Lecture 6

Lossless Compression

Multimedia System

Spring 2020

Introduction

- **Compression**: the process of coding that will effectively reduce the total number of bits needed to represent certain information.
- If the compression and decompression processes induce no information loss, then the compression scheme is **lossless**; otherwise, it is **lossy**.
- **Compression ratio**:

$$compression\ ratio = \frac{B_0}{B_1}$$

B_0 – number of bits before compression

B_1 – number of bits after compression

Basics of Information Theory

- The *entropy* of an information *symbols* with alphabet $S = \{s_1, s_2, \ldots, s_n\}$ is:

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

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

p_i *:* probability that symbol s_i will occur in S.

$\log_2 \frac{1}{p_i}$: 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.

Distribution of Gray-Level Intensities

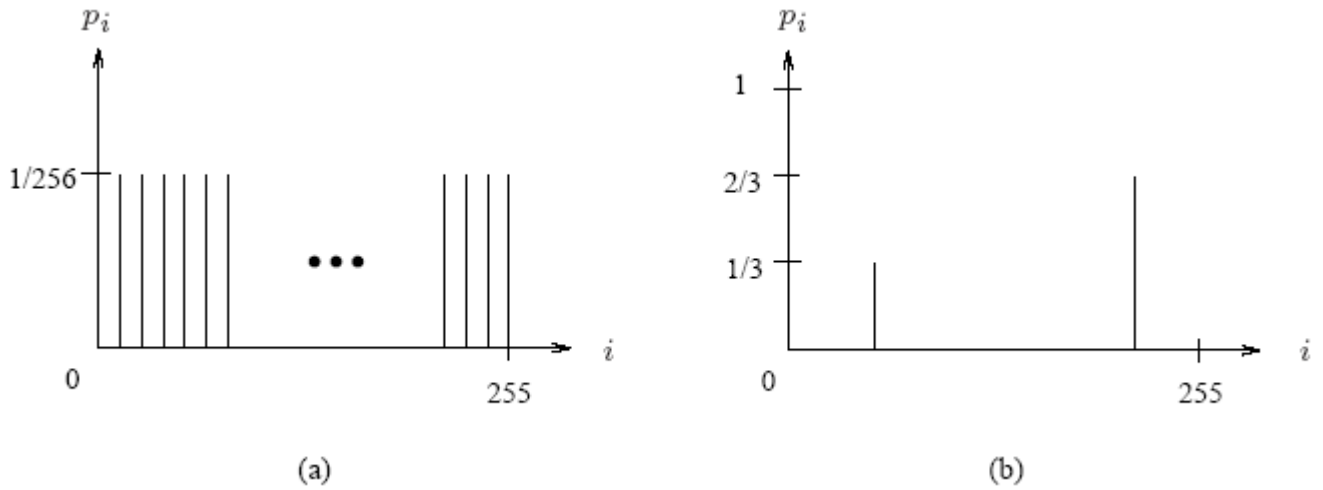


Fig. 1. Histograms for Two Gray-level Images.

- Fig. 1(a) shows the histogram of an image with *uniform* distribution of gray-level intensities, i.e., $p_i = 1/256 \quad \forall i$
  Hence, the entropy of this image is:
  $$\log_2 256 = 8$$
- Fig 1(b) : $\eta \sim 0.908$

Entropy and Code Length

- As can be seen in Eq. (1): the entropy is a weighted-sum of terms $\log_2 \frac{1}{p_i}$; hence it represents the *average* amount of information contained per symbol in the source *S*.
- The entropy η specifies the lower bound for the average number of bits to code each symbol in *S*, i.e.,

$$\eta \leq \bar{l} \qquad (2)$$

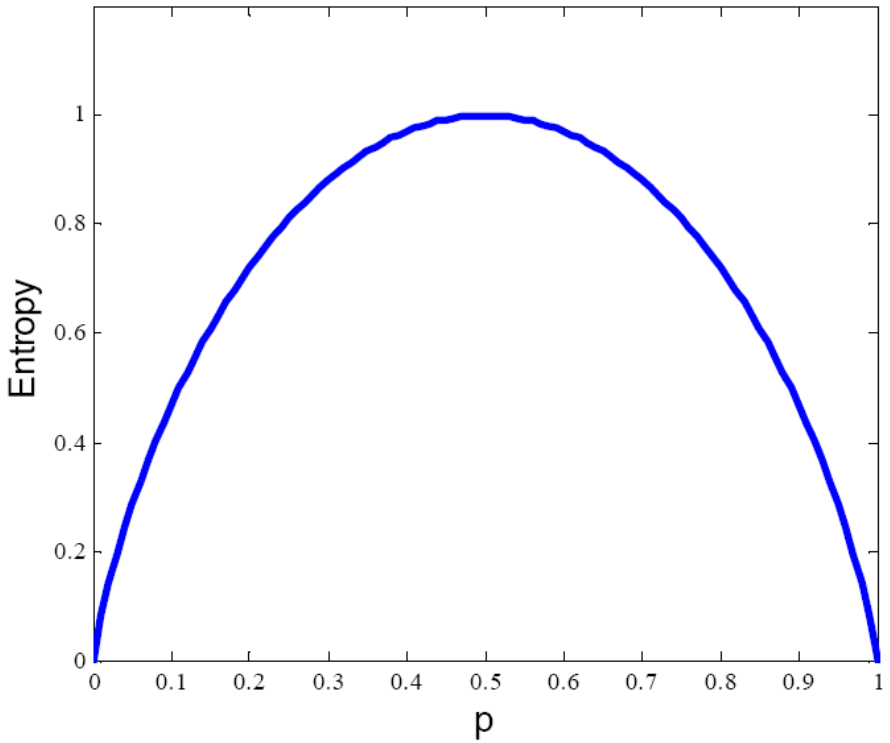
$\bar{l}$: the average length (measured in bits) of the codewords produced by the encoder.

Entropy example

- Alphabet set {0, 1}
- Probability: {p, 1−p}
- Entropy:

$\eta = -\ p \log2\ p - (1-p) \log2\ (1-p)$

  - when $p=0$, $\eta = 0$
  - when $p=1$, $\eta = 0$
  - when $p=1/2$, $\eta_{max} = 1$
    - 1 bit is enough!



Symbol Encoding: Shannon-Fano Algorithm

- A top-down approach
  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".

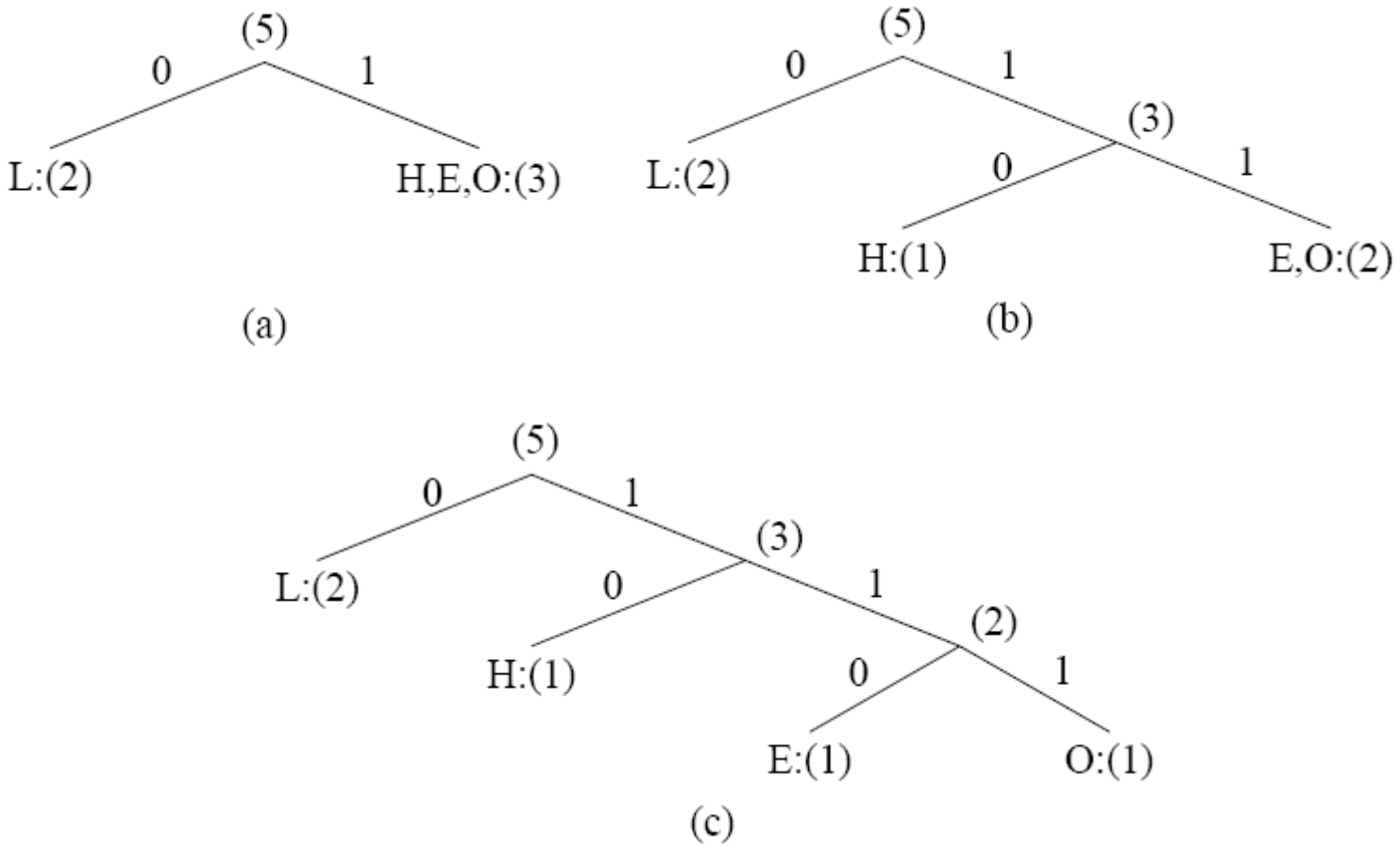


Fig. 2: Coding Tree for HELLO by Shannon-Fano.

Table 1: Result of Performing Shannon-Fano on HELLO

Symbol	Count	$\log_2 \frac{1}{p_i}$	Code	# 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

Average: 2.5 bits

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

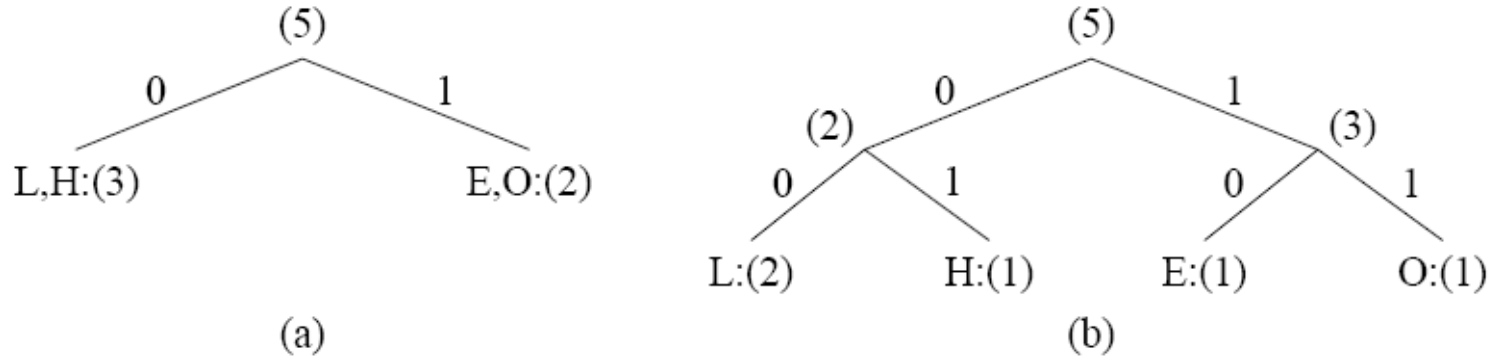


Figure 3: Another Result of Performing Shannon-Fano on HELLO (see Fig. 2)

Table. 2: Another coding tree for HELLO by Shannon-Fano.

Symbol	Count	$\log_2 \frac{1}{p_i}$	Code	# 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

Huffman Coding

- **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.

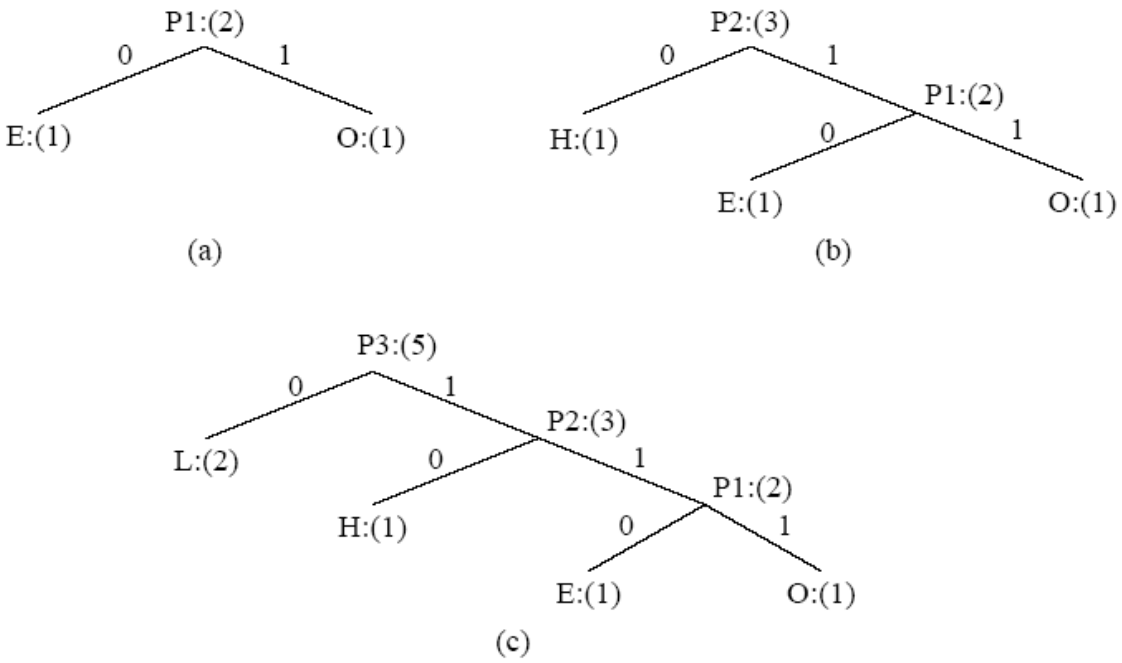
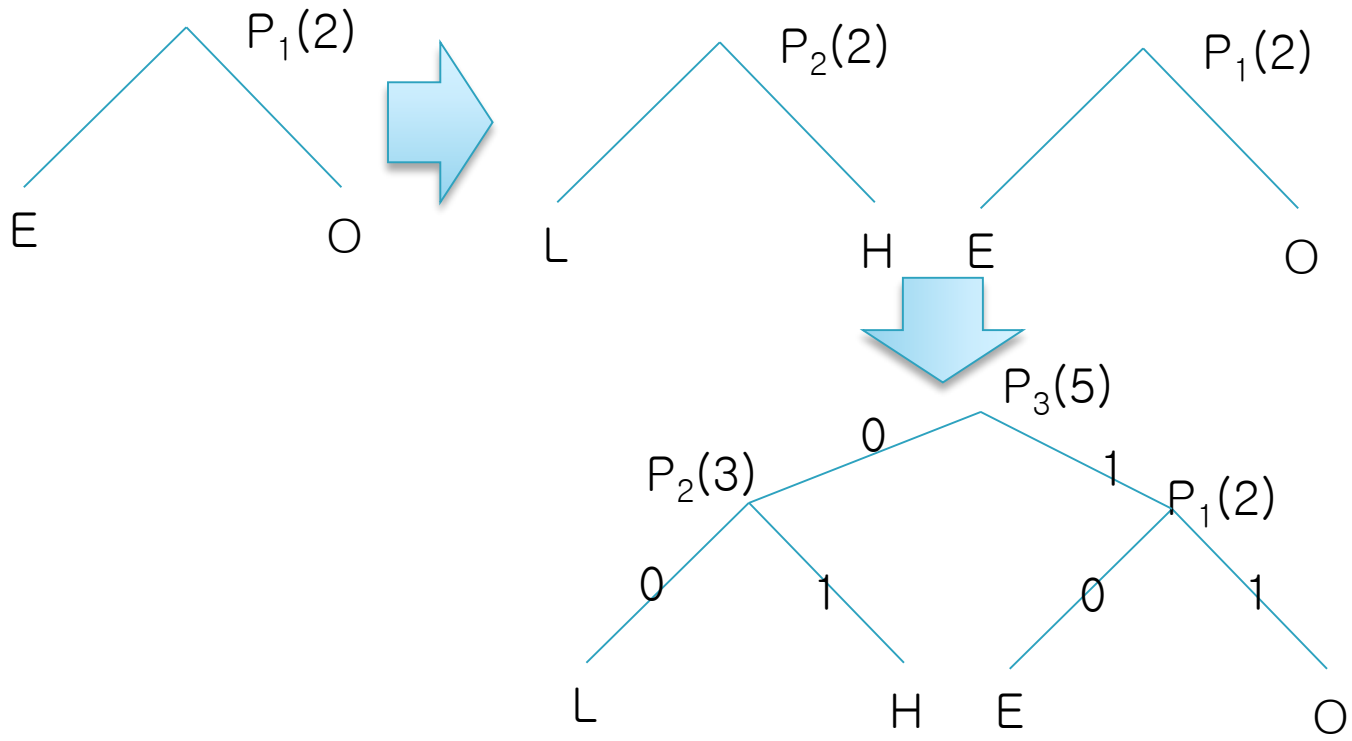


Fig. 4: Coding Tree for "HELLO" using the Huffman Algorithm.

- In Fig. 4, 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 P_1 H
  - After iteration (b): L P_2
  - After iteration (c): P_3

E: 110 O: 111 H: 10 L: 0 (# of bits used : 10 bits)

- Another Implementation
  - After initialization: L H E O
  - After iteration (a): P_1 L H
  - After iteration (b): P_1 P_2
  - After iteration (c): P_3



E: 10
O: 11
H: 01
L: 00
(# of bits used : 10 bits)

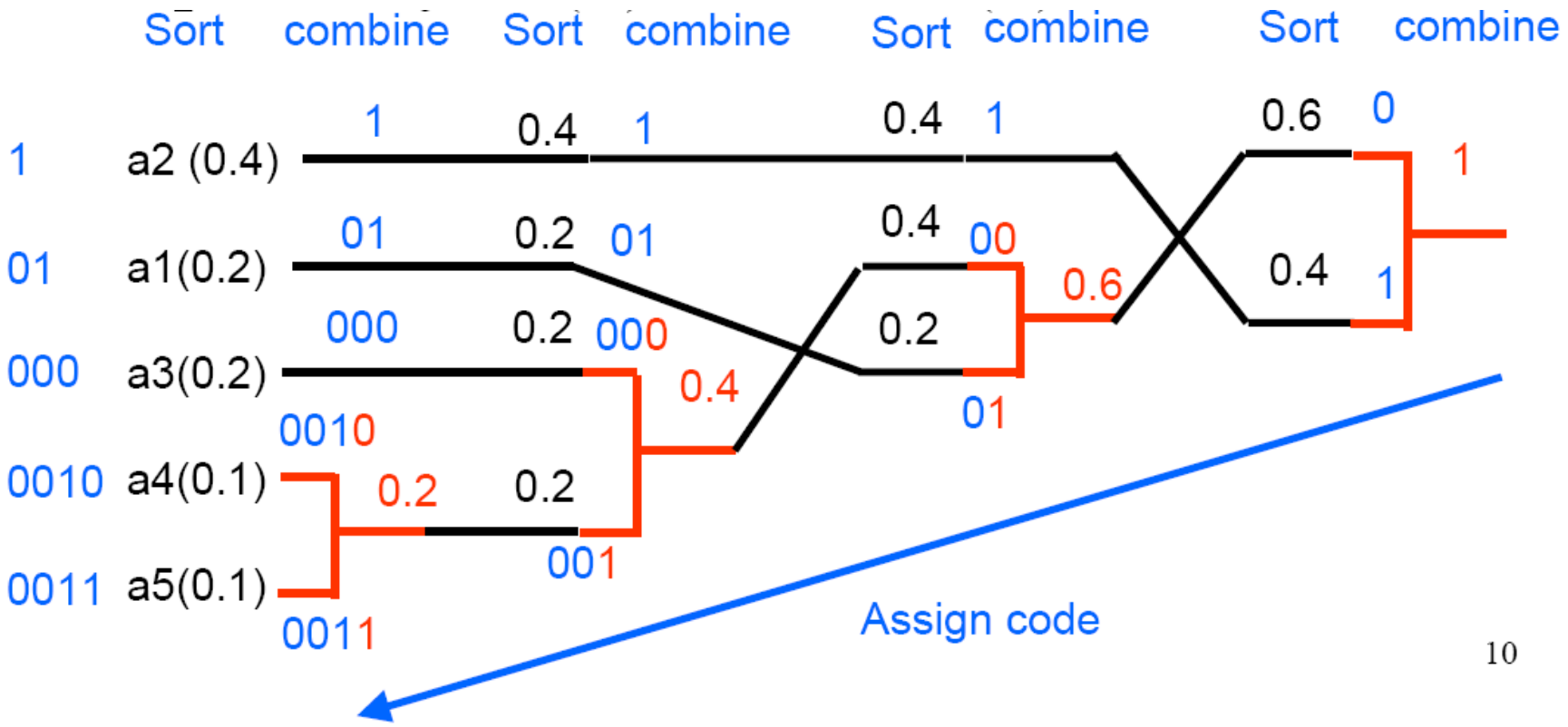
Properties of Huffman Coding

1. **Unique Prefix Property**: No Huffman code is a prefix of any other Huffman code – precludes any ambiguity in decoding.

2. **Optimality**: *minimum redundancy code* – proved *optimal* for a given data model (i.e., a given, accurate, probability distribution):
   - Symbols that occur more frequently will have shorter Huffman codes than symbols that occur less frequently.
   - The average code length for an information source *S* is strictly less than $\eta + 1$. Combined with Eq. (3), we have:

$$\eta \leq \bar{l} < \eta + 1 \qquad (3)$$

Huffman Coding example



Adaptive Huffman Coding

- **Adaptive Huffman Coding:** symbol counts are 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

- *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.

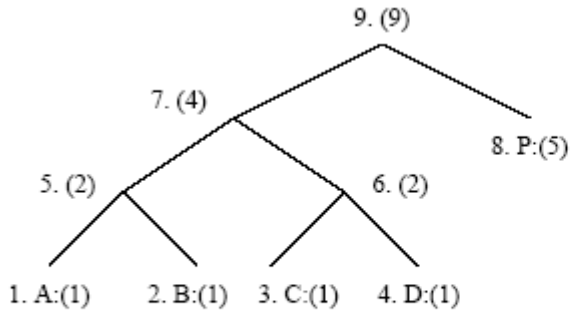
- The *encoder* and *decoder* must use exactly the same *initial_code* and *update_tree* routines.

Notes on Adaptive Huffman Tree Updating

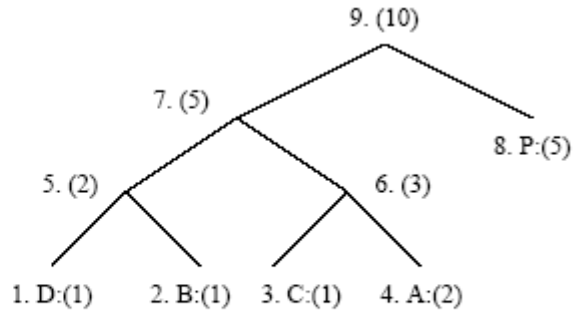
- Nodes are numbered in order from left to right, bottom to top. The numbers in parentheses indicates the count.

- 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.

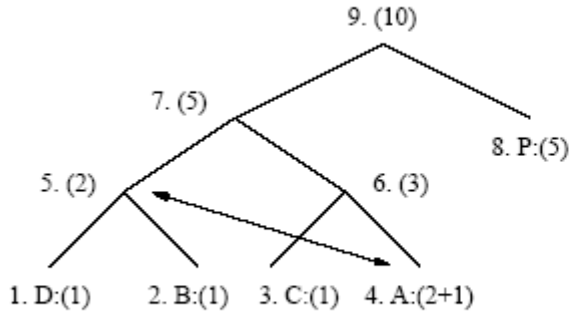
- When a swap is necessary, the farthest node with count N is swapped with the node whose count has just been increased to $N+1$.



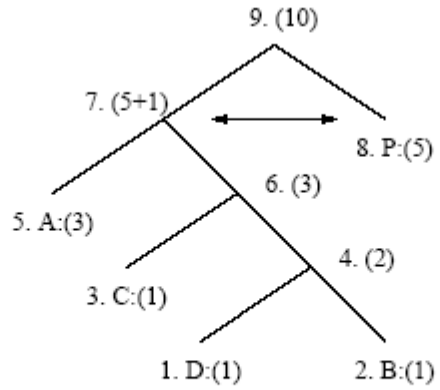
(a) A Huffman tree



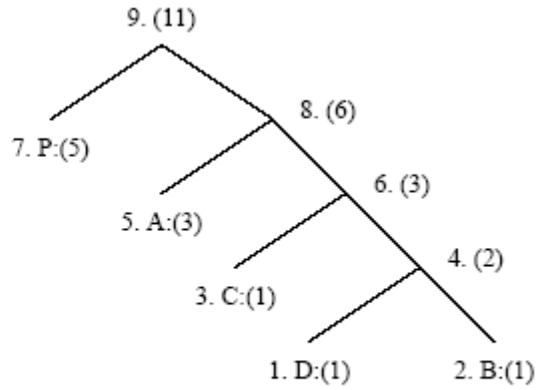
(b) Receiving 2nd 'A' triggered a swap



(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'

Fig. 5: Node Swapping for Updating an Adaptive Huffman Tree.

Practical Example: Adaptive Huffman Coding

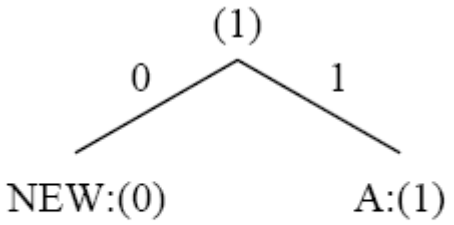
- This is to clearly illustrate more implementation details. We show exactly what *bits* are sent, as opposed to simply stating how the tree is updated.

- 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. 6.

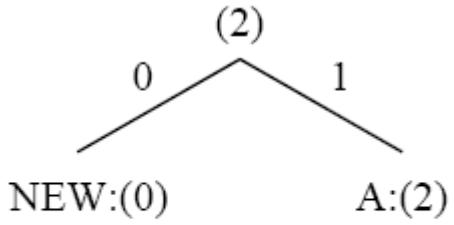
Figure 6: Initial code assignment for AADCCDD using adaptive Huffman coding.

Initial Code

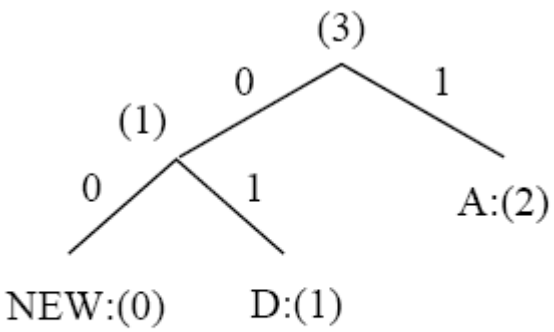
NEW:	0
A:	00001
B:	00010
C:	00011
D:	00100
.	.
.	.
.	.



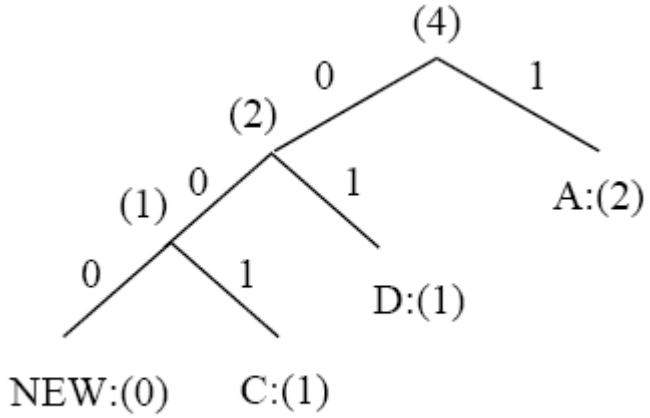
"A"



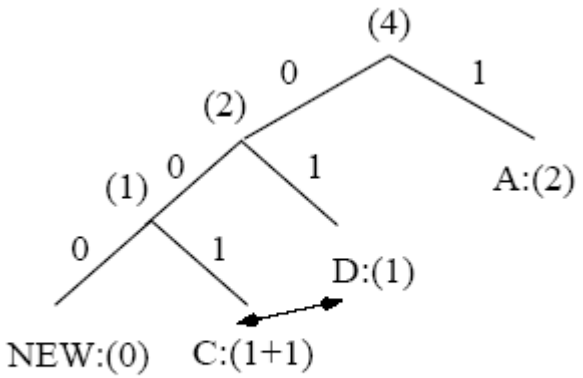
"AA"



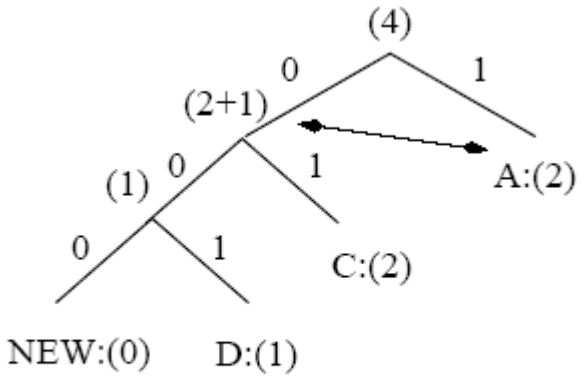
"AAD"



"AADC"



"AADCC" Step 1



"AADCC" Step 2

Fig. 7 Adaptive Huffman tree for AADCCDD.

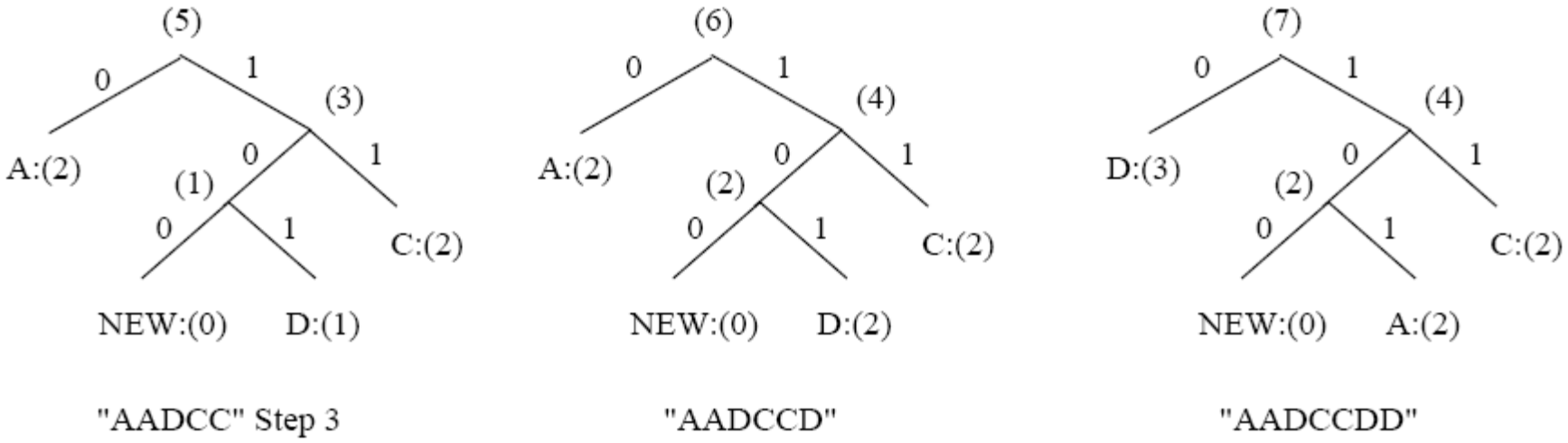


Fig. 7 (cont'd) Adaptive Huffman tree for AADCCDD.

Table 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

Dictionary-based Coding

- LZW (Lempel-ziv-Welch) 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.

LZW Compression

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

      if s + c exists in the dictionary
            s = s + c;  //append to characters
      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 “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:

Example 7.2: LZW compression "ABABBABCABABBA"

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		

```
BEGIN
  s = next input character;
  while not EOF
    { c = next input character;
      if s + c exists in the dictionary
          s = s + c;  //append to characters
      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

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		

- "ABABBABCABABBA"
- 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).

LZW Decompression

```
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 Decompression

- 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			

```
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
```

- The output string is "ABABBABCABABBA", a truly lossless result!

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.