# Multimedia Information Retrieval and Technology

# Lecture 19
# Compression Algorithms II

By : Laura Liu

Room: EE314

Tel. no. 7756

Xi'an Jiaotong-Liverpool University
西交利物浦大学

1. Introduction
2. Variable-Length Coding (VLC)
   - Shannon-Fano Algorithm
   - Huffman Coding Algorithm
   - **Adaptive Huffman Coding Algorithm**
3. Basics of Information Theory
4. **LZW Compression**

# Adaptive Huffman Coding

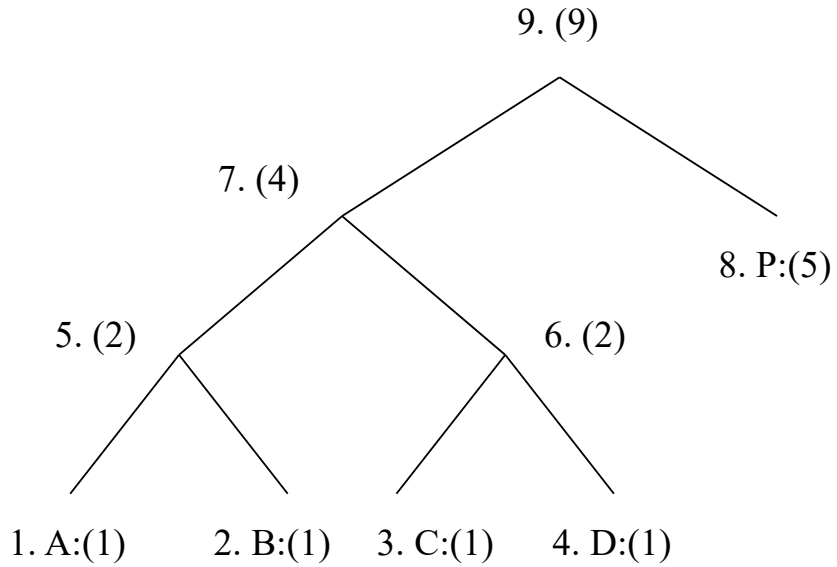
- The Huffman algorithm requires prior statistical knowledge about the information source and such information is often not available.

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

# Adaptive Huffman Coding (Cont'd)

- The Huffman tree must maintain its **sibling property** - that is, all nodes (internal and leaf) are arranged in the order of increasing counts.
- Nodes are arranged in increaing order **from left to right**, **bottom to top.**
- The numbers in parentheses indicates the count.

Figure 7.6(a) depicts a Huffman tree with some symbols already received. In which the first node is 1.A:(1), the second node is 2.B:(1), and so on.



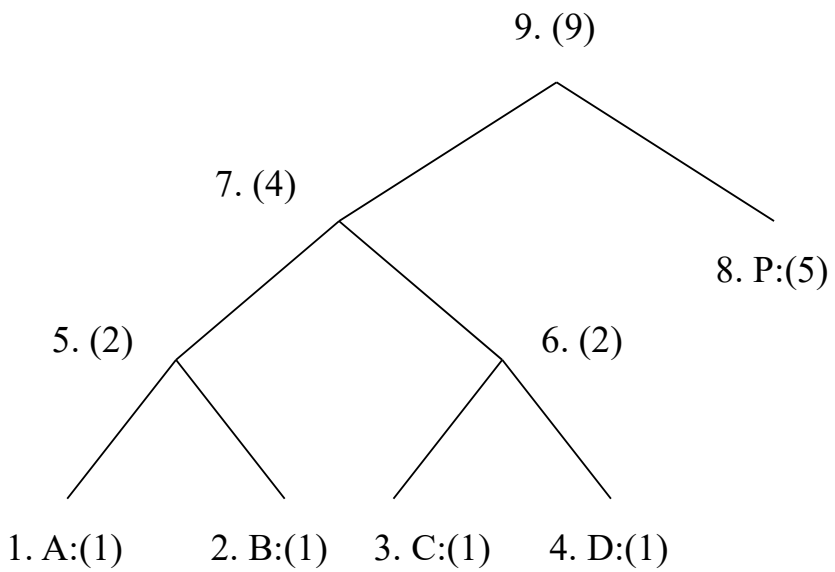

(a) A Huffman tree

Fig. 7.6: Node Swapping for Updating an Adaptive Huffman Tree

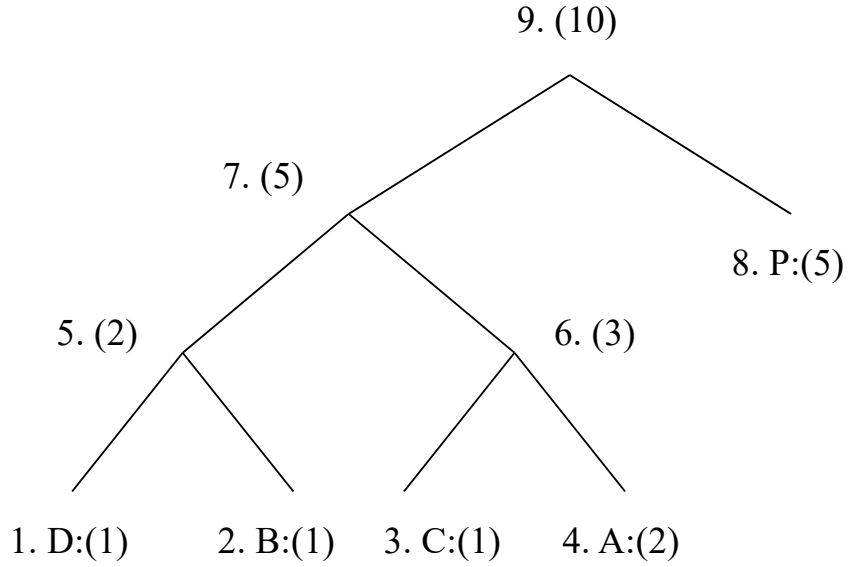
# Adaptive Huffman Coding (Cont'd)

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.
- If the node with count *N* is not a leaf-node (the root of a sub-tree) **the entire sub-tree** will go with it during the swap.




(a) A Huffman tree




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

Fig. 7.6: Node Swapping for Updating an Adaptive Huffman Tree

# Adaptive Huffman Coding (Cont'd)

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.

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

# Adaptive Huffman Coding (Cont'd)

What if the third A arrived？

# EXAMPLE
# Adaptive Huffman Coding for AADCCDD

Here we show exactly **what *bits* are sent**.

- 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);

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

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

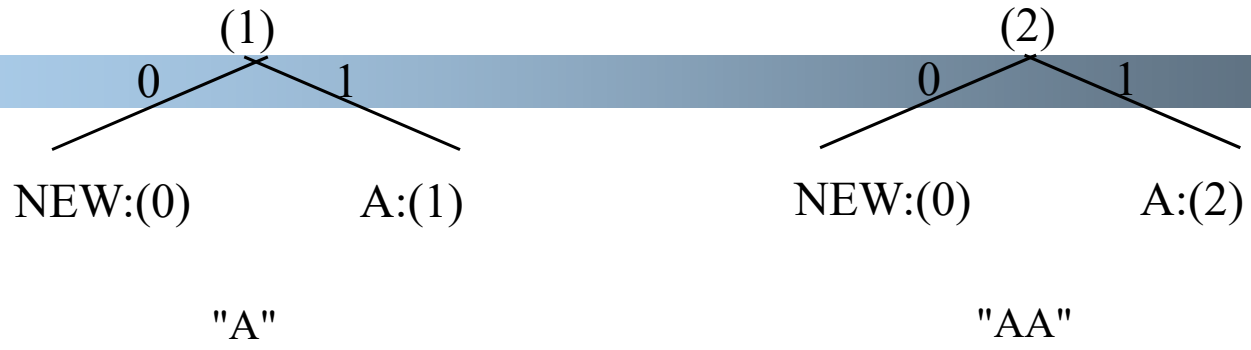



Fig. 7.7 Adaptive Huffman tree for AADCCDD.

Figure 7.7 shows the Huffman tree after each step.

**Step1**: Initially, there is no tree.

**Step2**: 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.

**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 | | | | | | | |

**Step 3**: 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.

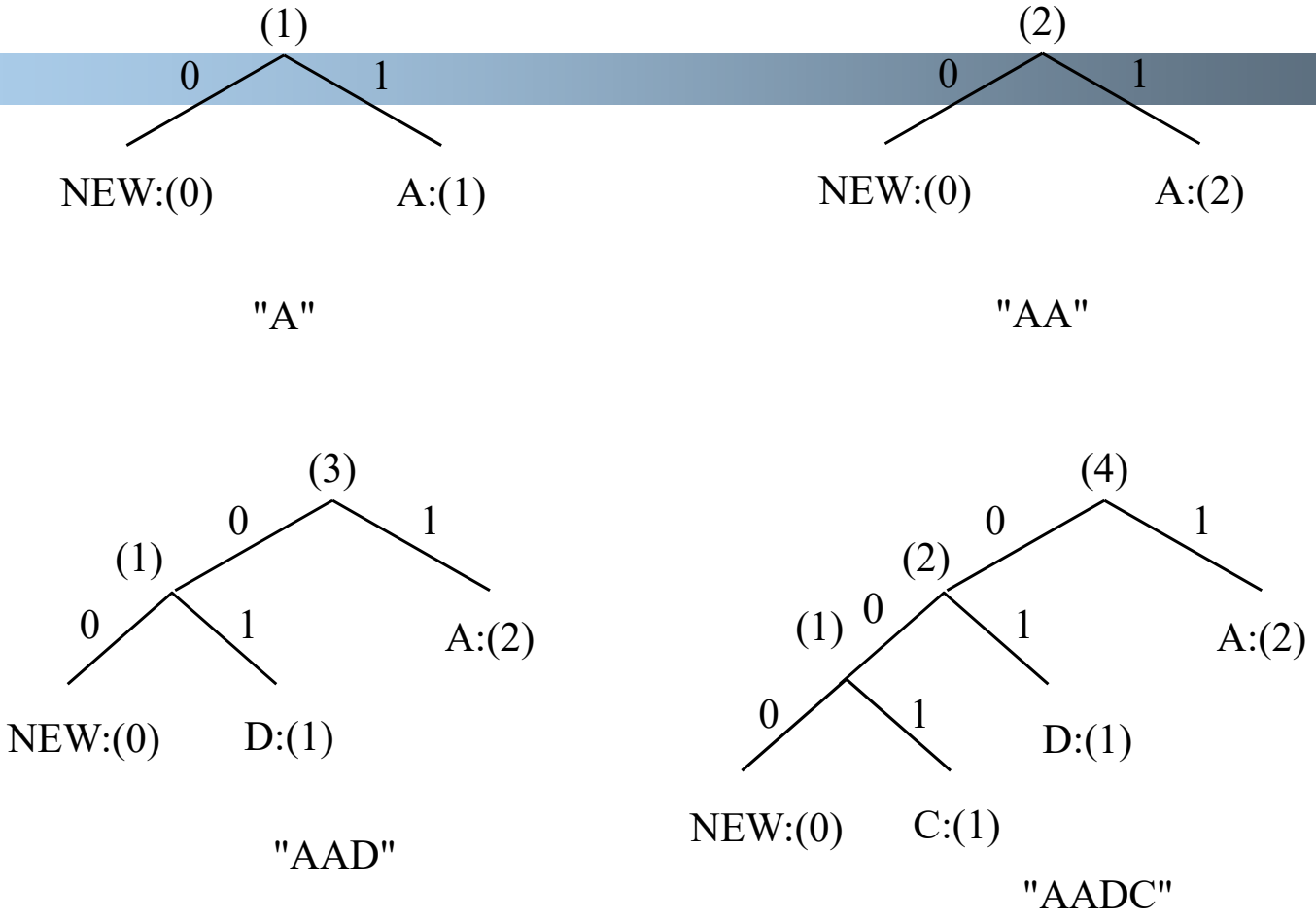



Fig. 7.7 Adaptive Huffman tree for AADCCDD.

# Adaptive Huffman Coding for AADCCDD

**Step 4:** After the second A, the tree is updated, shown labeled as AA.

**Step 5:** 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 7.4 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 | | | |

1. Show the Adaptive Huffman tree for AADCC、AADCCD、AADCCDD

2. Fill up the codes sent to the decoder in table 7.4

# 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);
  }
```

# Adaptive Huffman Coding (Cont'd)

- *Initial_code* assigns symbols with some initial agreement 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.

# Exercise I

Assume that Adaptive Huffman Coding is used to code an information source *S* with a vocabulary of four letters (a, b, c, d). Before any transmission, the initial coding is a = 00, b = 01, c = 10, d = 11. A special symbol NEW will be sent before any letter if it is to be sent the first time.

Figure 7.11 is the Adaptive Huffman tree after sending letters *aabb*. After that, the additional bitstream received by the decoder for the next few letters is 01010010101.

What are the additional letters received?

Draw the adaptive Huffman trees after each of the additional letters is received.

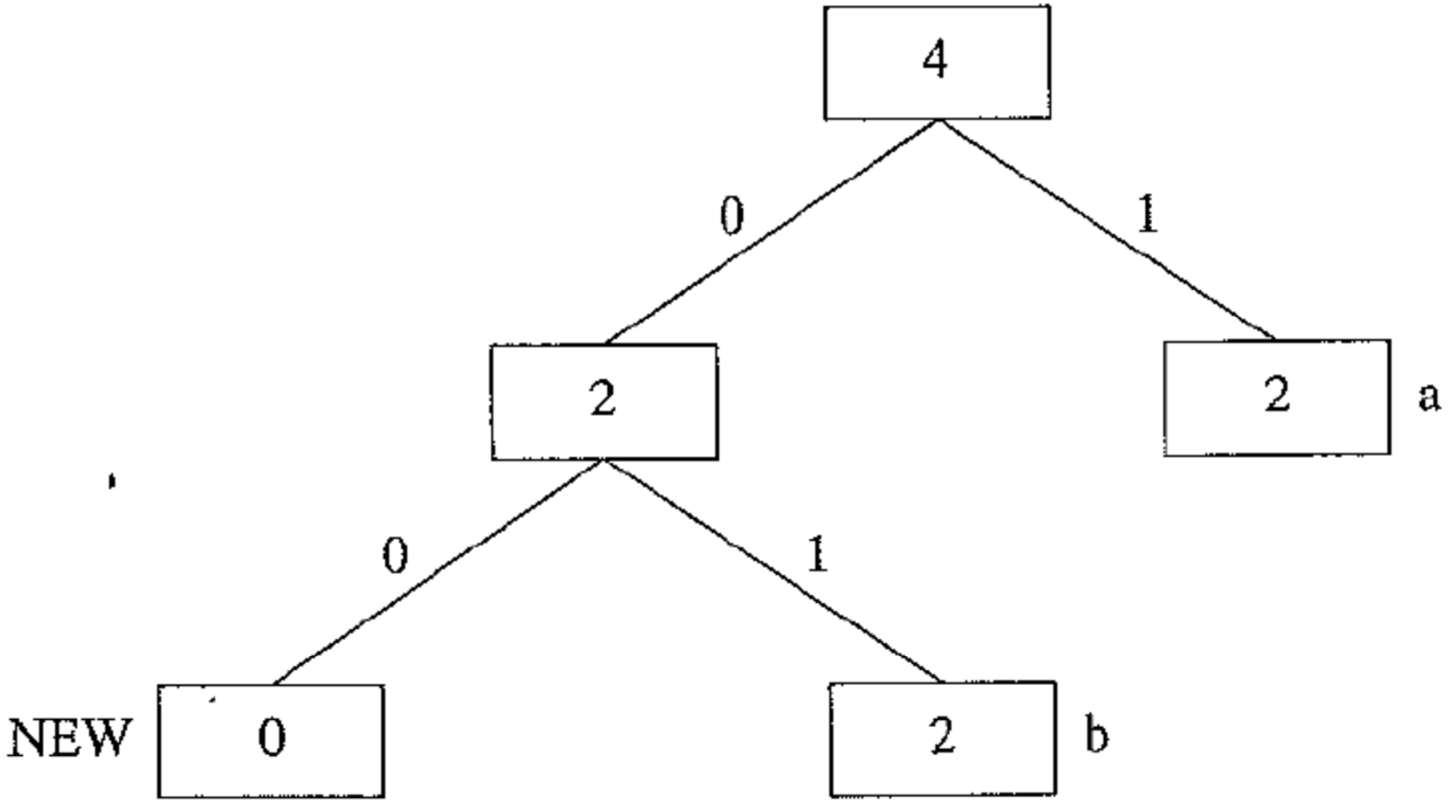



FIGURE 7.11: Adaptive Huffman tree.

1. Introduction
2. Variable-Length Coding (VLC)
   - **Shannon-Fano Algorithm**
   - **Huffman Coding Algorithm**
   - **Adaptive Huffman Coding Algorithm**
3. Basics of Information Theory
4. LZW Compression
   - Encoder
   - Decoder

# LZW 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 Coding

- LZW places longer and longer repeated entries (an element) 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 algorithm is an **adaptive**, **dictionary based** compression technique.

# LZW Coding

**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 "ABABBAB-CABABBA"

- 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 |

What's next?

# LZW Coding

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, the compression ratio would be 14/9 =1.56.

The LZW is an adaptive algorithm, in which the encoder and decoder independently build their own string tables.

No overhead involving transmitting the string table.

# LZW Coding

The LZW assigns fixed length code words to variable length sequences of source symbols.

Integrated into a variety of mainstream imaging file formats.

1. Introduction
2. Variable-Length Coding (VLC)
   - **Shannon-Fano Algorithm**
   - **Huffman Coding Algorithm**
   - **Adaptive Huffman Coding Algorithm**
3. Basics of Information Theory
4. LZW Compression
   - Encoder
   - Decoder

# LZW DECOMPRESSION

```
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 | | |