Chapter 3
Multimedia Data Compression

3.1 Lossless and Lossy compression

3.2 Entropy coding

3.3 Huffman coding

3.4 Adaptive coding

3.5 Dictionary-based coding (LZW)

3.1 Lossless and Lossy compression

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

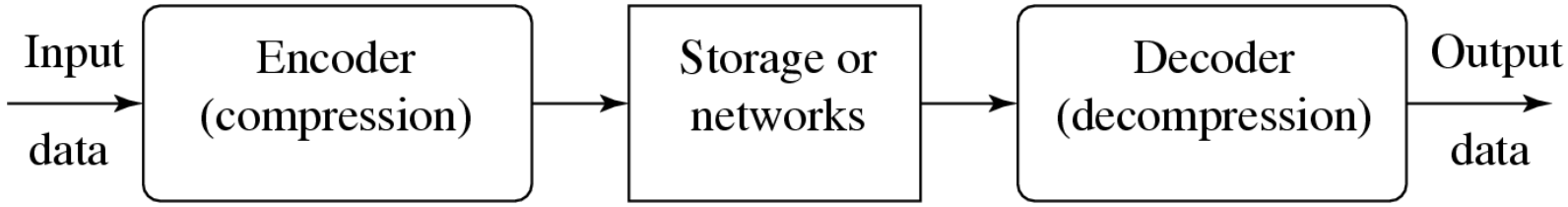


Fig 3.1 A general data compression scheme

- We call the output of the encoder *codes* or *codewords*.
- The intermediate medium could either be data storage or a communication/computer network.
- If the compression and decompression processes induce no information loss, the compression scheme is *lossless*; otherwise, it is *lossy*.

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

B_0 – number of bits before compression

B_1 – number of bits after compression

- In general, we would desire any *codec* (encoder/decoder scheme) to have a *compression ratio* much larger than 1.0.
- The higher the *compression ratio*, the better the lossless compression scheme, as long as it is computationally feasible.

3.2 Entropy coding

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

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

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

- p_i – probability that symbol s_i will occur in S.

- $\log_2 \frac{1}{p_i}$ – indicates the amount of information contained in s_i, which corresponds to the number of bits needed to encode s_i.

- The definition of entropy is aimed at identifying often-occurring symbols in the datastream as good candidates for *short* codewords in the compressed bitstream.
- We use a *variable-length coding* scheme for entropy coding—frequently occurring symbols are given codes that are quickly transmitted, while infrequently occurring ones are given longer codes.
- For example, *E* occurs frequently in English, so we should give it a shorter code than *Q*, say.

- If we use $\bar{l}$ to denote the average length (measured in bits) of the codewords produced by the encoder, the Shannon Coding Theorem states that the entropy is the *best* we can do (under certain conditions):

$$\eta \leq \bar{l}$$

- Coding schemes aim to get as close as possible to this theoretical lower bound.

3.3 Huffman coding

- Huffman coding is an efficient method of compressing data without losing information.
- Huffman coding provides an efficient, unambiguous code by analyzing the frequencies that certain symbols appear in a message.
- Symbols that appear more often will be encoded as a shorter-bit string while symbols that aren't used as much will be encoded as longer strings.
- There are mainly two major parts in Huffman Coding
    1) Build a Huffman Tree from input characters.
    2) Traverse the Huffman Tree and assign codes to characters.

Algorithm

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.

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):
   a) The two least frequent symbols will have the same length for their Huffman codes, differing only at the last bit.
   b) Symbols that occur more frequently will have shorter Huffman codes than symbols that occur less frequently.
   c) The average code length for an information source S is strictly less than η + 1.

$$\bar{l} < \eta + 1$$

Example:

- Suppose the string below is to be sent over a network.

- Each character occupies 8 bits. There are a total of 15 characters in the above string. Thus, a total of *8*15 = 120* bits are required to send this string.
- Using the Huffman Coding technique, we can compress the string to a smaller size.
- Huffman coding first creates a tree using the frequencies of the character and then generates code for each character.
- Once the data is encoded, it has to be decoded. Decoding is done using the same tree.

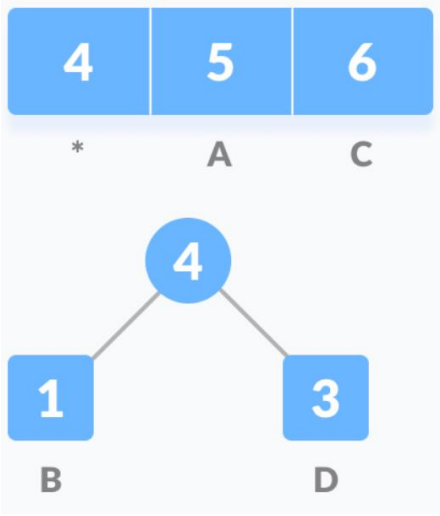
Huffman coding is done with the help of the following steps.

1. Calculate the frequency of each character in the string.

2. Sort the characters in increasing order of the frequency. These are stored in a priority queue *Q*.

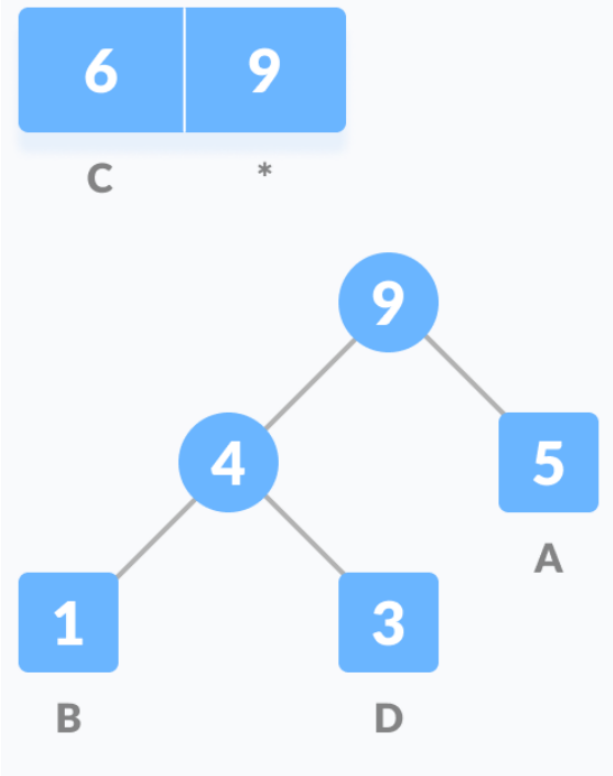
3. Make each unique character as a leaf node.

4. Create an empty node *z*. Assign the minimum frequency to the left child of *z* and assign the second minimum frequency to the right child of *z*. Set the value of the *z* as the sum of the above two minimum frequencies.

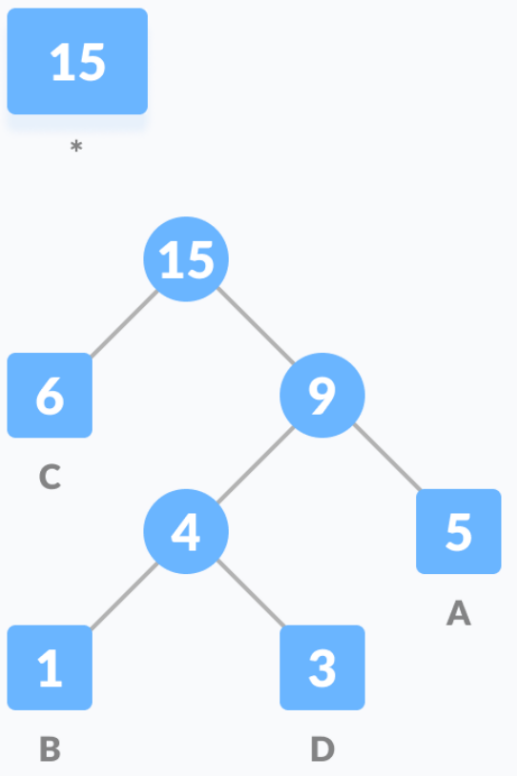


5. Remove these two minimum frequencies from *Q* and add the sum into the list of frequencies (* denote the internal nodes in the figure above).
6. Insert node *z* into the tree.

7. Repeat steps 3 to 5 for all the characters.

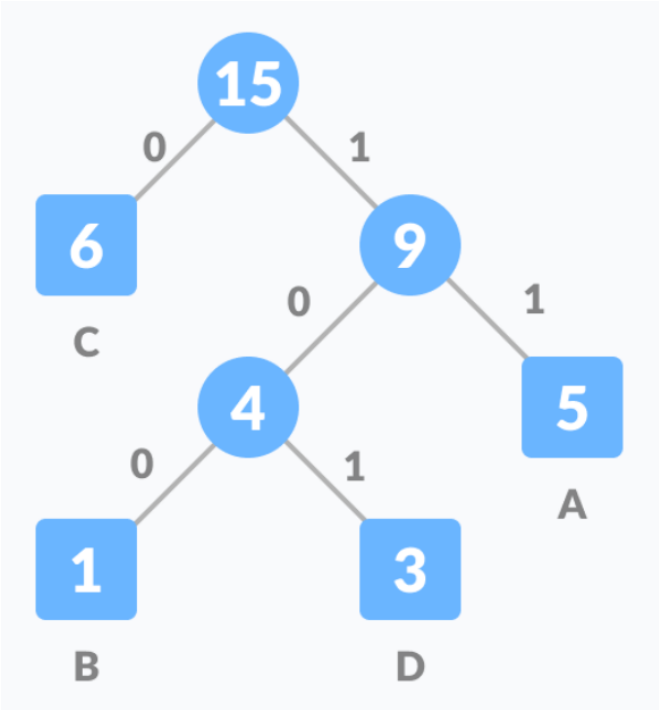


(a)



(b)

8. For each non-leaf node, assign 0 to the left edge and 1 to the right edge.



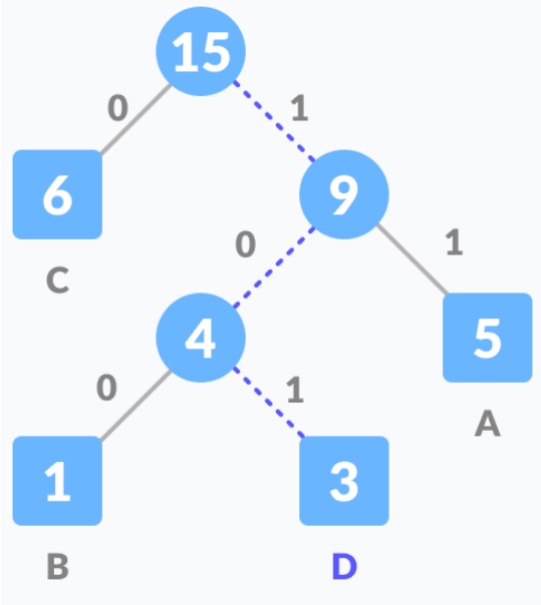
- For sending the above string over a network, we have to send the tree as well as the above compressed-code. The total size is given by the table below.

Character	Frequency	Code	Size
A	5	11	5*2 = 10
B	1	100	1*3 = 3
C	6	0	6*1 = 6
D	3	101	3*3 = 9
4 * 8 = 32 bits	15 bits		28 bits

- Without encoding, the total size of the string was 120 bits. After encoding the size is reduced to *32+15+28 = 75* bits.

Decoding the code

- For decoding the code, we can take the code and traverse through the tree to find the character.
- Let 101 is to be decoded, we can traverse from the root as in the figure below.



3.4 Adaptive 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.

- 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.

Adaptive Huffman coding

Procedures:

```
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);
   }
```

- `Initial_code` assigns symbols with some initially agreed-upon codes, without any prior knowledge of the frequency counts for them. For example, some conventional codes 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.

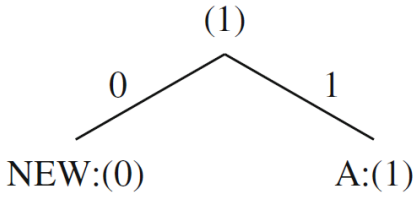
Example

Adaptive Huffman Coding for Symbol String AADCCDD

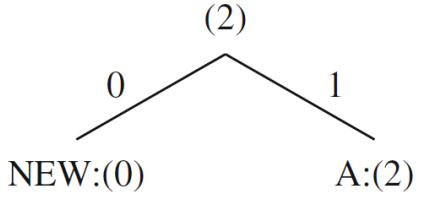
Symbol	Initial code
NEW	0
A	00001
B	00010
C	00011
D	00100
⋮	⋮

Initial code assignment for AADCCDD using adaptive Huffman coding

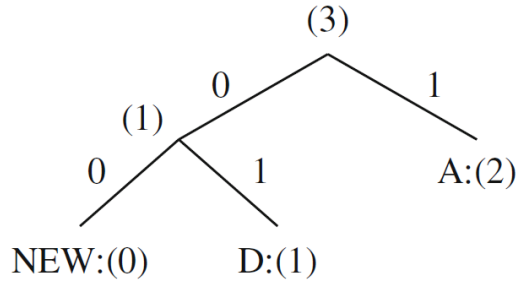
- Let us assume that the initial code assignment for both the encoder and decoder simply follows the ASCII order for the 26 symbols in an alphabet, A through Z, as the table above shows.
- To improve the implementation of the algorithm, we adopt an additional rule: if any character/symbol is to be sent 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)



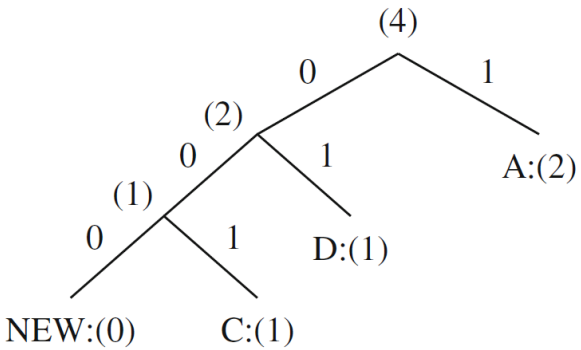
"A"



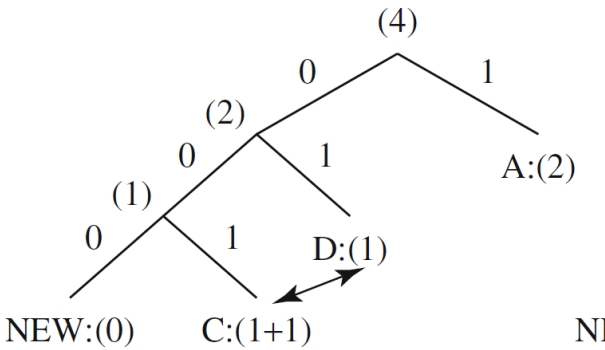
"AA"



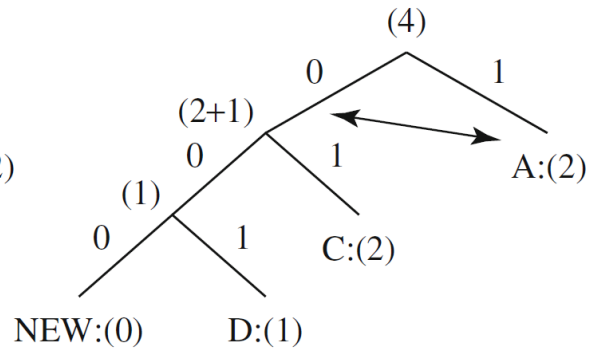
"AAD"



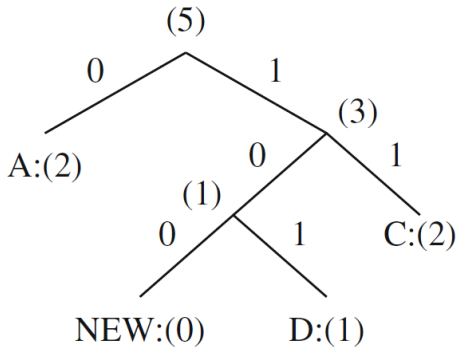
"AADC"



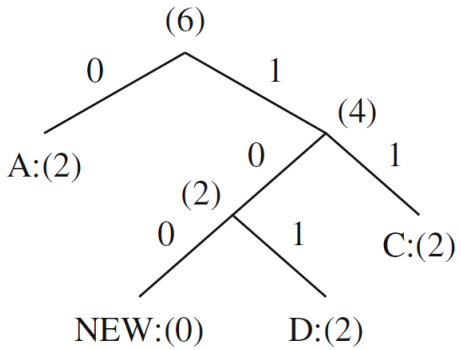
"AADCC" step 1



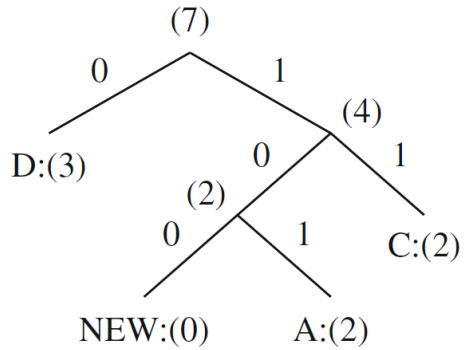
"AADCC" step 2



"AADCC" step 3



"AADCCD"



"AADCCDD"

- 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.

3.5 Dictionary-based coding (LZW)

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

```
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

LZW Compression for String ABABBABCABABBA

- Let us 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).
- 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.