

# Data Compression and Huffman Coding

- What is Data Compression?
- Why Data Compression?
- How is Data Compression possible?
- Lossless and Lossy Data Compression
- Static, Adaptive, and Hybrid Compression
- Compression Utilities and Formats
- Run-length Encoding
- Static Huffman Coding
- The Prefix property

# What is Data Compression?

Data compression is the representation of an information source (e.g. a data file, a speech signal, an image, or a video signal) as accurately as possible using the fewest number of bits.

Compressed data can only be understood if the decoding method is known by the receiver.

# Why Data Compression?

- Data storage and transmission cost money. This cost increases with the amount of data available.
  - This cost can be reduced by processing the data so that it takes less memory and less transmission time.
- Some data types consist of many chunks of repeated data (e.g. multimedia data such as audio, video, images, ...).
  - Such “raw” data can be transformed into a compressed data representation form saving a lot of storage and transmission costs.
- Disadvantage of Data compression:  
Compressed data must be decompressed to be viewed (or heard), thus extra processing is required.

# Lossless and Lossy Compression Techniques

- Data compression techniques are broadly classified into lossless and lossy.
- Lossless techniques enable exact reconstruction of the original document from the compressed information.
  - Exploit redundancy in data
  - Applied to general data
  - Examples: Run-length, Huffman, LZ77, LZ78, and LZW
- Lossy compression - reduces a file by permanently eliminating certain redundant information
  - Exploit redundancy and human perception
  - Applied to audio, image, and video
  - Examples: JPEG and MPEG
- Lossy techniques usually achieve higher compression rates than lossless ones but the latter are more accurate.

# Classification of Lossless Compression Techniques

- Lossless techniques are classified into static, adaptive (or dynamic), and hybrid.
- In a *static* method the mapping from the set of messages to the set of codewords is fixed before transmission begins, so that a given message is represented by the same codeword every time it appears in the message being encoded.
  - Static coding requires two passes: one pass to compute probabilities (or frequencies) and determine the mapping, and a second pass to encode.
  - **Examples: Static Huffman Coding**
- In an *adaptive method* the mapping from the set of messages to the set of codewords changes over time.
  - All of the adaptive methods are *one-pass* methods; only one scan of the message is required.
  - **Examples: LZ77, LZ78, LZW, and Adaptive Huffman Coding**
- An algorithm may also be a *hybrid*, neither completely static nor completely dynamic.



# Static Huffman Coding

- Static Huffman coding assigns **variable length codes** to symbols based on their frequency of occurrences in the given message. **Low frequency symbols are encoded using many bits, and high frequency symbols are encoded using fewer bits.**
- The message to be transmitted is first analyzed to find the relative frequencies of its constituent characters.
- The coding process generates a binary tree, the Huffman code tree, with branches labeled with bits (0 and 1).
- The Huffman tree (or the character codeword pairs) must be sent with the compressed information to enable the receiver decode the message.

# Static Huffman Coding Algorithm

Find the frequency of each character in the file to be compressed;

For each distinct character create a one-node binary tree containing the character and its frequency as its priority;

Insert the one-node binary trees in a priority queue in increasing order of frequency;

```
while (there are more than one tree in the priority queue) {  
    dequeue two trees t1 and t2;  
    Create a tree t that contains t1 as its left subtree and t2 as its right subtree; // 1  
    priority (t) = priority(t1) + priority(t2);  
    insert t in its proper location in the priority queue; // 2  
}
```

Assign 0 and 1 weights to the edges of the resulting tree, such that the left and right edge of each node do not have the same weight; // 3

**Note:** The Huffman code tree for a particular set of characters is not unique.  
(Steps 1, 2, and 3 may be done differently).

# Static Huffman Coding example

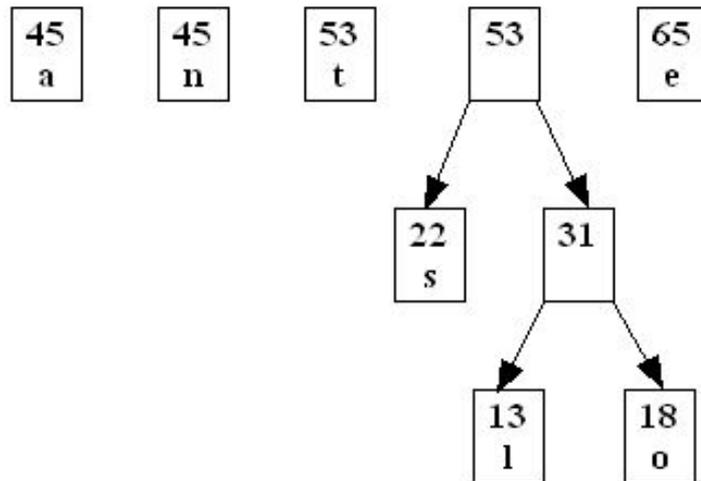
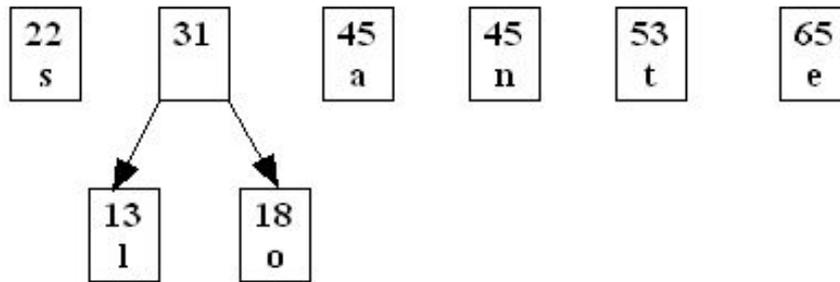
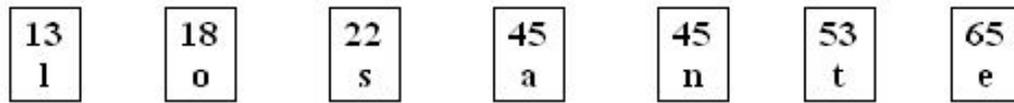
**Example:** Information to be transmitted over the internet contains the following characters with their associated frequencies:

Character	a	e	l	n	o	s	t
Frequency	45	65	13	45	18	22	53

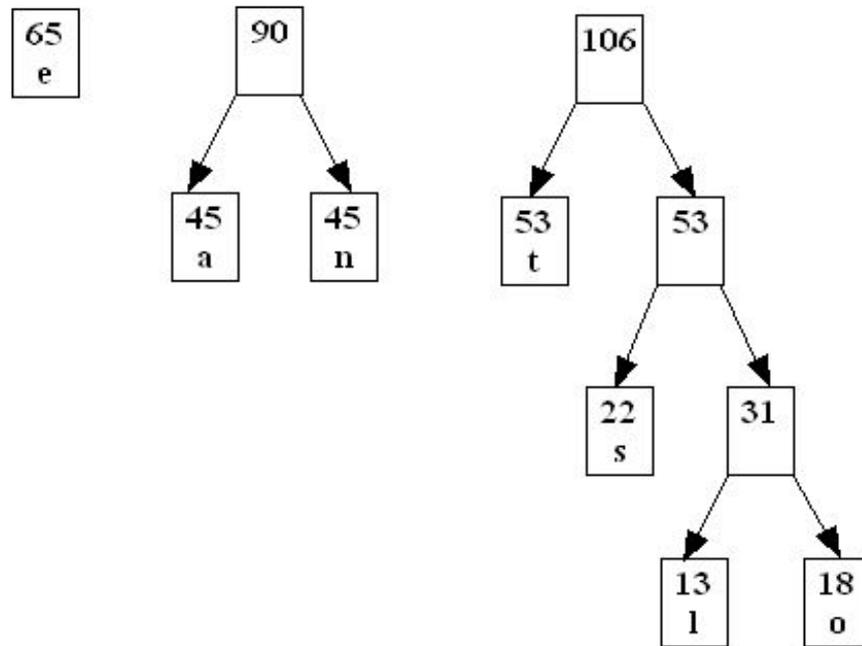
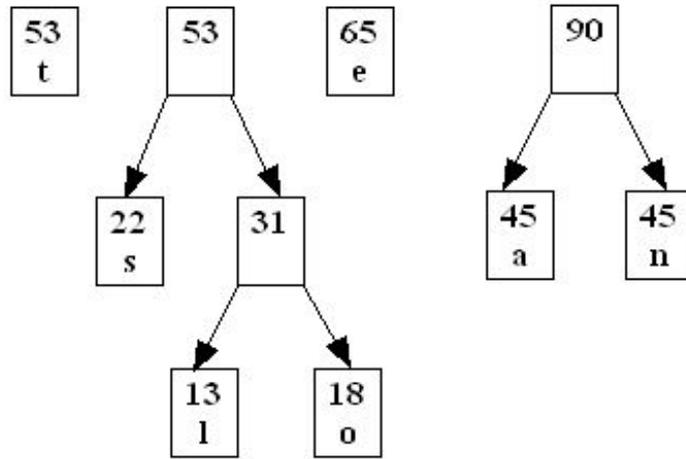
Use Huffman technique to answer the following questions:

- Build the Huffman code tree for the message.
- Use the Huffman tree to find the codeword for each character.
- If the data consists of only these characters, what is the total number of bits to be transmitted? What is the compression ratio?
- Verify that your computed Huffman codewords satisfy the Prefix property.

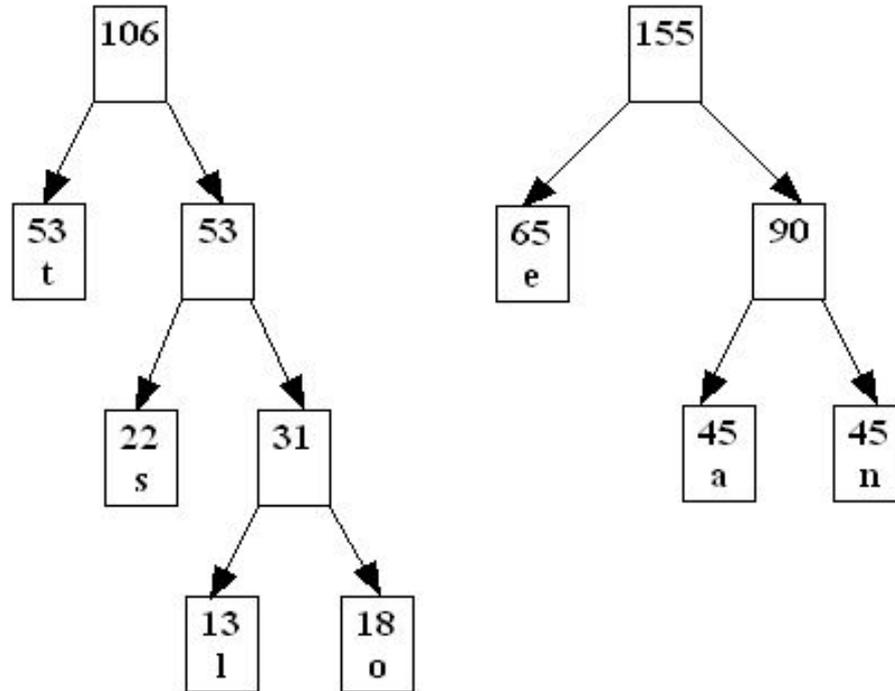
# Static Huffman Coding example (cont'd)



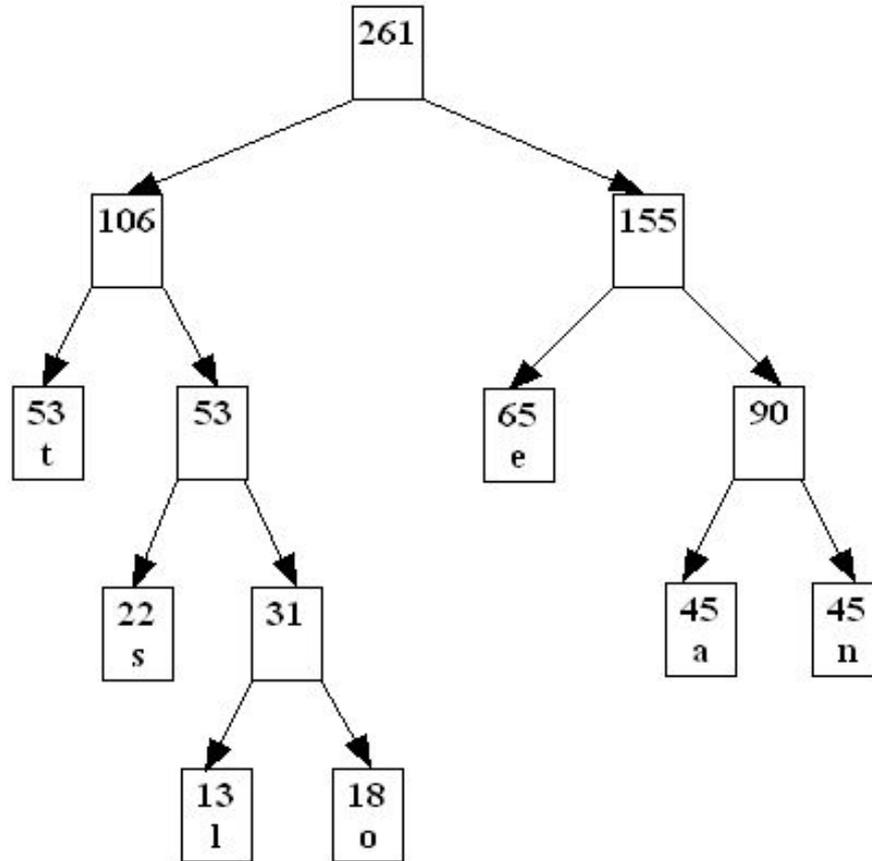
# Static Huffman Coding example (cont'd)



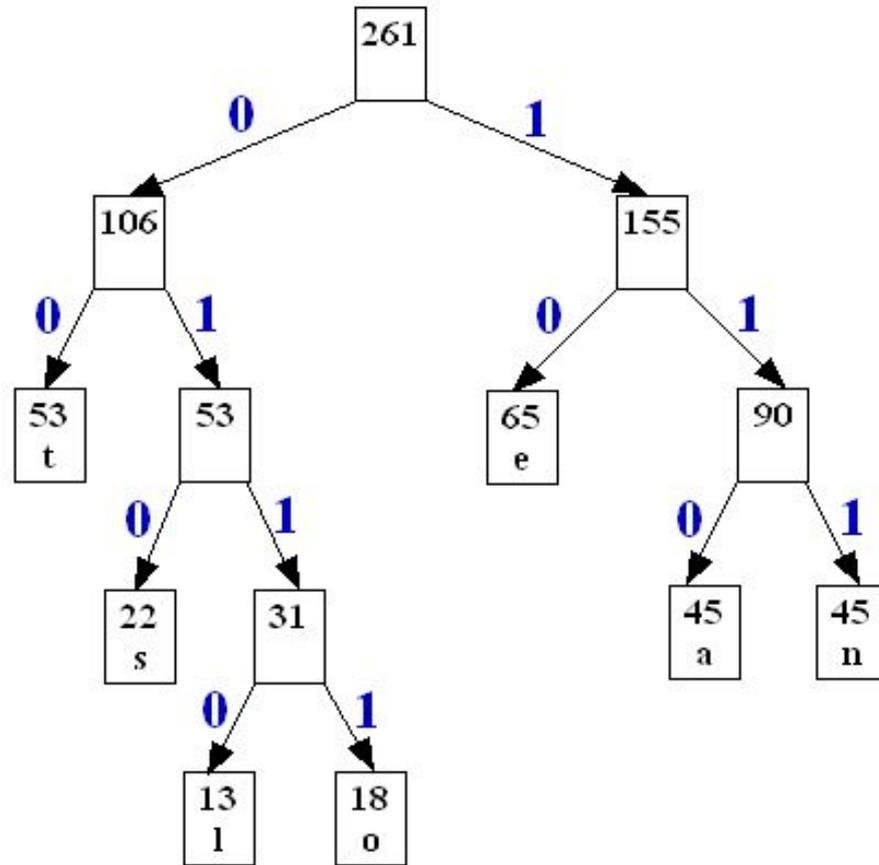
# Static Huffman Coding example (cont'd)



# Static Huffman Coding example (cont'd)



# Static Huffman Coding example (cont'd)



The sequence of zeros and ones that are the arcs in the path from the root to each leaf node are the desired codes:

character	a	e	l	n	o	s	t
Huffman codeword	110	10	0110	111	0111	010	00

# Static Huffman Coding example (cont'd)

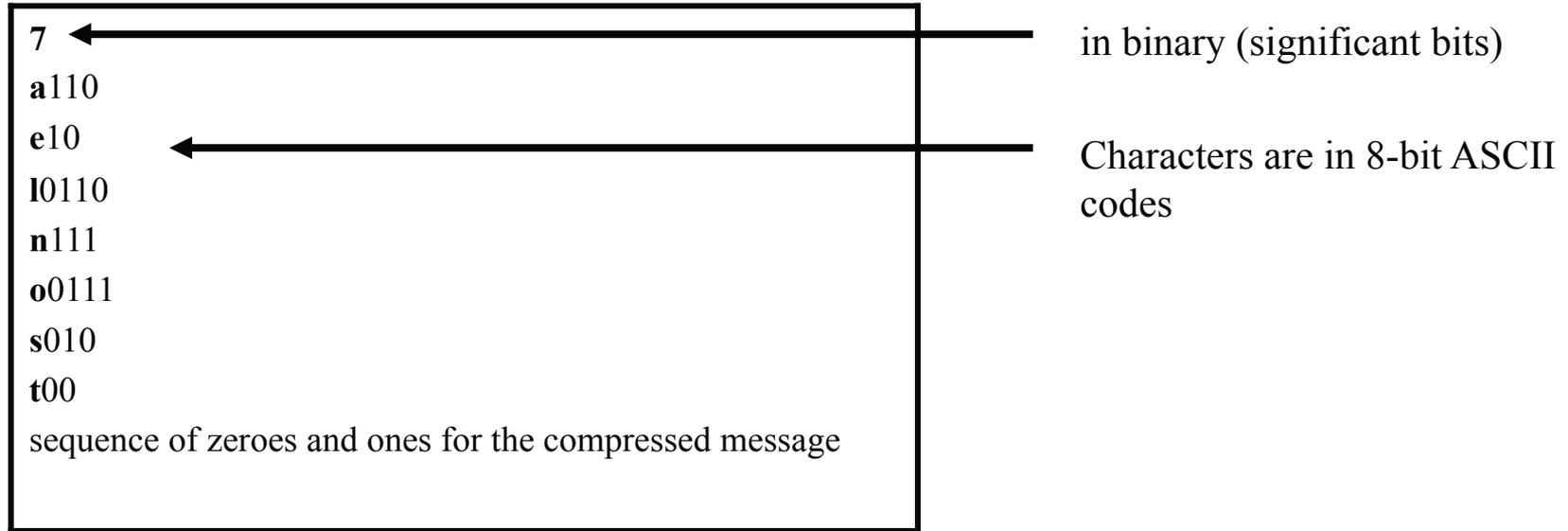
If we assume the message consists of only the characters a,e,l,n,o,s,t then the number of bits for the compressed message will be 696:

<b>character</b>	<b>a</b>	<b>e</b>	<b>l</b>	<b>n</b>	<b>o</b>	<b>s</b>	<b>t</b>	
<b>codeword</b>	<b>110</b>	<b>10</b>	<b>0110</b>	<b>111</b>	<b>0111</b>	<b>010</b>	<b>00</b>	
<b>codeword bits</b>	<b>3</b>	<b>2</b>	<b>4</b>	<b>3</b>	<b>4</b>	<b>3</b>	<b>2</b>	
<b>character frequency</b>	<b>45</b>	<b>65</b>	<b>13</b>	<b>45</b>	<b>18</b>	<b>22</b>	<b>53</b>	
<b>codeword bits * frequency</b>	<b>135</b>	<b>130</b>	<b>52</b>	<b>135</b>	<b>72</b>	<b>66</b>	<b>106</b>	<b>sum 696</b>

If the message is sent uncompressed with 8-bit ASCII representation for the characters, we have  $261 * 8 = 2088$  bits.

# Static Huffman Coding example (cont'd)

Assuming that the number of character-codeword pairs and the pairs are included at the beginning of the binary file containing the compressed message in the following format:



$$\begin{aligned}\text{Number of bits for the transmitted file} &= \text{bits}(7) + \text{bits}(\text{characters}) + \text{bits}(\text{codewords}) + \text{bits}(\text{compressed message}) \\ &= 3 + (7*8) + 21 + 696 = 776\end{aligned}$$

$$\begin{aligned}\text{Compression ratio} &= \text{bits for ASCII representation} / \text{number of bits transmitted} \\ &= 2088 / 776 = 2.69\end{aligned}$$

Thus, the size of the transmitted file is  $100 / 2.69 = 37\%$  of the original ASCII file

# The Prefix Property

- Data encoded using Huffman coding is uniquely decodable. This is because Huffman codes satisfy an important property called the prefix property:

In a given set of Huffman codewords, no codeword is a prefix of another Huffman codeword

- For example, in a given set of Huffman codewords, **10** and **101** cannot simultaneously be valid Huffman codewords because the first is a prefix of the second.
- We can see by inspection that the codewords we generated in the previous example are valid Huffman codewords.

# The Prefix Property (cont'd)

To see why the prefix property is essential, consider the codewords given below in which “e” is encoded with **110** which is a prefix of “f”

character	a	b	c	d	e	f
codeword	0	101	100	111	110	1100

The decoding of 11000100110 is ambiguous:

**11000100110**  $\Rightarrow$  **face**

**11000100110**  $\Rightarrow$  **eaace**

# Encoding and decoding examples

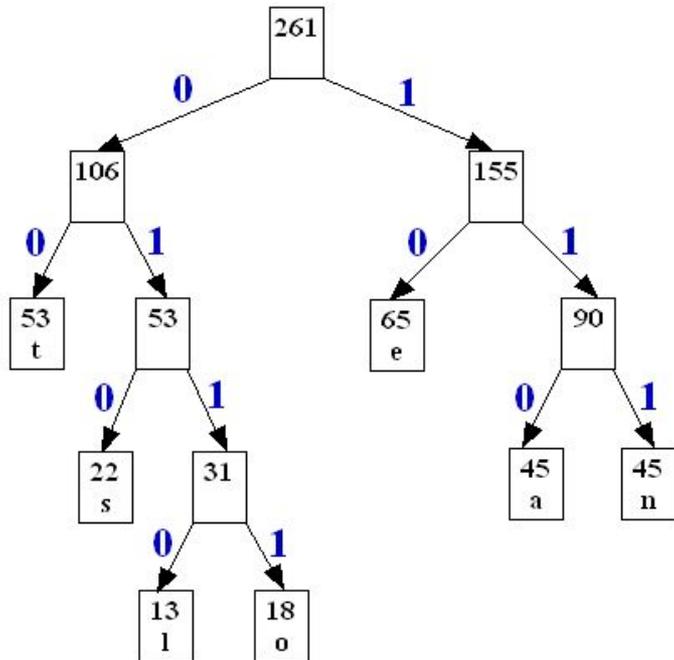
- Encode (compress) the message **tenseas** using the following codewords:

character	a	e	l	n	o	s	t
Huffman codeword	110	10	0110	111	0111	010	00

**Answer:** Replace each character with its codeword:

**001011101010110010**

- Decode (decompress) each of the following encoded messages, if possible, using the Huffman codeword tree given below **0110011101000** and **11101110101011**:



**Answer:** Decode a bit-stream by starting at the root and proceeding down the tree according to the bits in the message (0 = left, 1 = right). When a leaf is encountered, output the character at that leaf and restart at the root. If a leaf cannot be reached, the bit-stream cannot be decoded.

(a) **0110011101000** => **lost**

(b) **11101110101011**

The decoding fails because the corresponding node for **11** is not a leaf

# Exercises

1. Using the Huffman tree constructed in this session, decode the following sequence of bits, if possible. Otherwise, where does the decoding fail?

10100010111010001000010011

2. Using the Huffman tree constructed in this session, write the bit sequences that encode the messages:

test , state , telnet , notes

3. Mention one disadvantage of a lossless compression scheme and one disadvantage of a lossy compression scheme.
5. Write a Java program that implements the Huffman coding algorithm.