King Fahd University of Petroleum & Minerals
College of Computer Science & Engineering

Information & Computer Science Department

Unit 13

Data Compression

Information and Computer Science ICS

Reading Assignment

- "Data Structures and Algorithms in Java", 3rd Edition, Adam Drozdek, Cengage Learning, ISBN 978-9814239233
  - Chapter 11
    - Section 11.2: Huffman Coding (11.2.1 is not included)
    - Section 11.3: Run-Length Encoding
    - Section 11.4: Ziv-Lempel Code (LZW is not included)

Data Compression and Huffman Coding

- Introduction
  - What is Data Compression?
  - Why Data Compression?
- Lossless and Lossy Data Compression
  - Static, Adaptive, and Hybrid Compression
- Compression Utilities and Formats
- Compression Techniques
  - Run-length Encoding
  - Static Huffman Coding
  - Lempel-Ziv Encoding

Introduction: 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.

Introduction: 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.

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)

Compression Techniques

- Run-length Encoding
- Static Huffman Coding
- Lempel-Ziv Encoding
  - LZ78 Encoding and Decoding

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:

- B4H2D2X4K4W2Z4

Here "B4" means four B's, and “H2” means two H's, etc. 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:

```
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
00000000001111111111111111111100000000000
00000000001000000000000000000100000000000
00000000001000000000000000000100000000000
00000000001000000000000000000100000000000
00000000001111111111111111111100000000000
00000000000000000000000000000000000000000
```

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

B_0 = # bits required before compression

B_1 = # bits required after compression

Compression Ratio = B_0 / B_1.

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 exists 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

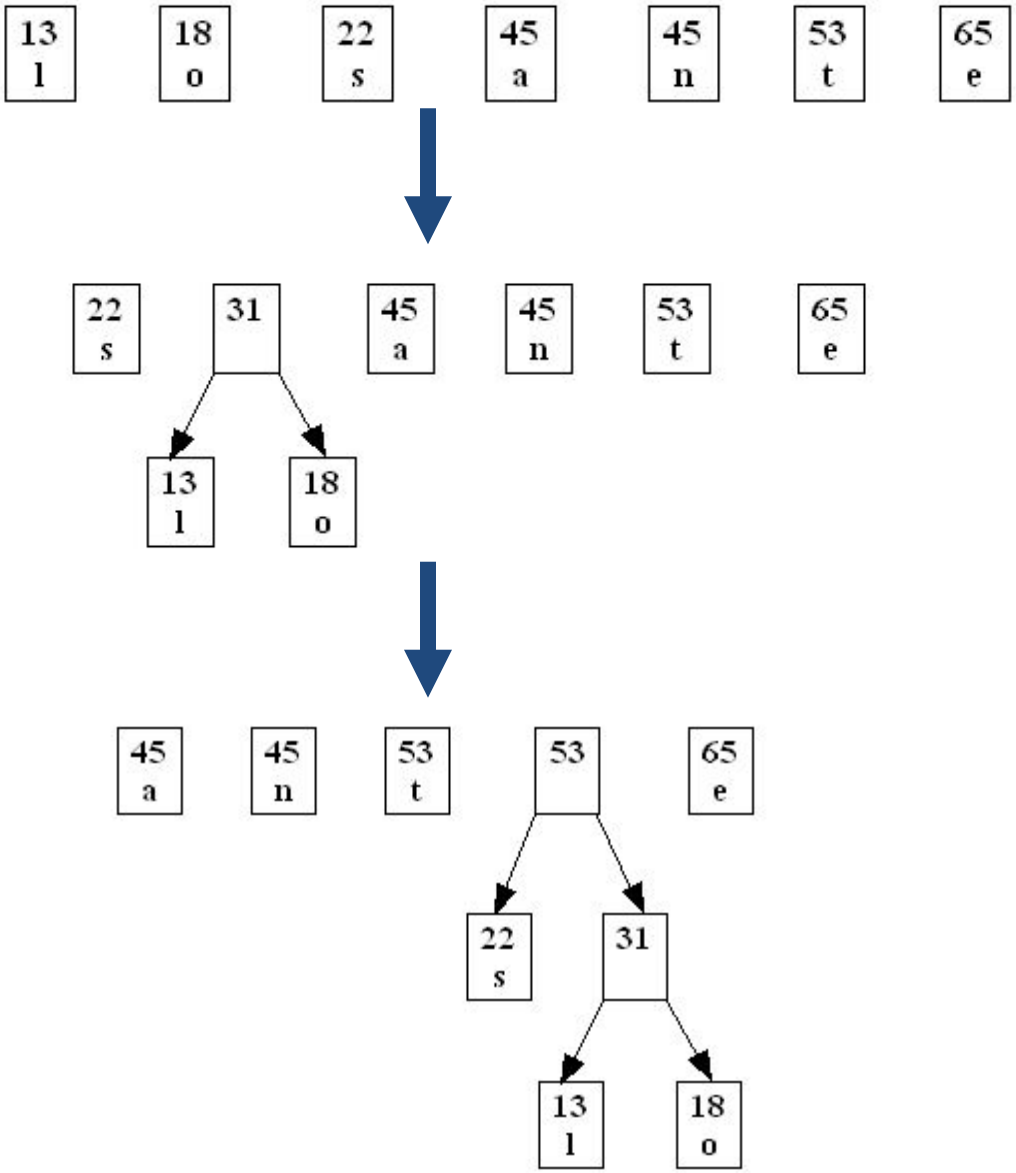
Example: Information to be transmitted over the internet contains the following characters with their associated frequencies:

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

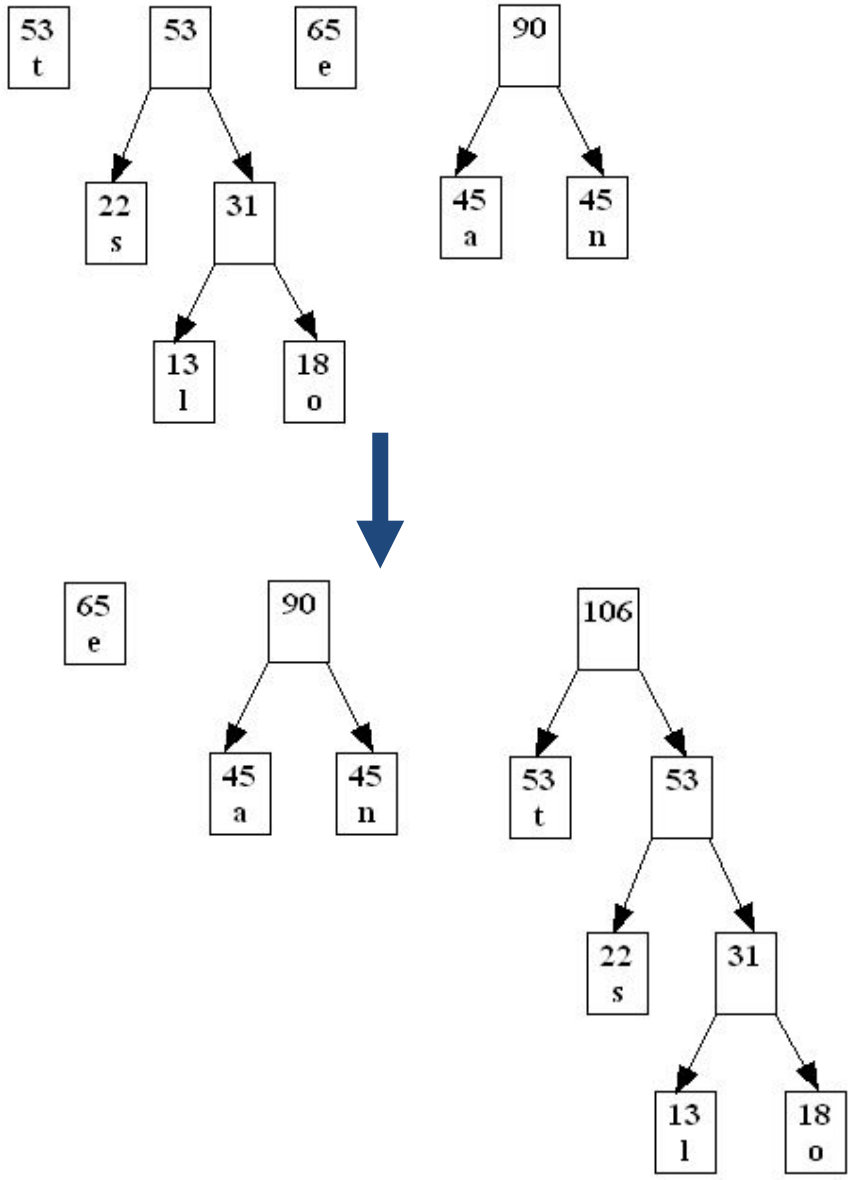
Use Huffman technique to answer the following questions:

- 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.

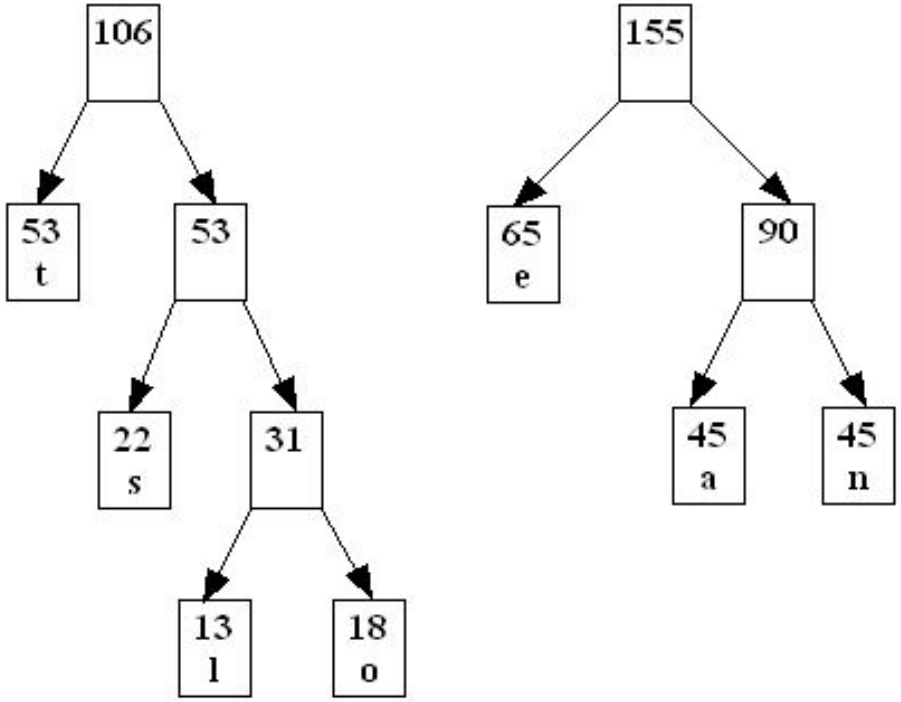
Static Huffman Coding example (cont'd)



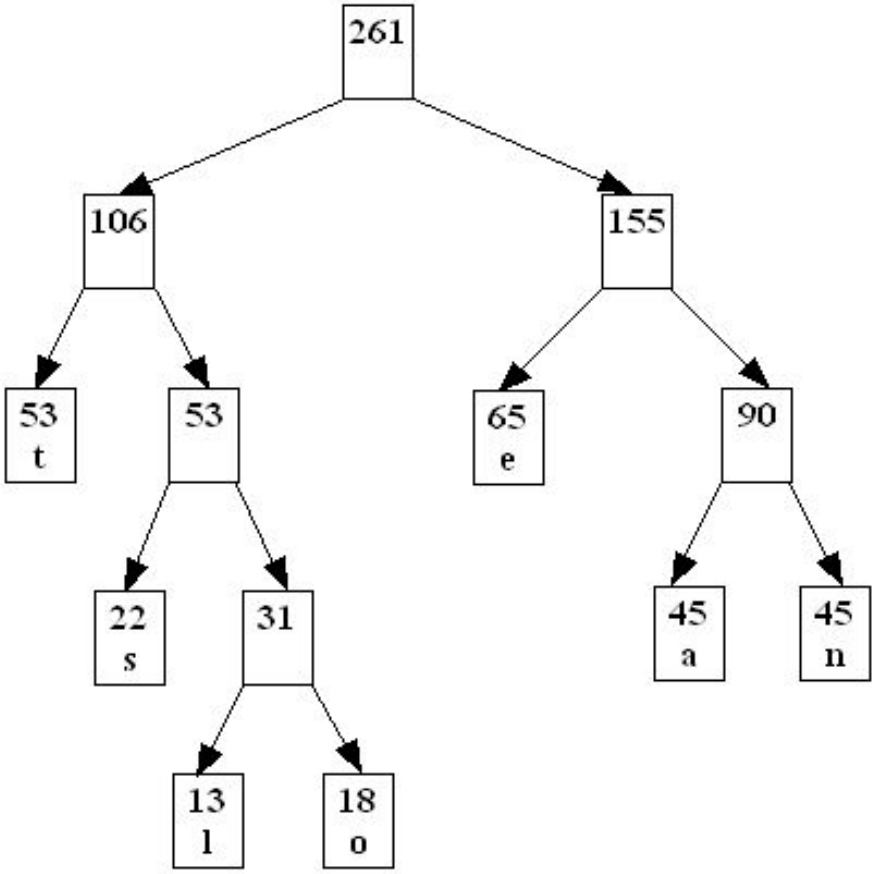
Static Huffman Coding example (cont'd)



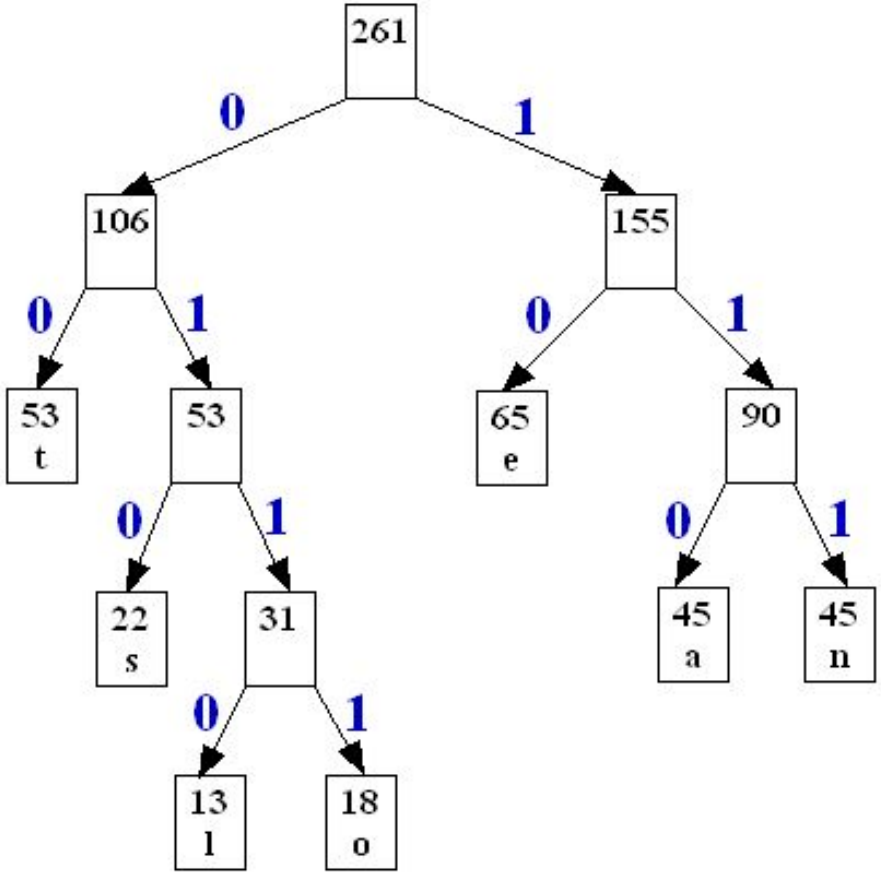
Static Huffman Coding example (cont'd)



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)

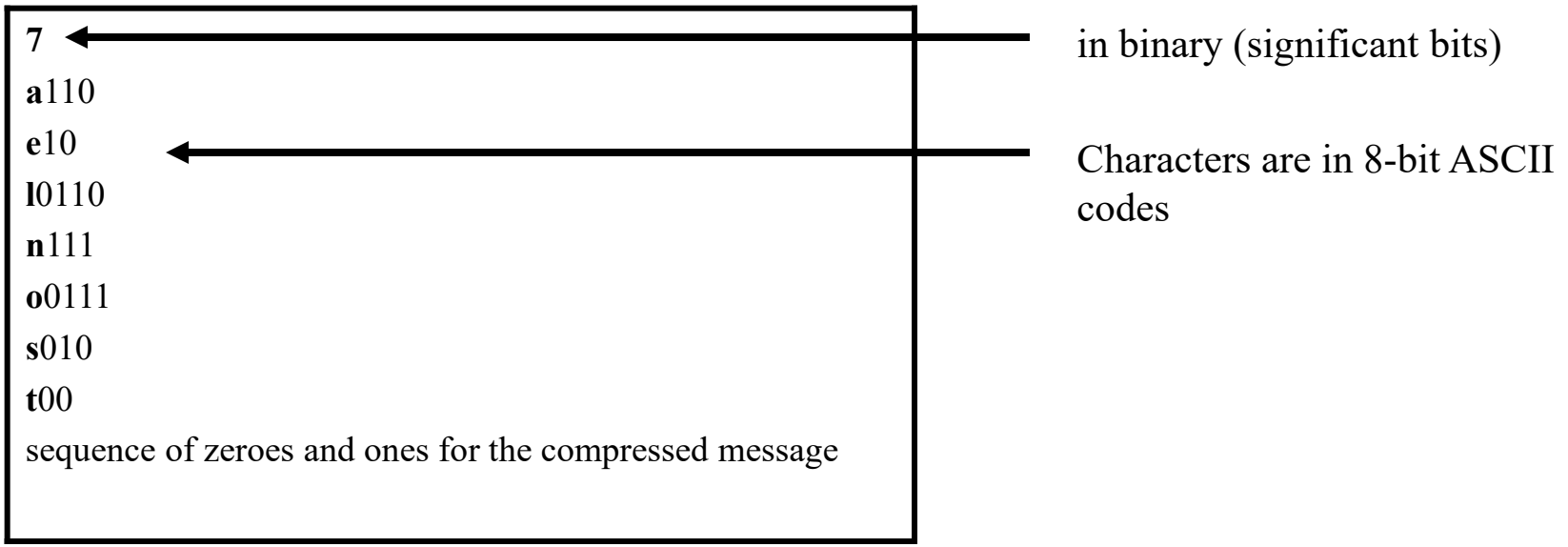
If we assume the message consists of only the characters a,e,l,n,o,s,t then the number of bits for the compressed message will be 696:

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:



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 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 =>

11000100110 =>

Encoding and decoding examples

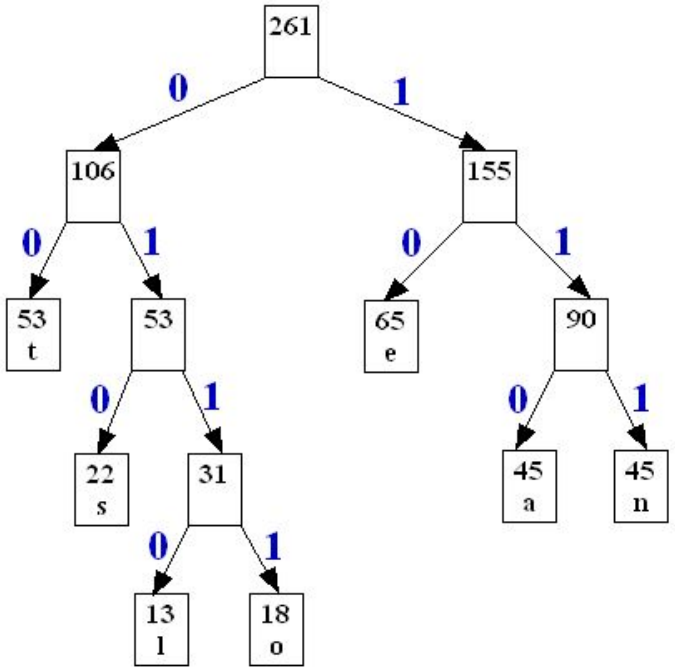
- Encode (compress) the message **tenseas** using the following codewords:

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

Answer: Replace each character with its codeword:

00101110101011010

- 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 =>

(b) 11101110101011

Lempel-Ziv Encoding

- Data compression up until the late 1970's mainly directed towards creating better methodologies for Huffman coding.
- An innovative, radically different method was introduced in1977 by Abraham Lempel and Jacob Ziv.
- This technique (called Lempel-Ziv) actually consists of two considerably different algorithms, LZ77 and LZ78.
- Due to patents, LZ77 and LZ78 led to many variants:

LZ77 Variants	LZR	LZSS	LZB	LZH		
LZ78 Variants	LZW	LZC	LZT	LZMW	LZJ	LZFG

- The **zip** and **unzip** use the LZH technique while UNIX's **compress** methods belong to the LZW and LZC classes.

LZ78 Compression Algorithm

LZ78 inserts one- or multi-character, non-overlapping, distinct patterns of the message to be encoded in a Dictionary.

The multi-character patterns are of the form: $C_0C_1 \ldots C_{n-1}C_n$. The prefix of a pattern consists of all the pattern characters except the last: $C_0C_1 \ldots C_{n-1}$

LZ78 Output:

(0, char)	if one-character pattern is not in Dictionary.
(DictionaryPrefixIndex, lastPatternCharacter)	if multi-character pattern is not in Dictionary.
(DictionaryPrefixIndex,)	if the last input character or the last pattern is in the Dictionary.

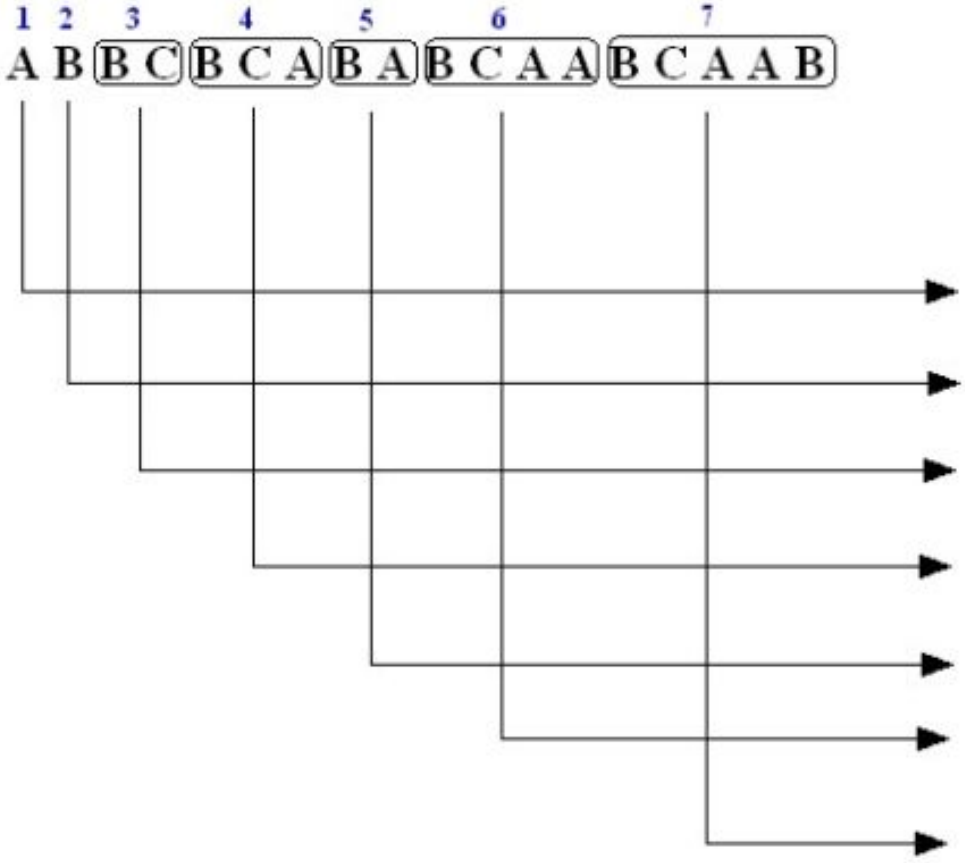
Note: The dictionary is usually implemented as a hash table.

LZ78 Compression Algorithm (cont'd)

```
Dictionary ← empty ; Prefix ← empty ; DictionaryIndex ← 1;
while(characterStream is not empty)
{
      Char ← next character in characterStream;
     if(Prefix + Char exists in the Dictionary)
          Prefix ← Prefix + Char ;
      else
       {
            if(Prefix is empty)
                 CodeWordForPrefix ← 0 ;
           else
                 CodeWordForPrefix ← DictionaryIndex for Prefix ;
            Output: (CodeWordForPrefix, Char) ;
             insertInDictionary( ( DictionaryIndex , Prefix + Char) );
             DictionaryIndex++ ;
             Prefix ← empty ;
        }
}
if(Prefix is not empty)
{
      CodeWordForPrefix ← DictionaryIndex for Prefix;
    Output: (CodeWordForPrefix ,   ) ;
}
```

Example 1: LZ78 Compression

Encode (i.e., compress) the string **ABBCBCABABCAABCAAB** using the LZ78 algorithm.



	Dictionary	
output	index	string
(0, A)	1	A
(0, B)	2	B
(2, C)	3	BC
(3, A)	4	BCA
(2, A)	5	BA
(4, A)	6	BCAA
(6, B)	7	BCAAB

The compressed message is: **(0,A)(0,B)(2,C)(3,A)(2,A)(4,A)(6,B)**

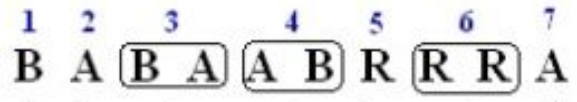
Note: The above is just a representation, the commas and parentheses are not transmitted; we will discuss the actual form of the compressed message later on in slide 32.

Example 1: LZ78 Compression (cont'd)

1. **A** is not in the Dictionary; insert it
2. **B** is not in the Dictionary; insert it
3. **B** is in the Dictionary.
   BC is not in the Dictionary; insert it.
4. **B** is in the Dictionary.
   BC is in the Dictionary.
   BCA is not in the Dictionary; insert it.
5. **B** is in the Dictionary.
   BA is not in the Dictionary; insert it.
6. **B** is in the Dictionary.
   BC is in the Dictionary.
   BCA is in the Dictionary.
   BCAA is not in the Dictionary; insert it.
7. **B** is in the Dictionary.
   BC is in the Dictionary.
   BCA is in the Dictionary.
   BCAA is in the Dictionary.
   BCAAB is not in the Dictionary; insert it.

Example 2: LZ78 Compression

Encode (i.e., compress) the string **BABAABRRRA** using the LZ78 algorithm.



	Dictionary	
output	index	string
(0, B)	1	B
(0, A)	2	A
(1, A)	3	BA
(2, B)	4	AB
(0, R)	5	R
(5, R)	6	RR
(2,)		

The compressed message is:

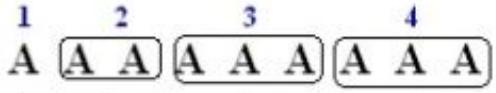
(0,B)(0,A)(1,A)(2,B)(0,R)(5,R)(2,)

Example 2: LZ78 Compression (cont'd)

1. **B** is not in the Dictionary; insert it
2. **A** is not in the Dictionary; insert it
3. **B** is in the Dictionary.
   BA is not in the Dictionary; insert it.
4. **A** is in the Dictionary.
   AB is not in the Dictionary; insert it.
5. **R** is not in the Dictionary; insert it.
6. **R** is in the Dictionary.
   RR is not in the Dictionary; insert it.
7. **A** is in the Dictionary and it is the last input character; output a pair containing its index: **(2,)**

Example 3: LZ78 Compression

Encode (i.e., compress) the string **AAAAAAAAA** using the LZ78 algorithm.



	Dictionary	
output	index	string
(0, A)	1	A
(1, A)	2	AA
(2, A)	3	AAA
(3,)		

1. A is not in the Dictionary; insert it
2. A is in the Dictionary
   AA is not in the Dictionary; insert it
3. A is in the Dictionary.
   AA is in the Dictionary.
   AAA is not in the Dictionary; insert it.
4. A is in the Dictionary.
   AA is in the Dictionary.
   AAA is in the Dictionary and it is the last pattern; output a pair containing its index: (3,)

LZ78 Compression: Number of bits transmitted

- Example: Uncompressed String: **ABBCBCABABCAABCAAB**

  Number of bits = Total number of characters * 8

  = 18 * 8

  = 144 bits

- Suppose the codewords are indexed starting from 1:

  Compressed string(codewords): **(0, A) (0, B) (2, C) (3, A) (2, A) (4, A) (6, B)**

Codeword index	**1**	**2**	**3**	**4**	**5**	**6**	**7**

• Each code word consists of an integer and a character:

- The character is represented by **8** bits.
- The number of bits **n** required to represent the integer part of the codeword with index **i** is given by:

$$n = \begin{cases} 1 & \text{if } i = 1 \\ \lceil \log_2 i \rceil & \text{if } i > 1 \end{cases}$$

• Alternatively, the number of bits required to represent the integer part of the codeword with index **i** is the number of significant bits required to represent the integer **i – 1**

LZ78 Compression: Number of bits transmitted (cont'd)

index	index - 1	bits	Number of significant bits
1	0	0	1
2	1	1	
3	2	10	2
4	3	11	
5	4	100	3
6	5	101	
7	6	110	
8	7	111	
9	8	1000	4
10	9	1001	
11	10	1010	
12	11	1011	
13	12	1100	
14	13	1101	
15	14	1110	
16	15	1111	

Codeword **(0, A) (0, B) (2, C) (3, A) (2, A) (4, A) (6, B)**

index 1 2 3 4 5 6 7

Bits: (1 + 8) + (1 + 8) + (2 + 8) + (2 + 8) + (3 + 8) + (3 + 8) + (3 + 8) = 71 bits

The actual compressed message is: 0A0B10C11A010A100A110B

where each character is replaced by its binary 8-bit ASCII code.

LZ78 Decompression Algorithm

```
Dictionary ← empty ; DictionaryIndex ← 1 ;
while(there are more (CodeWord, Char) pairs in codestream){
        CodeWord ← next CodeWord in codestream ;
        Char ← character corresponding to CodeWord ;
        if(CodeWord = = 0)
            String ← empty ;
        else
            String ← string at index CodeWord in Dictionary ;
        Output: String + Char ;
        insertInDictionary( (DictionaryIndex ,  String + Char) ) ;
      DictionaryIndex++;
}
```

Summary:

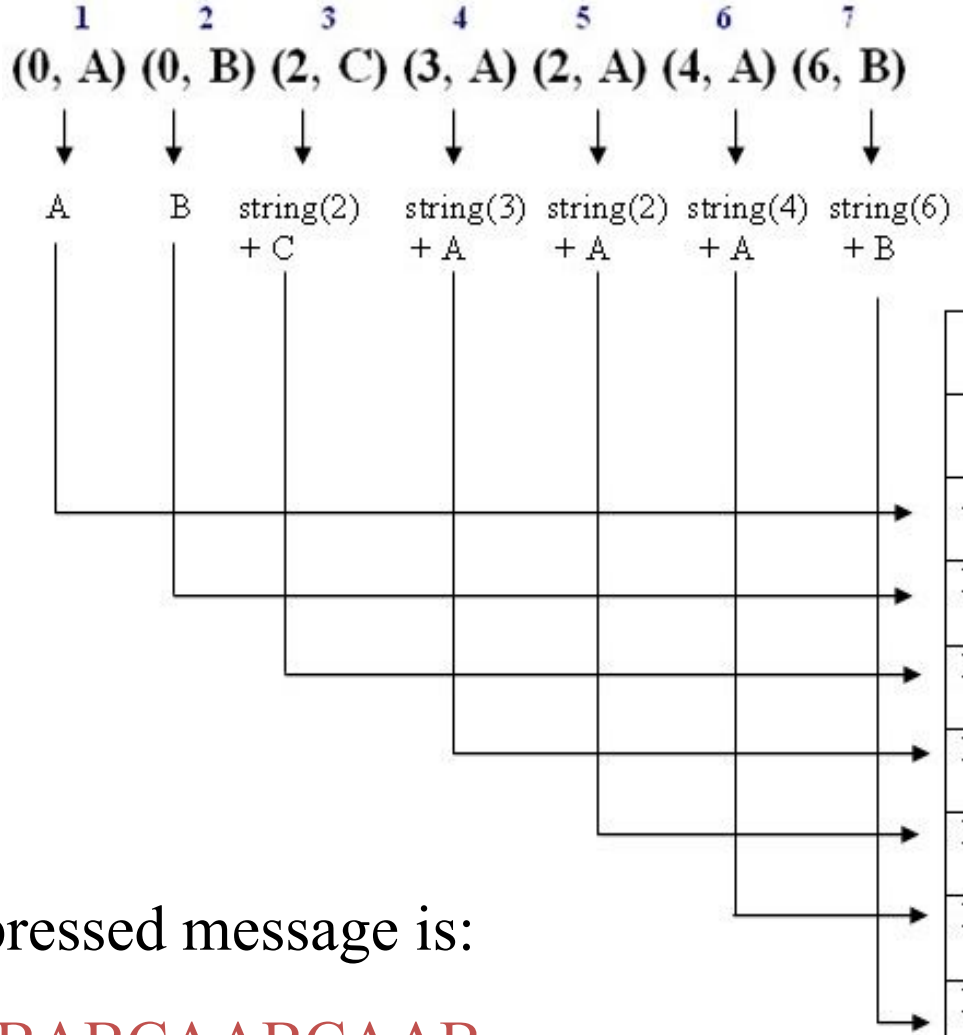
- **input:** (CW, character) pairs
- **output**:

```
if(CW == 0)
    output: currentCharacter
else
    output: stringAtIndex CW + currentCharacter
```

- **Insert:** current output in dictionary

Example 1: LZ78 Decompression

Decode (i.e., decompress) the sequence (0, A) (0, B) (2, C) (3, A) (2, A) (4, A) (6, B)



	Dictionary	
output	index	string
A	1	A
B	2	B
BC	3	BC
BCA	4	BCA
BA	5	BA
BCAA	6	BCAA
BCAAB	7	BCAAB

The decompressed message is:

ABBCBCABABCAABCAAB

Example 2: LZ78 Decompression

Decode (i.e., decompress) the sequence (0, B) (0, A) (1, A) (2, B) (0, R) (5, R) (2,)

	Dictionary	
output	index	string
B	1	B
A	2	A
BA	3	BA
AB	4	AB
R	5	R
RR	6	RR
A		

The decompressed message is: BABAABRRRA

Example 3: LZ78 Decompression

Decode (i.e., decompress) the sequence (0, A) (1, A) (2, A) (3,)

	Dictionary	
output	index	string
A	1	A
AA	2	AA
AAA	3	AAA
AAA		

The decompressed message is: AAAAAAAAAA

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.

Exercises

5. Use LZ78 to trace encoding the string

SATATASACITASA.

6. Write a Java program that encodes a given string using LZ78.

7. Write a Java program that decodes a given set of encoded codewords using LZ78.