Introduction to Data Compression

- 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.

- Disadvantage of Data compression:

  Compressed data must be decompressed to be viewed (or heard), thus extra processing is required.

- The design of data compression schemes therefore involve trade-offs between various factors, including the degree of compression, the amount of distortion introduced (if using a lossy compression scheme), and the computational resources required to compress and uncompress the data.

How is data compression possible?

➢ Compression is possible because information usually contains redundancies, or information that is often repeated.

    ➢ Examples include reoccurring letters, numbers or pixels

➢ File compression programs remove this redundancy.

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.

Compression Utilities and Formats

- Compression tool examples:

  winzip, pkzip, compress, gzip

- General compression formats:

  .zip, .gz

- Common image compression formats:

  JPEG, JPEG 2000, BMP, GIF, PCX, PNG, TGA, TIFF, WMP

- Common audio (sound) compression formats:

  MPEG-1 Layer III (known as MP3), RealAudio (RA, RAM, RP), AU, Vorbis, WMA, AIFF, WAVE, G.729a

- Common video (sound and image) compression formats:

  MPEG-1, MPEG-2, MPEG-4, DivX, Quicktime (MOV), RealVideo (RM), Windows Media Video (WMV), Video for Windows (AVI), Flash video (FLV)

Run-length encoding

The following string:

BBBBHHDDXXXXKKKKWWZZZZ

can be encoded more compactly by replacing each repeated string of characters by a single instance of the repeated character and a number that represents the number of times it is repeated:

4B2H2D4X4K2W4Z

Here "4B" means four B's, and 2H means two H's, and so on. Compressing a string in this way is called *run-length encoding*.

As another example, consider the storage of a rectangular image. As a single color bitmapped image, it can be stored as:

```
0000000000000000000000000000000000000000
000000000000000000000000000000000000000
00000000001111111111111111111110000000000
00000000001000000000000000000001000000000
00000000001000000000000000000001000000000
00000000001000000000000000000001000000000
00000000001111111111111111111110000000000
000000000000000000000000000000000000000
```

The rectangular image can be compressed with run-length encoding by counting identical bits as follows:

0, 40
0, 40
0,10 1,20 0,10
0,10 1,1 0,18 1,1 0,10
0,10 1,1 0,18 1,1 0,10
0,10 1,1 0,18 1,1 0,10
0,10 1,20 0,10
0,40

The first line says that the first line of the bitmap consists of 40 0's. The third line says that the third line of the bitmap consists of 10 0's followed by 20 1's followed by 10 more 0's, and so on for the other lines

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).
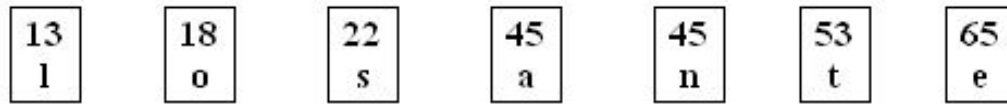
Static Huffman Coding example

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

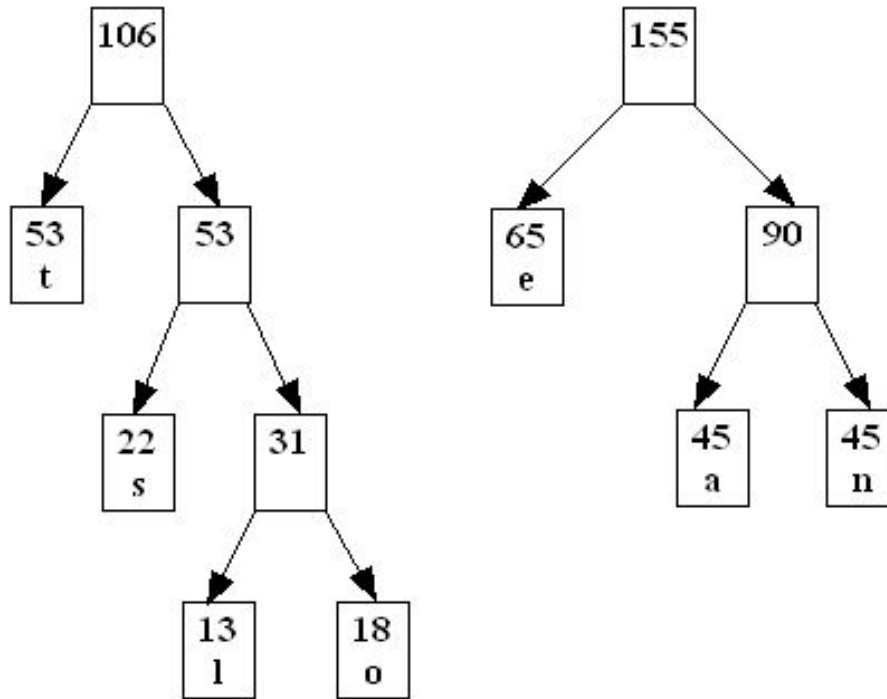| Character	a	e	l	n	o	s	t
Frequency	45	65	13	45	18	22	53

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?

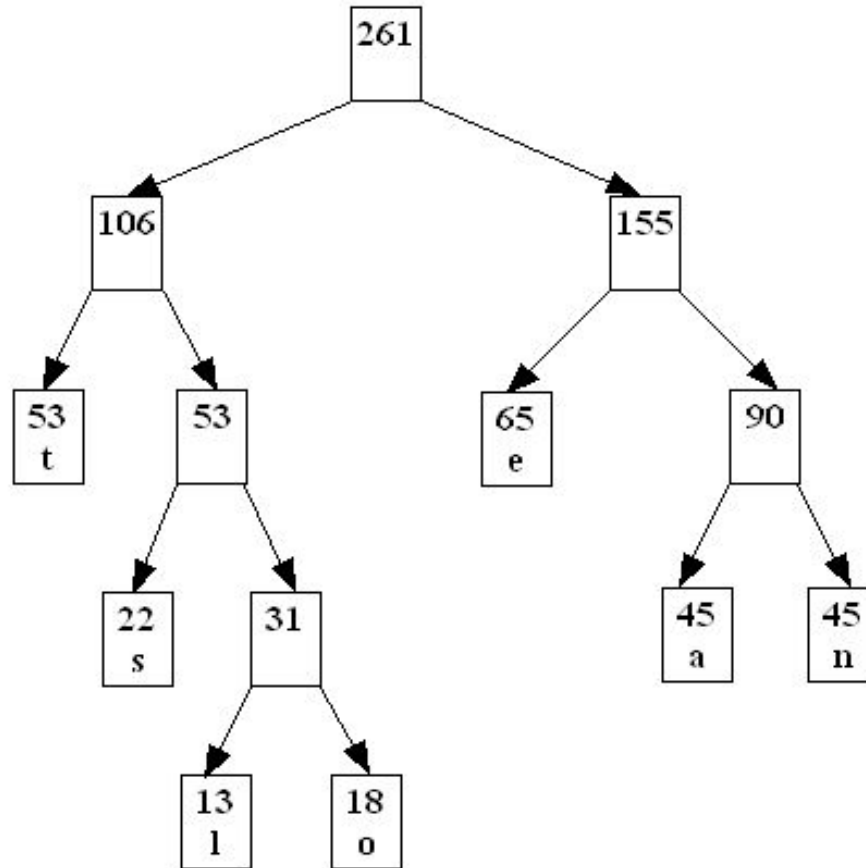➢ Verify that your computed Huffman codewords satisfy the Prefix property.
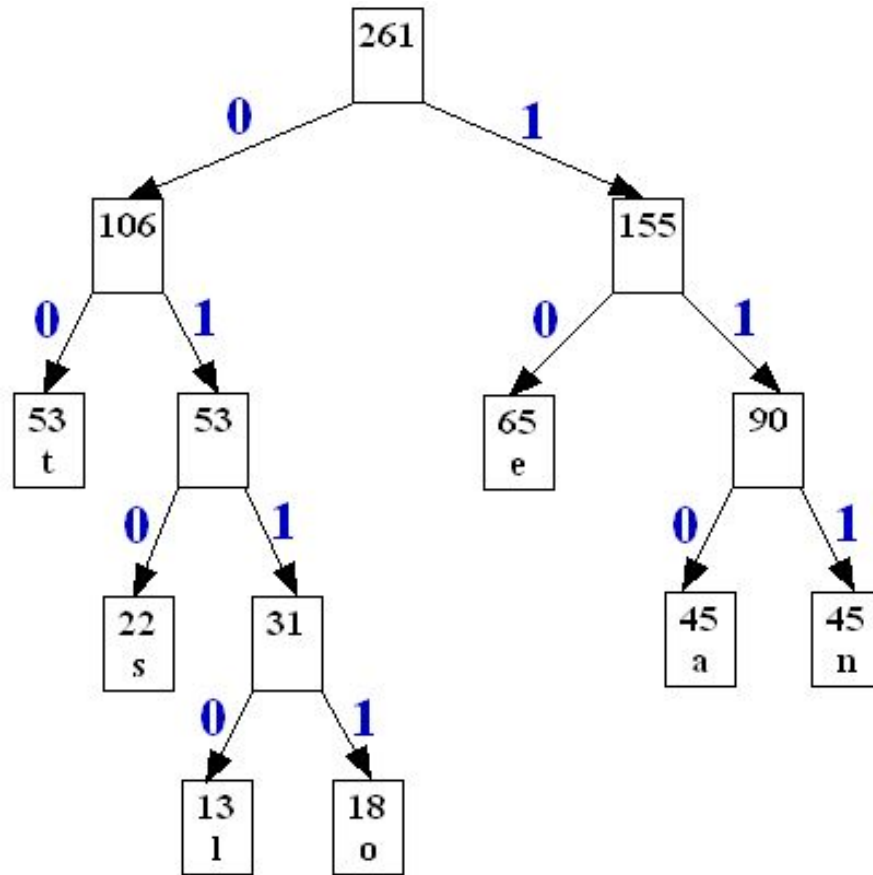
Static Huffman Coding example (cont'd)
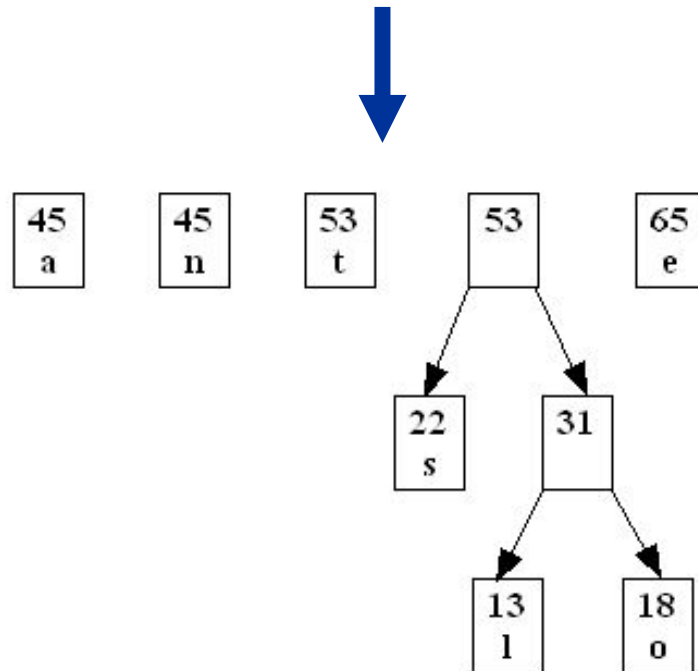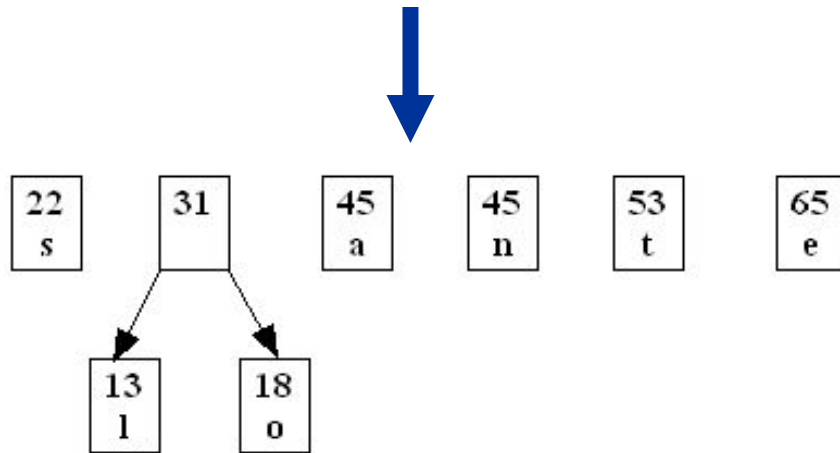
Static Huffman Coding example (cont'd)

Static Huffman Coding example (cont'd)

Static Huffman Coding example (cont'd)

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

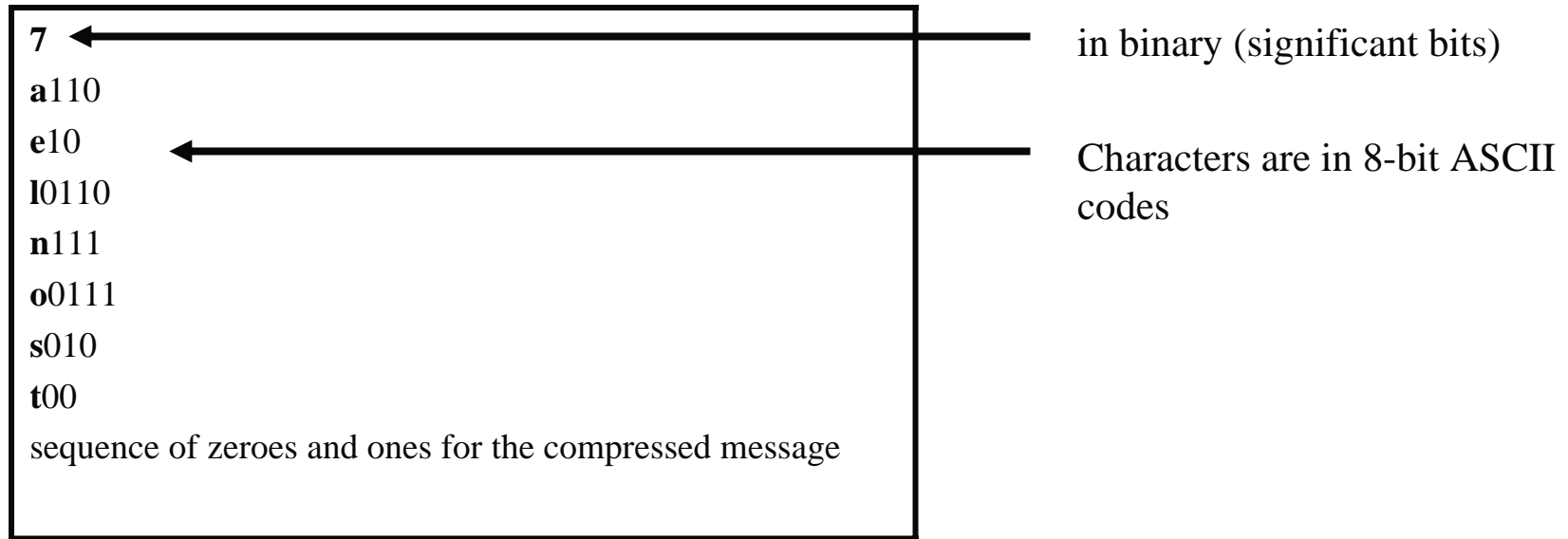# Static Huffman Coding example (cont'd)

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:

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

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:

7 ← in binary (significant bits)

a110

e10 ← Characters are in 8-bit ASCII codes

l0110

n111

o0111

s010

t00

sequence of zeroes and ones for the compressed message

Number of bits for the transmitted file = bits(7) + bits(characters) + bits(codewords) + bits(compressed message)

$$= 3 + (7*8) + 21 + 696 = 776$$

Compression ratio = bits for ASCII representation / number of bits transmitted

$$= 2088 / 776 = 2.69$$

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 <u>prefix</u> 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 => **face**

**11000100110** => **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**:
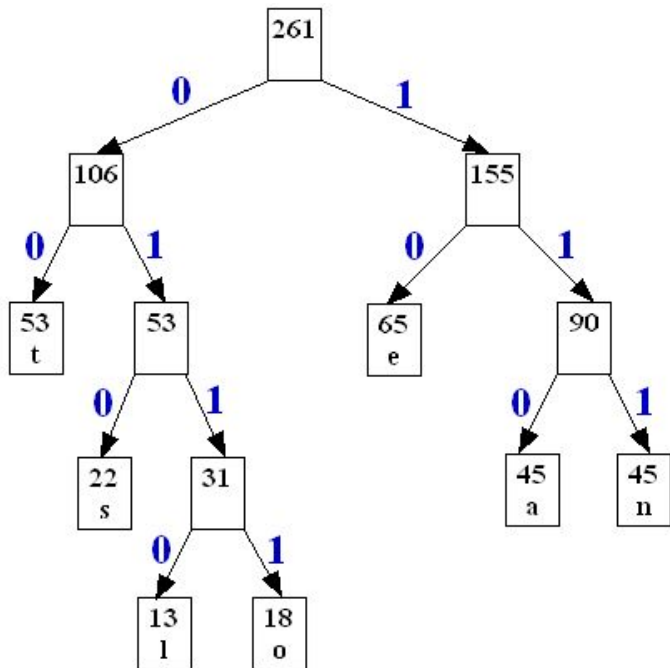
Answer: Decode a bit-stream by starting at the root and proceeding down the tree according to the bits in the message (0 = left, 1 = right). When a leaf is encountered, output the character at that leaf and restart at the root .If a leaf cannot be reached, the bit-stream cannot be decoded.

(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?

   101000101110100010000010011

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.

4. Write a Java program that implements the Huffman coding algorithm.