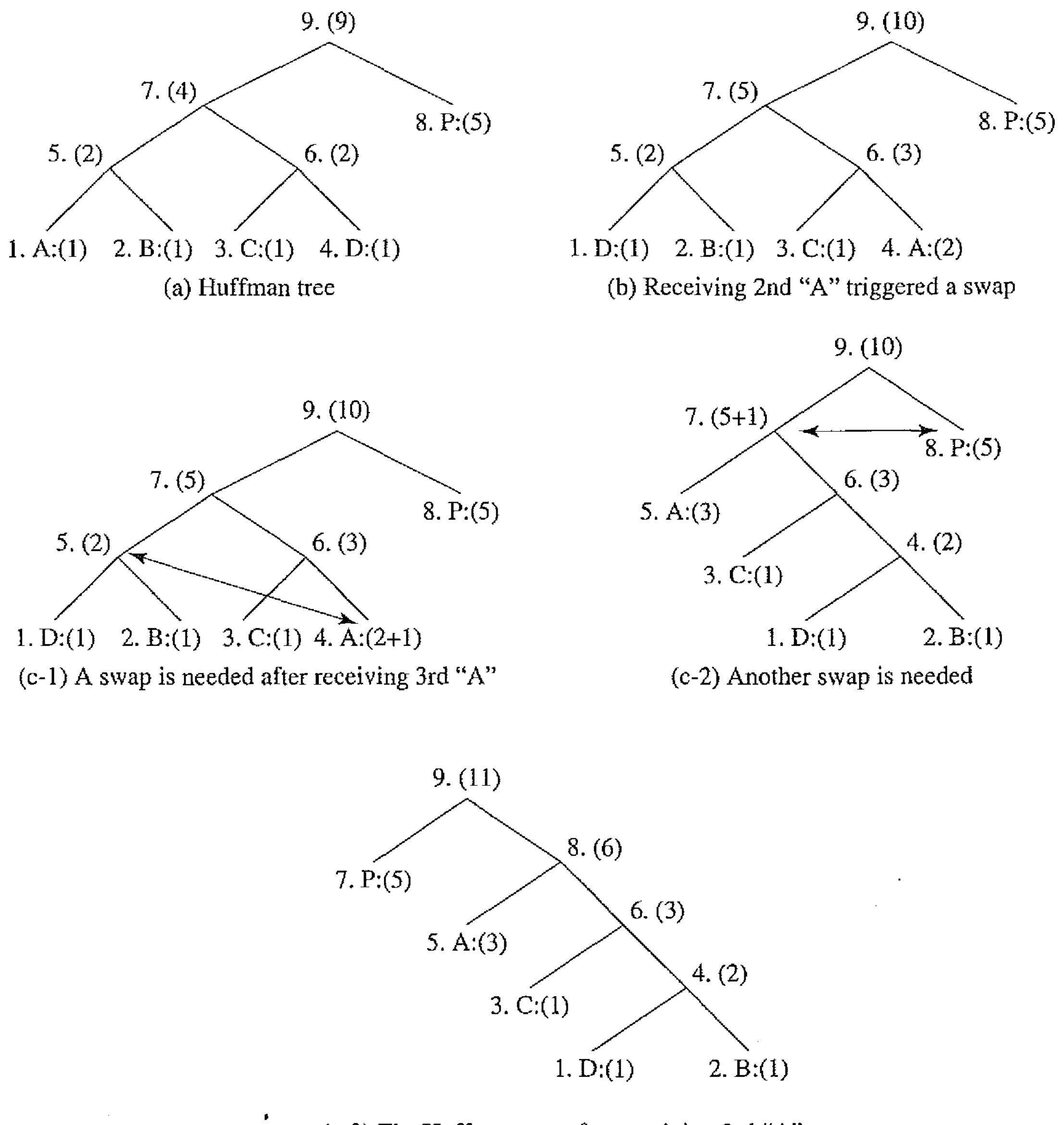
```
ENCODER                          DECODER
-------                          -------

Initial_code();                  Initial_code();
while not EOF                    while not EOF
  {                                  {
    get(c);                           decode(c);
    encode(c);                        output(c);
    update_tree(c);                   update_tree(c);
  }                                  }
```

- `Initial_code` assigns symbols with some initially agreed-upon codes, without any prior knowledge of the frequency counts for them. For example, some conventional code such as ASCII may be used for coding character symbols.
- `update_tree` is a procedure for constructing an adaptive Huffman tree. It basically does two things: it increments the frequency counts for the symbols (including any new ones), and updates the configuration of the tree.
  - The Huffman tree must always maintain its *sibling property* — that is, all nodes (internal and leaf) are arranged in the order of increasing counts. Nodes are numbered in order from left to right, bottom to top. (See Figure 7.6, in which the first node is 1.A:(1), the second node is 2.B:(1), and so on, where the numbers in parentheses indicates the count.) If the sibling property is about to be violated, a *swap* procedure is invoked to update the tree by rearranging the nodes.
  - When a swap is necessary, the farthest node with count N is swapped with the node whose count has just been increased to $N + 1$. Note that if the node with count N is not a leaf-node — it is the root of a subtree — the entire subtree will go with it during the swap.
- The encoder and decoder must use exactly the same `Initial_code` and `update_tree` routines.

Figure 7.6(a) depicts a Huffman tree with some symbols already received. Figure 7.6(b) shows the updated tree after an additional A (i.e., the second A) was received. This increased the count of As to $N + 1 = 2$ and triggered a swap. In this case, the farthest node with count $N = 1$ was D:(1). Hence, A:(2) and D:(1) were swapped.

Apparently, the same result could also be obtained by first swapping A:(2) with B:(1), then with C:(1), and finally with D:(1). The problem is that such a procedure would take three swaps; the rule of swapping with "the farthest node with count N" helps avoid such unnecessary swaps.



(a) Huffman tree

(b) Receiving 2nd "A" triggered a swap

(c-1) A swap is needed after receiving 3rd "A"

(c-2) Another swap is needed

(c-3) The Huffman tree after receiving 3rd "A"

FIGURE 7.6: Node swapping for updating an adaptive Huffman tree: (a) a Huffman tree; (b) receiving 2nd "A" triggered a swap; (c-1) a swap is needed after receiving 3rd "A"; (c-2) another swap is needed; (c-3) the Huffman tree after receiving 3rd "A".

The update of the Huffman tree after receiving the third A is more involved and is illustrated in the three steps shown in Figure 7.6(c-1) to (c-3). Since A:(2) will become A:(3) (temporarily denoted as A:(2+1)), it is now necessary to swap A:(2+1) with the fifth node. This is illustrated with an arrow in Figure 7.6(c-1).

Since the fifth node is a non-leaf node, the subtree with nodes 1. D:(1), 2. B:(1), and 5. (2) is swapped as a whole with A:(3). Figure 7.6(c-2) shows the tree after this first swap. Now the seventh node will become (5+1), which triggers another swap with the eighth node. Figure 7.6(c-3) shows the Huffman tree after this second swap.

The above example shows an update process that aims to maintain the sibling property of the adaptive Huffman tree — the update of the tree sometimes requires more than one swap. When this occurs, the swaps should be executed in multiple steps in a "bottom-up" manner, starting from the lowest level where a swap is needed. In other words, the update is carried out sequentially: tree nodes are examined in order, and swaps are made whenever necessary.

To clearly illustrate more implementation details, let's examine another example. Here, we show exactly what *bits* are sent, as opposed to simply stating how the tree is updated.

EXAMPLE 7.1 Adaptive Huffman Coding for Symbol String AADCCDD

Let's assume that the initial code assignment for both the encoder and decoder simply follows the ASCII order for the 26 symbols in an alphabet, A through Z, as Table 7.3 shows. To improve the implementation of the algorithm, we adopt an additional rule: if any character/symbol is to be sent the first time, it must be preceded by a special symbol, NEW. The initial code for NEW is 0. The *count* for NEW is always kept as 0 (the count is never increased); hence it is always denoted as NEW:(0) in Figure 7.7.

Figure 7.7 shows the Huffman tree after each step. Initially, there is no tree. For the first A, 0 for NEW and the initial code 00001 for A are sent. Afterward, the tree is built and shown as the first one, labeled A. Now both the encoder and decoder have constructed the same first tree, from which it can be seen that the code for the second A is 1. The code sent is thus 1.

After the second A, the tree is updated, shown labeled as AA. The updates after receiving D and C are similar. More subtrees are spawned, and the code for NEW is getting longer — from 0 to 00 to 000.

TABLE 7.3: Initial code assignment for AADCCDD using adaptive Huffman coding.

	Initial Code
NEW:	0
A:	00001
B:	00010
C:	00011
D:	00100
.	.
.	.
.	.

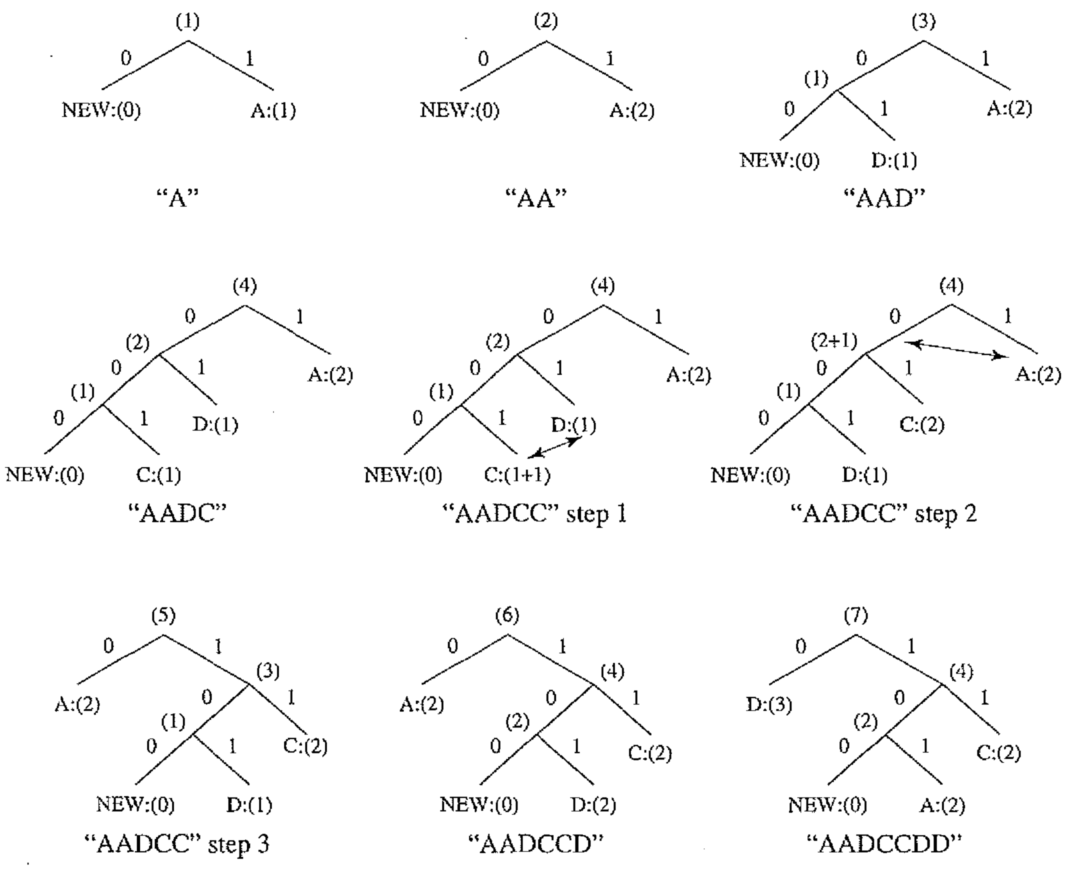


FIGURE 7.7: Adaptive Huffman tree for AADCCDD.

From AADC to AADCC takes two swaps. To illustrate the update process clearly, this is shown in three steps, with the required swaps again indicated by arrows.

- **AADCC Step 1.** The frequency count for C is increased from 1 to $1 + 1 = 2$; this necessitates its swap with D:(1).
- **AADCC Step 2.** After the swap between C and D, the count of the parent node of C:(2) will be increased from 2 to $2 + 1 = 3$; this requires its swap with A:(2).
- **AADCC Step 3.** The swap between A and the parent of C is completed.

Table 7.4 summarizes the sequence of symbols and code (zeros and ones) being sent to the decoder.

TABLE 7.4: Sequence of symbols and codes sent to the decoder

Symbol	NEW	A	A	NEW	D	NEW	C	C	D	D
Code	0	00001	1	0	00100	00	00011	001	101	101

It is important to emphasize that the code for a particular symbol often changes during the adaptive Huffman coding process. The more frequent the symbol up to the moment, the shorter the code. For example, after AADCCDD, when the character D overtakes A as the most frequent symbol, its code changes from 101 to 0. This is of course fundamental for the adaptive algorithm — codes are reassigned dynamically according to the new probability distribution of the symbols.

The "Squeeze Page" on this book's web site provides a Java applet for adaptive Huffman coding that should aid you in learning this algorithm.

7.5 DICTIONARY-BASED CODING

The Lempel-Ziv-Welch (LZW) algorithm employs an adaptive, dictionary-based compression technique. Unlike variable-length coding, in which the lengths of the codewords are different, LZW uses fixed-length codewords to represent variable-length strings of symbols/characters that commonly occur together, such as words in English text.

As in the other adaptive compression techniques, the LZW encoder and decoder builds up the same dictionary dynamically while receiving the data — the encoder and the decoder both develop the same dictionary. Since a single code can now represent more than one symbol/character, data compression is realized.

LZW proceeds by placing longer and longer repeated entries into a dictionary, then emitting the *code* for an element rather than the string itself, if the element has already been placed in the dictionary. The predecessors of LZW are LZ77 [9] and LZ78 [10], due to Jacob Ziv and Abraham Lempel in 1977 and 1978. Terry Welch [11] improved the technique in 1984. LZW is used in many applications, such as UNIX `compress`, GIF for images, V.42 bis for modems, and others.

ALGORITHM 7.2 LZW COMPRESSION

```
BEGIN
   s = next input character;
   while not EOF
       {
         c = next input character;

         if s + c exists in the dictionary
            s = s + c;
```

```
        else
          {
          output the code for s;
          add string s + c to the dictionary with a new code;
          s = c;
          }
        }
   output the code for s;
END
```

EXAMPLE 7.2 LZW Compression for String ABABBABCABABBA

Let's start with a very simple dictionary (also referred to as a *string table*), initially containing only three characters, with codes as follows:

code	string
1	A
2	B
3	C

Now if the input string is ABABBABCABABBA, the LZW compression algorithm works as follows:

s	c	output	code	string
			1	A
			2	B
			3	C
A	B	1	4	AB
B	A	2	5	BA
A	B			
AB	B	4	6	ABB
B	A			
BA	B	5	7	BAB
B	C	2	8	BC
C	A	3	9	CA
A	B			
AB	A	4	10	ABA
A	B			
AB	B			
ABB	A	6	11	ABBA
A	EOF	1		

The output codes are 1 2 4 5 2 3 4 6 1. Instead of 14 characters, only 9 codes need to be sent. If we assume each character or code is transmitted as a byte, that is quite a saving (the compression ratio would be 14/9 = 1.56). (Remember, the LZW is an adaptive algorithm, in which the encoder and decoder independently build their own string tables. Hence, there is no overhead involving transmitting the string table.)

Obviously, for our illustration the above example is replete with a great deal of redundancy in the input string, which is why it achieves compression so quickly. In general, savings for LZW would not come until the text is more than a few hundred bytes long.

The above LZW algorithm is simple, and it makes no effort in selecting optimal new strings to enter into its dictionary. As a result, its string table grows rapidly, as illustrated above. A typical LZW implementation for textual data uses a 12-bit codelength. Hence, its dictionary can contain up to 4,096 entries, with the first 256 (0–255) entries being ASCII codes. If we take this into account, the above compression ratio is reduced to $(14 \times 8)/(9 \times 12) = 1.04$.

ALGORITHM 7.3 LZW DECOMPRESSION (SIMPLE VERSION)

```
BEGIN
   s = NIL;
   while not EOF
       {
         k = next input code;
         entry = dictionary entry for k;
         output entry;
         if (s != NIL)
            add string s + entry[0] to dictionary
            with a new code;
         s = entry;
       }
END
```

EXAMPLE 7.3 LZW decompression for string ABABBABCABABBA

Input codes to the decoder are 1 2 4 5 2 3 4 6 1. The initial string table is identical to what is used by the encoder.

The LZW decompression algorithm then works as follows:

```
   s    k   entry/output   code  string
  ---------------------------------------
                            1      A
                            2      B
                            3      C
  ---------------------------------------
  NIL   1         A
```

A	2	B	4	AB
B	4	AB	5	BA
AB	5	BA	6	ABB
BA	2	B	7	BAB
B	3	C	8	BC
C	4	AB	9	CA
AB	6	ABB	10	ABA
ABB	1	A	11	ABBA
A	EOF			

Apparently the output string is ABABBABCABABBA — a truly lossless result!

LZW Algorithm Details A more careful examination of the above simple version of the LZW decompression algorithm will reveal a potential problem. In adaptively updating the dictionaries, the encoder is sometimes ahead of the decoder. For example, after the sequence ABABB, the encoder will output code 4 and create a dictionary entry with code 6 for the new string ABB.

On the decoder side, after receiving the code 4, the output will be AB, and the dictionary is updated with code 5 for a new string, BA. This occurs several times in the above example, such as after the encoder outputs another code 4, code 6. In a way, this is anticipated — after all, it is a sequential process, and the encoder had to be ahead. In this example, this did not cause problem.

Welch [11] points out that the simple version of the LZW decompression algorithm will break down when the following scenario occurs. Assume that the input string is ABABBABCABBABBAX....

The LZW encoder:

s	c	output	code	string
			1	A
			2	B
			3	C
A	B	1	4	AB
B	A	2	5	BA
A	B			
AB	B	4	6	ABB
B	A			
BA	B	5	7	BAB
B	C	2	8	BC
C	A	3	9	CA
A	B			
AB	B			

ABB	A	6	10	ABBA
A	B			
AB	B			
ABB	A			
ABBA	X	10	11	ABBAX
		.		
		.		
		.		

The sequence of output codes from the encoder (and hence the input codes for the decoder) is 1 2 4 5 2 3 6 10....

The simple LZW decoder:

s	k	entry/output	code	string
			1	A
			2	B
			3	C
NIL	1	A		
A	2	B	4	AB
B	4	AB	5	BA
AB	5	BA	6	ABB
BA	2	B	7	BAB
B	3	C	8	BC
C	6	ABB	9	CA
ABB	10	???		

"???" indicates that the decoder has encountered a difficulty: no dictionary entry exists for the last input code, 10. A closer examination reveals that code 10 was most recently created at the encoder side, formed by a concatenation of Character, String, Character. In this case, the character is A, and string is BB — that is, A + BB + A. Meanwhile, the sequence of the output symbols from the encoder are A, BB, A, BB, A.

This example illustrates that whenever the sequence of symbols to be coded is Character, String, Character, String, Character, and so on, the encoder will create a new code to represent Character + String + Character and use it right away, before the decoder has had a chance to create it!

Fortunately, this is the only case in which the above simple LZW decompression algorithm will fail. Also, when this occurs, the variable s = Character + String. A modified version of the algorithm can handle this exceptional case by checking whether the input code has been defined in the decoder's dictionary. If not, it will simply assume that the code represents the symbols $s + s[0]$; that is Character + String + Character.