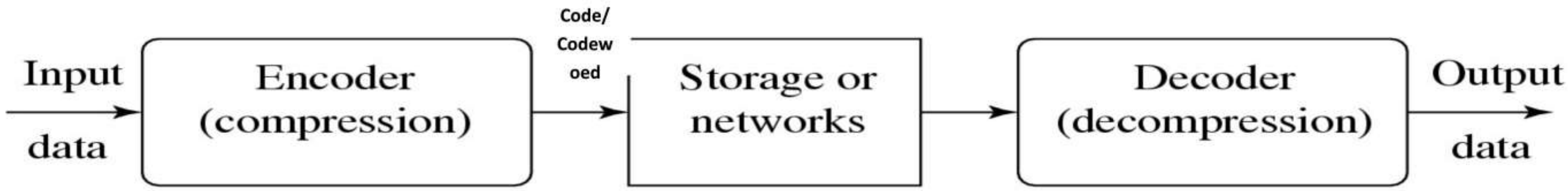
Chapter 7
Lossless Compression Algorithms

Presented By
Dr. Samaa Shohieb

Agenda

7.1 Introduction

- **Compression**: the process of coding that will effectively reduce the **total number of bits (data)** needed to represent **certain information.**



- **Fig. 7.1**: A General Data Compression Scheme.

7.1 Introduction (Cont'd)

- If the compression and decompression processes induce **no information loss**, then the compression scheme is ***lossless***; otherwise, it is ***lossy***.

- **Compression ratio**= $(B_0 - B_1) / B_0$
- B_0 : number of bits **before** compression
- B_1 : number of bits **after** compression

7.2 Basics of Information Theory

- The ***entropy*** ***η*** of an information ***source*** with **alphabet** $S = \{s_1, s_2, \ldots, s_n\}$ with a length L is:

$$\eta = H(S) = \sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i} \qquad (7.2)$$

$$= -\sum_{i=1}^{n} p_i \log_2 p_i \qquad (7.3)$$

- **p_i – probability that symbol s_i will occur in S.**
- ***η<=L***

- $\log_2 \frac{1}{p_i}$ – indicates the **amount of information** (self-information as defined by Shannon) contained in s_i, which corresponds to the number of bits needed to encode s_i.

Efficiency

- Given an alphabet of symbols, the goal of any coding system is to transmit the information contained in a string composed of these symbols in the most efficient manner, that is, in the least number of bits.
- The coding system needs to evaluate the frequency of occurrence of each symbol in the string and then decide how many bits to code each symbol so that the overall number of bits used for that string is the least.
- Optimal average symbol length is given by the entropy.

Efficiency

- Thus, any coding system's goal is to assign binary codes to each symbol in a manner so as to minimize the average symbol length.
- In such cases, *efficiency* is an important metric to evaluate the coding system's ability of compression and is defined as

$$\text{Efficiency} = \frac{Entropy}{Average\ Symbol\ length}$$

7.3 Run-Length Coding

- ***Run-Length Coding* (*RLC*)** exploits memory present in the information source.

- **Rationale for RLC**: **if** the information source **has the property that symbols tend to form continuous groups,**➔ then such **symbol and the length** of the **group can be coded**.

- Ex. BBBBEEEECCCDDDD (15Bytes)
  - 4B4E3C4D (8Bytes)

7.4 Variable-Length Coding (VLC)

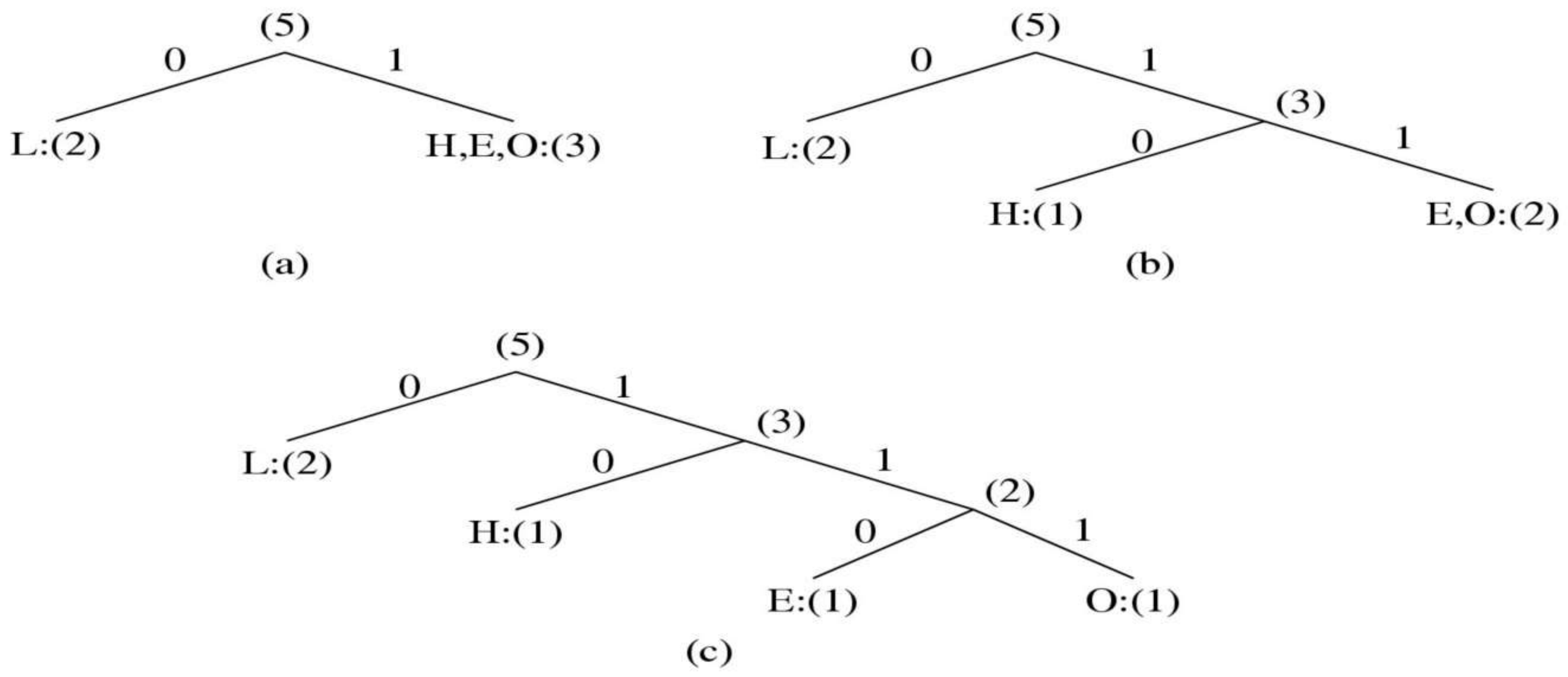
- **7.4.1 Shannon-Fano Algorithm**

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.

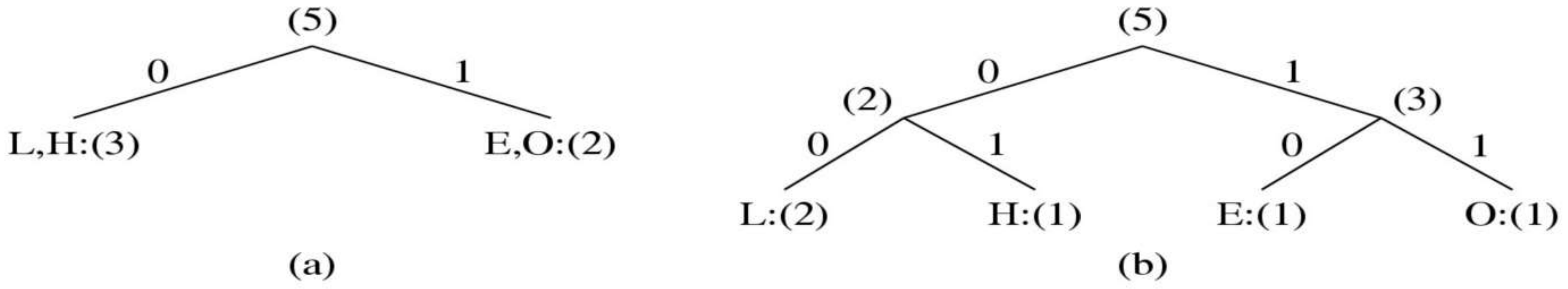
- **An Example: coding of “HELLO”**

Symbol	H	E	L	O
Count	1	1	2	1

- Frequency count of the symbols in ”HELLO”.



- **Fig. 7.3**: Coding Tree for HELLO by Shannon-Fano.



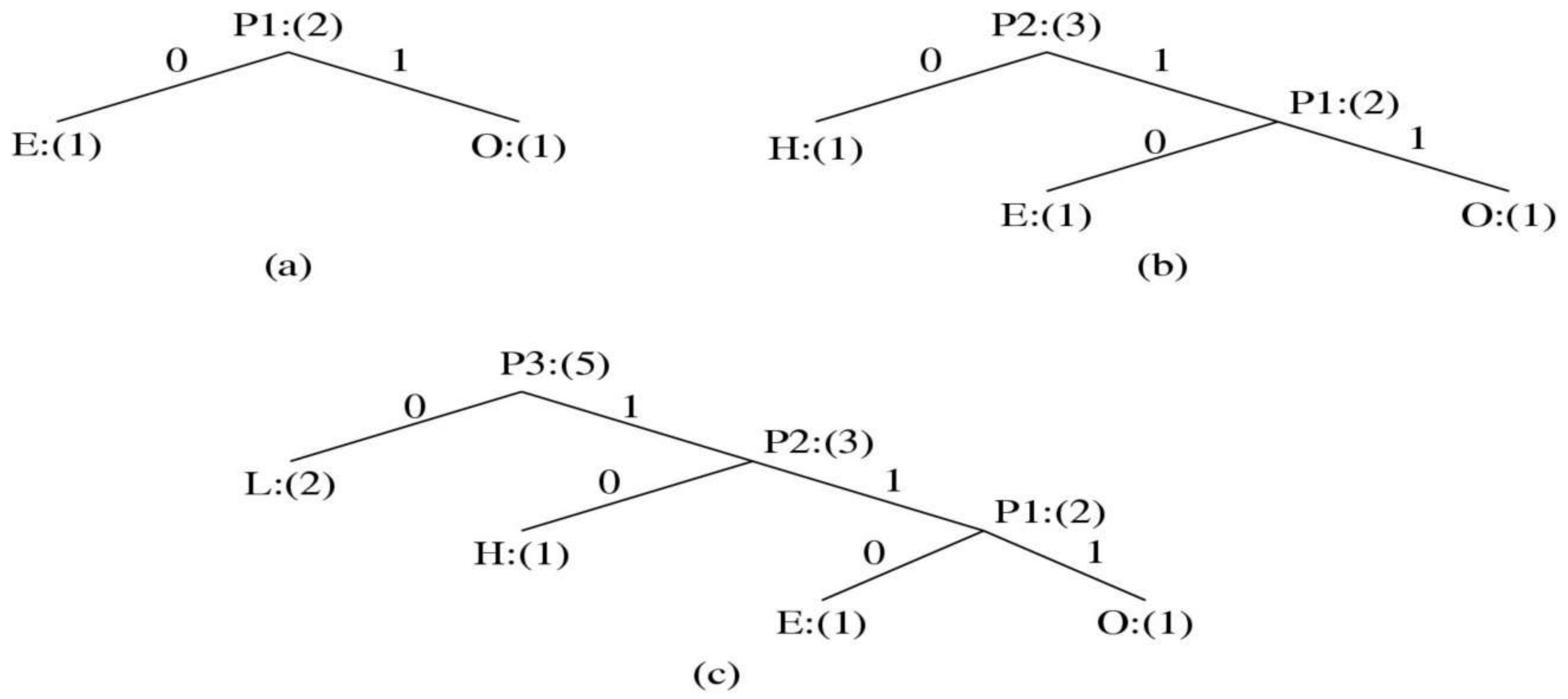
- **Fig. 7.4**: Another coding tree for HELLO by Shannon-Fano.

7.4.2 Huffman Coding

- **ALGORITHM 7.1 Huffman Coding Algorithm**
- — a bottom-up approach

- 1. Initialization: Put all symbols on a list sorted according to their frequency counts.
- 2. Repeat until the list has only one symbol left:
  1) 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.
  2) Assign the sum of the children's frequency counts to the parent and insert it into the list such that the order is maintained.
  3) Delete the children from the list.

– 3. Assign a codeword for each leaf based on the path from the root.



- **Fig. 7.5**: Coding Tree for "HELLO" using the Huffman Algorithm.

Huffman Coding (Cont’d)

•In Fig. 7.5, 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			

Extended Huffman Coding

- **Motivation**: All codewords in Huffman coding have integer bit lengths. It is wasteful when p_i is very large and hence is close to 0. $\log_2 \frac{1}{p_i}$
- Why not group several symbols together and assign a single codeword to the group as a whole?
- **Extended Alphabet**: For alphabet $S = \{s_1, s_2, \ldots, s_n\}$, if k symbols are grouped together, then the *extended alphabet* is:

$$S^{(k)} = \{\overbrace{s_1 s_1 \text{¼} s_1}^{k\ \ symbols}, s_1 s_1 \text{¼} s_2, \text{¼}, s_1 s_1 \text{¼} s_n, s_1 s_1 \text{¼} s_2 s_1, \text{¼}, s_n s_n \text{¼} s_n\}.$$

- — the size of the new alphabet $S^{(k)}$ is n^k.

Tree is as below:

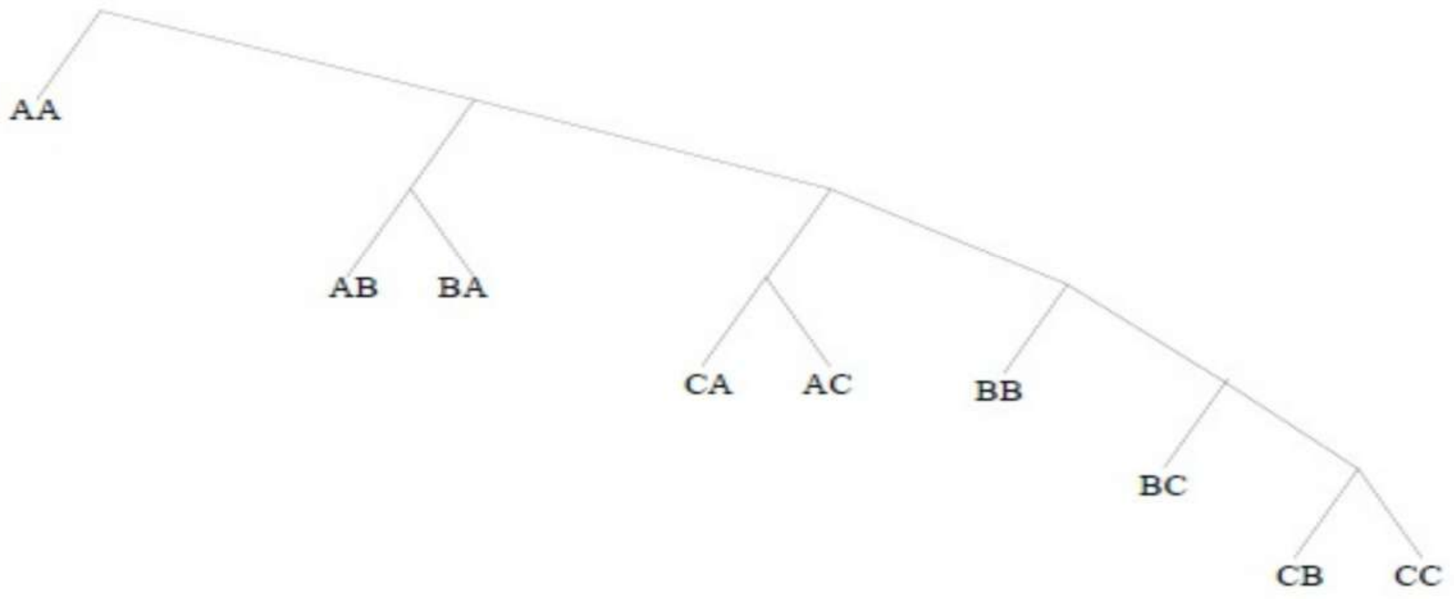


Fig. 7.6: Huffman Tree for Extended Alphabet

Codeword bitlengths are:

Symbol group	Probabililty	Codeword	Bitlength
AA	.36	0	1
AB	.18	100	3
BA	.18	101	3
CA	.06	1100	4
AC	.06	1101	4
BB	.09	1110	4
BC	.03	11110	5
CB	.03	111110	6
CC	.01	111111	6

Average = 0.5*(0.36 + 3x0.18 + 3x0.18 + 4x0.09 + 4x0.06 + 4x0.06 + 5x0.03 + 6x0.03 + 6x0.01)
= 1.3350 (recall, was 1.4 for length-1 symbols).

The best possible is the entropy, given by:
$H = -0.6 \times \log_2 0.6 - 0.3 \times \log_2 0.3 - 0.1 \times \log_2 0.1 \approx 1.2955.$

7.4.3 Adaptive Huffman Coding

- **Adaptive Huffman Coding**: statistics are gathered and updated dynamically as the data stream arrives.

```
ENCODER
-------
Initial_code();

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

```
DECODER
-------
Initial_code();

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

Adaptive Huffman Coding (Cont'd)

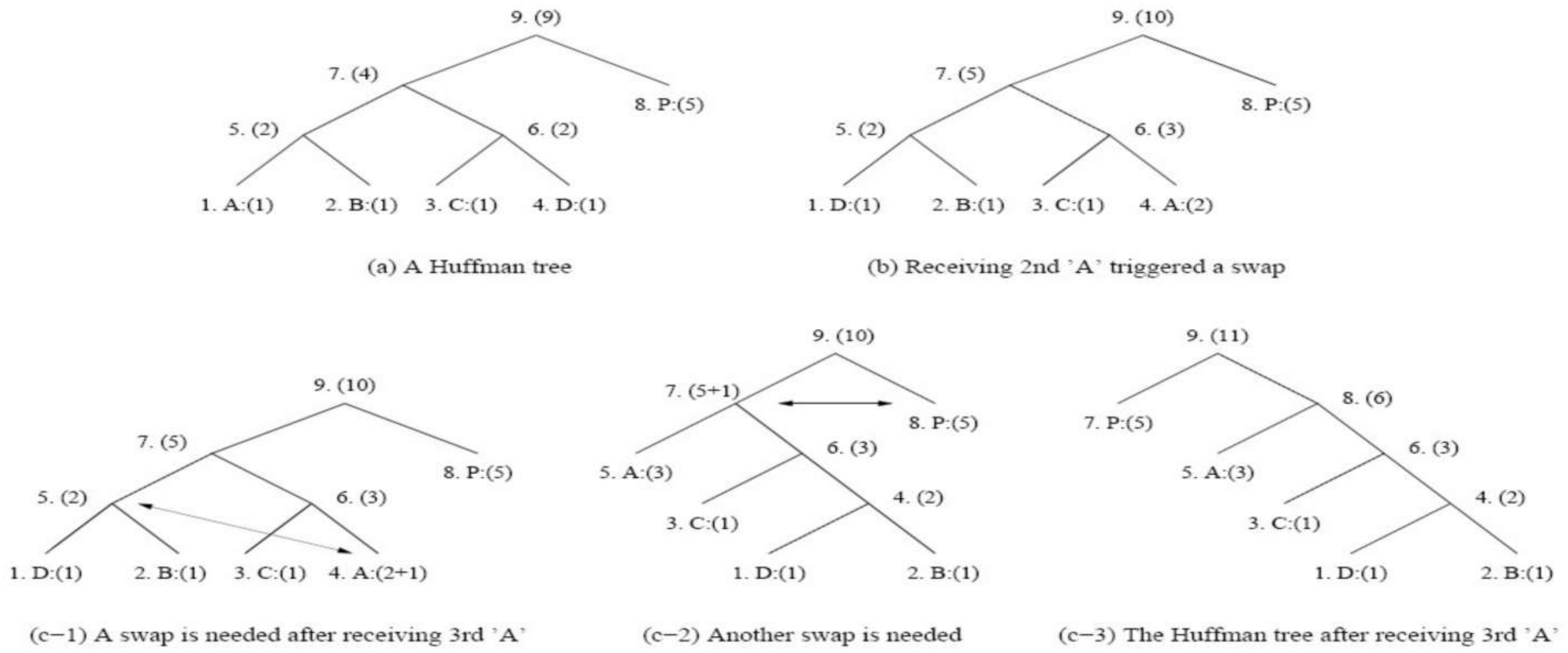
- *Initial_code* assigns symbols with some initially agreed upon codes, without any prior knowledge of the frequency counts.

- *update_tree* constructs an Adaptive Huffman tree.
- It basically does two things:
  a) increments the frequency counts for the symbols (including any new ones).
  b) updates the configuration of the tree.

- The *encoder* and *decoder* must use exactly the same *initial_code* and *update_tree* routines.

Notes on Adaptive Huffman Tree Updating

- Nodes are numbered in order from left to right, bottom to top. The numbers in parentheses indicates the count.
- The tree must always maintain its *sibling* property, i.e., all nodes (internal and leaf) are arranged in the order of increasing counts.
- 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 $N+1$.



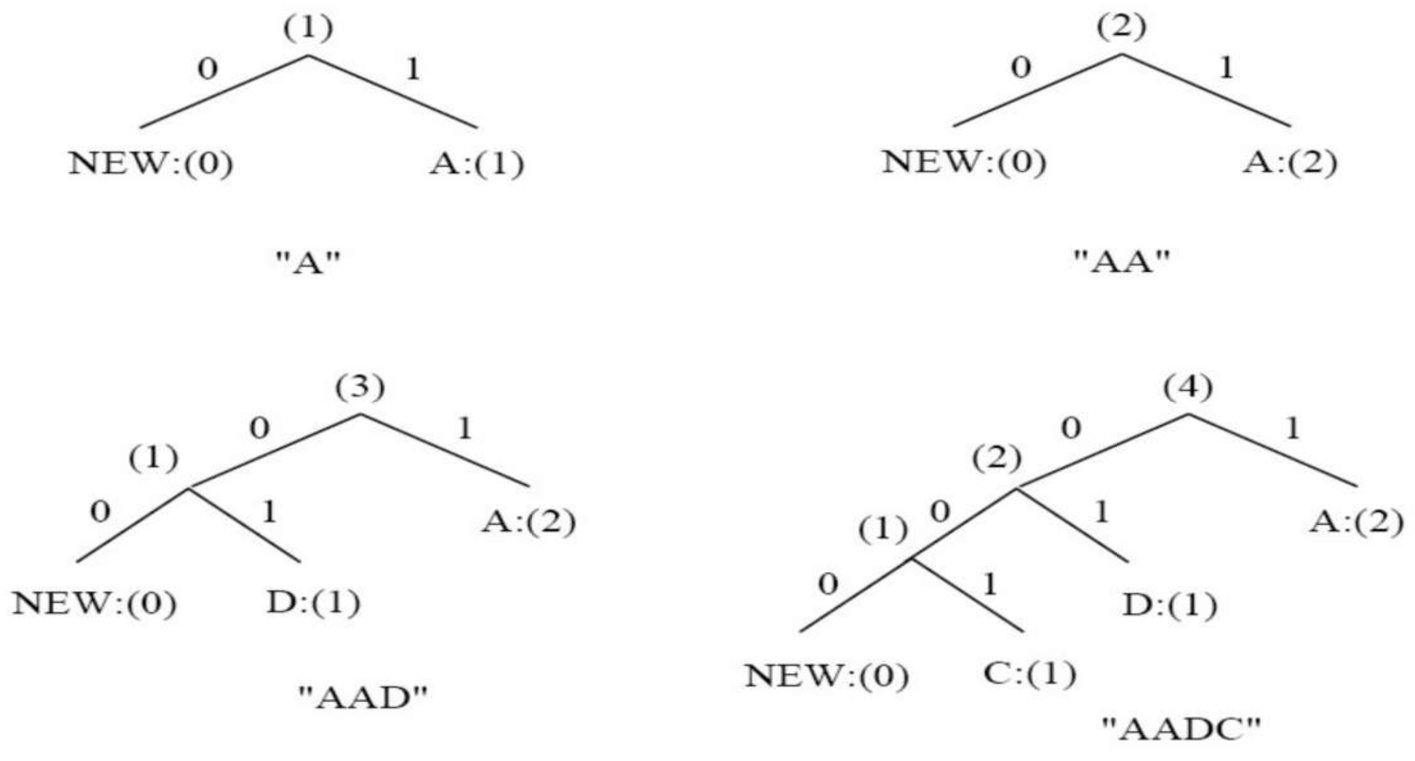
- **Fig. 7.7**: Node Swapping for Updating an Adaptive Huffman Tree

Another Example: Adaptive Huffman Coding

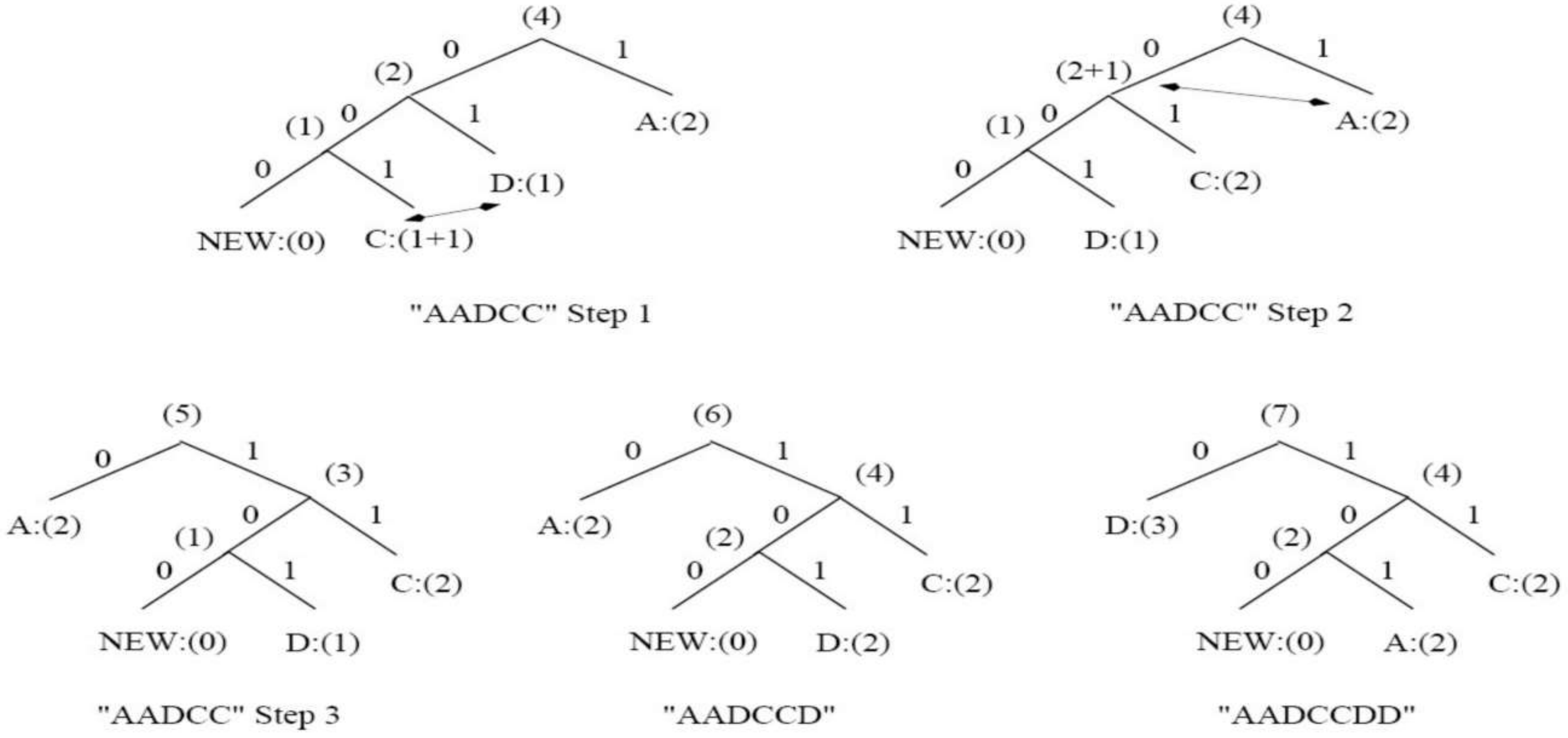
- This is to clearly illustrate more implementation details. We show exactly what *bits* are sent, as opposed to simply stating how the tree is updated.
- An additional rule: if any character/symbol is to be sent the first time, it must be preceded by a special symbol, NEW. The initial code for NEW is 0. The *count* for NEW is always kept as 0 (the count is never increased); hence it is always denoted as NEW:(0) in Fig. 7.7.

- **Table 7.3:** Initial code assignment for AADCCDD using Adaptive Huffman Coding.

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



- **Fig. 7.8**: Adaptive Huffman tree for AADCCDD.



- **Fig. 7.8** (Cont'd): Adaptive Huffman tree for AADCCDD.

- **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 changes during the adaptive Huffman coding process.

  For example, after AADCCDD, when the character D overtakes A as the most frequent symbol, its code changes from 101 to 0.

- The "Squeeze Page" on this book's web site provides a Java applet for adaptive Huffman coding.

7.5 Dictionary-based Coding

- LZW uses fixed-length codewords to represent variable-length strings of symbols/characters that commonly occur together, e.g., words in English text.
- The LZW encoder and decoder build up the same dictionary dynamically while receiving the data.
- LZW places longer and longer repeated entries into a dictionary, and then emits the *code* for an element, rather than the string itself, if the element has already been placed in the dictionary.

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 3 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 sending 14 characters, only 9 codes need to be sent (compression ratio = 14/9 = 1.56).

• 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!

• ALGORITHM 7.4 LZW Decompression (modified)

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

LZW Coding (Cont'd)

- In real applications, the code length l is kept in the range of $[l_0, l_{max}]$. The dictionary initially has a size of 2^{l_0}. When it is filled up, the code length will be increased by 1; this is allowed to repeat until $l = l_{max}$.
- When l_{max} is reached and the dictionary is filled up, it needs to be flushed (as in Unix *compress*, or to have the LRU (least recently used) entries removed.

7.6 Arithmetic Coding

- Arithmetic coding is a more modern coding method that usually out-performs Huffman coding.
- Huffman coding assigns each symbol a codeword which has an integral bit length. Arithmetic coding can treat the whole message as one unit.
- A message is represented by a half-open interval $[a, b)$ where a and b are real numbers between 0 and 1. Initially, the interval is $[0, 1)$. When the message becomes longer, the length of the interval shortens and the number of bits needed to represent the interval increases.

ALGORITHM 7.5 Arithmetic Coding Encoder

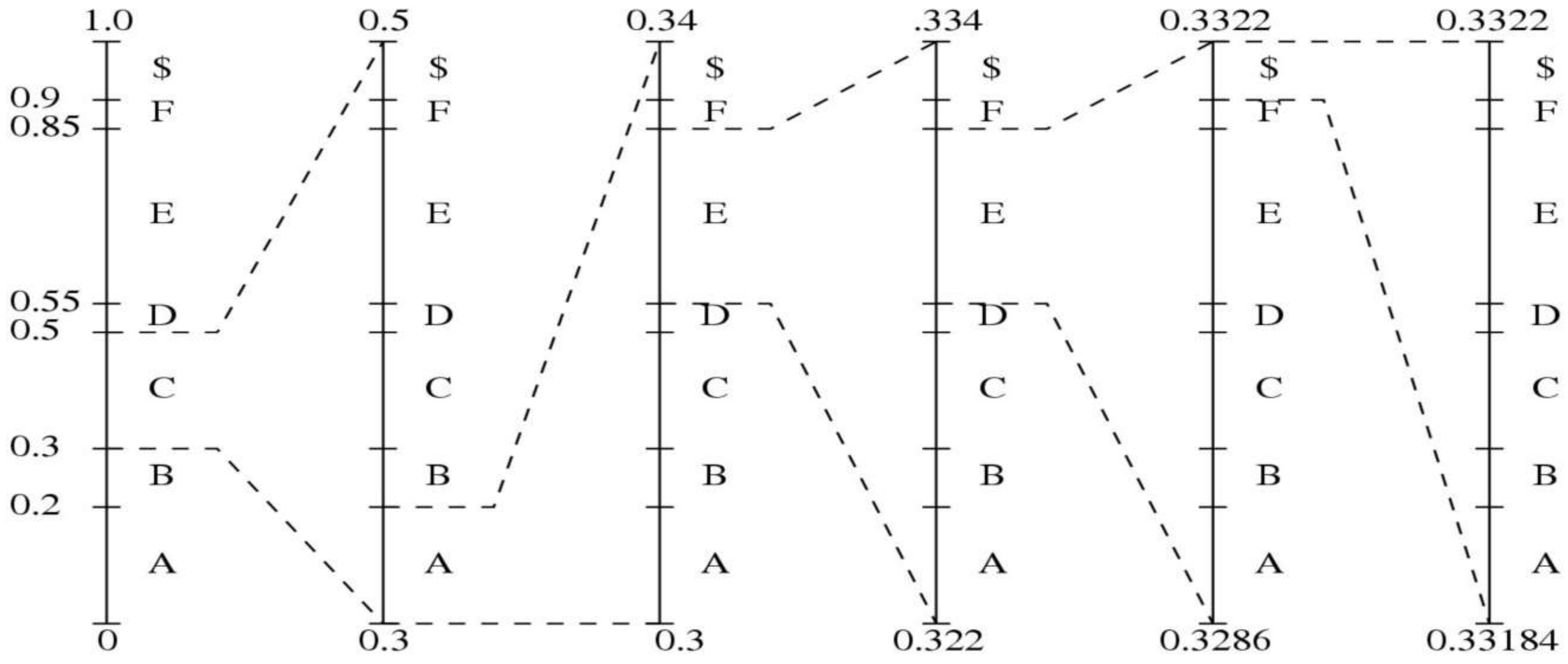
```
BEGIN
    low = 0.0;   high = 1.0;   range = 1.0;

    while (symbol != terminator)
    {
        get (symbol);
        high = low + range * Range_high(symbol);
        low = low + range * Range_low(symbol);
        range = high - low;
    }

    output a code so that low <= code < high;
END
```

• Example: Encoding in Arithmetic Coding

Symbol	Probability	Range
A	0.2	[0, 0.2)
B	0.1	[0.2, 0.3)
C	0.2	[0.3, 0.5)
D	0.05	[0.5, 0.55)
E	0.3	[0.55, 0.85)
F	0.05	[0.85, 0.9)
$	0.1	[0.9, 1.0)

(a) Probability distribution of symbols.

- **Fig. 7.9**: Arithmetic Coding: Encode Symbols "CAEE$"



- **Fig. 7.9(b):** Graphical display of shrinking ranges.

Example: Encoding in Arithmetic Coding

Symbol	Low	High	Range
	0	1.0	1.0
C	0.3	0.5	0.2
A	0.30	0.34	0.04
E	0.322	0.334	0.012
E	0.3286	0.3322	0.0036
$	0.33184	0.33220	0.00036

- (c) New *low*, *high*, and *range* generated.

- **Fig. 7.9 (Cont'd):** Arithmetic Coding: Encode Symbols "CAEE$"

PROCEDURE 7.2 Generating Codeword for Encoder

```
BEGIN
  code = 0;
  k = 1;
  while (value(code) < low)
  {
     assign 1 to the kth binary fraction bit
     if (value(code) > high)
           replace the kth bit by 0
     k = k + 1;
  }
END
```

- The final step in Arithmetic encoding calls for the generation of a number that falls within the range [*low*, *high*). The above algorithm will ensure that the shortest binary codeword is found.

ALGORITHM 7.6 Arithmetic Coding Decoder

```
• BEGIN
  – get binary code and convert to
  – decimal value = value(code);
  – DO
  – {
  –  find a symbol s so that
  –        Range_low(s) <= value < Range_high(s);
  –  output s;
  –  low = Rang_low(s);
  –  high = Range_high(s);
  –  range = high - low;
  –  value = [value - low] / range;
  – }
  – UNTIL symbol s is a terminator
• END
```

- **Table 7.5:** Arithmetic coding: decode symbols “CAEE$”

Value	Output Symbol	Range_low	Range_high	range
0.33203125	C	0.3	0.5	0.2
0.16015625	A	0.0	0.2	0.2
0.80078125	E	0.55	0.85	0.3
0.8359375	E	0.55	0.85	0.3
0.953125	$	0.9	1.0	0.1

7.6.2 Scaling and Incremental Coding

- The basic algorithm described in the last section has the following *limitations* that make its practical implementation infeasible.
- When it is used to code long sequences of symbols, the tag intervals shrink to a very small range. Representing these small intervals requires very high-precision numbers.
- The encoder will not produce any output codeword until the entire sequence is entered. Likewise, the decoder needs to have the codeword for the entire sequence of the input symbols before decoding.

- **Some key observations:**

1. Although the binary representations for the *low*, *high*, and any number within the small interval usually require many bits, they always have the same MSBs (Most Significant Bits). For example, 0.1000110 for 0.5469 (low), 0.1000111 for 0.5547 (high).

2. Subsequent intervals will always stay within the current interval. Hence, we can output the common MSBs and remove them from subsequent considerations.

THANK YOU