

Chapter 5

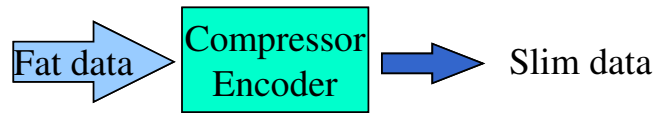
Data Compression: Entropy Coding

Storage Size for Media Types

	Text	Image	Audio	Video
Object Type	-ASCII -EBCDIC -BIG5 -GB	-Bitmapped graphics -Still photos -Faxes	-Noncoded stream of digitized audio or voice	-Digital video at 25-30 frames/s
Size and Bandwidth	2KB per page	-Simple: 1MB per image -Detailed (color) 7.5 MB per image	Voice/Phone 8kHz/8 bits (mono) 8-44 KB/s Audio CD 44.1 kHz/16 bits (stereo) 176 KB/s	27.7 MB/s for 640x480x24 pixels per frame 30 frame/s

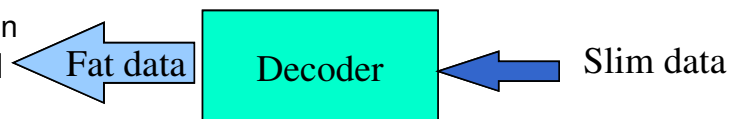
Compression

- We can view compression as an encoding method that transforms a sequence of bits into an another representation which takes less space than the input.



- The output data can be transformed back to the input representation without any loss (**lossless compression**) or with acceptable loss (**lossy compression**).

May be same
or approximation
of the original
data



3

Redundancy

- How can we represent the same information with lesser no of bits?
- Compression is made possible by exploiting redundancy in source data:

For example, redundancy found in:

- Character distribution
- Character repetition
- High-usage pattern
- Positional redundancy
(these categories overlap to some extent)

4



Redundancy (2)

■ **Character Distribution**

- Some characters are used more frequently than others.
- In English text, 'e' and *space* occur most often.
- Example: “abacabacabacabacabacabac”
- Technique: use variable length encoding.
- For your interest:
 - In 1838, Samuel Morse and Alfred Vail designed the “Morse Code” for telegraph.
 - Each letter of the alphabet is represented by dots (electric current of short duration) and dashes (3 dots).
 - e.g., E => “.” Z => “--..”
 - Morse’s principle: assign short code strings to frequently occurring letters and longer code to letters that occur less often.

5



Redundancy (3)

■ **Character Repetition**

- When a string of repetitions of a single character occurs, e.g., “aaaaaaaaaaaaabc”
- Example: long strings of spaces/tabs in forms/tables.
- Example: in graphics, it is common to have a line of pixels using the same color, e.g. fax
- Technique: Run-length encoding.

■ **High Usage Patterns**

- Certain patterns (e.g., sequences of characters) occur very often.
- Example: “qu”, “th”,.....
- Technique: Use a single special character to represent a common sequence of characters. $\theta = "th"$

6



Redundancy (4)

■ ***Positional Redundancy***

- A certain thing appears persistently at a predictable place(s).
- Example: the left-uppermost pixel of my slides is always white.
- Technique: Let me make a prediction, if wrong, tell me the “differences” only.

7



Categories of Compression Techniques

- One way to classify a compression method is to see whether it takes the source semantics into consideration.
- ***Entropy encoding***
 - Regard data stream as a simple digital sequence, its semantics is ignored.
 - Lossless compression.
- ***Source encoding***
 - Take into account the semantics of data and the characteristics of specific medium to achieve a higher compression ratio.
 - Lossless or lossy compression.

8



Categorization

Entropy Coding	Huffman coding	
	Run-length coding	
	Arithmetic coding	
	LZ algorithms	
Source Coding	Differential coding	DPCM
		DM
	Transform coding	FFT
		DCT
	Vector Quantization	
	Geometry Compression	

9



Run-Length Encoding

- Image, video and audio data streams often contain sequence of the same bytes. e.g. background in images, and silence in sound files.
- Substantial compression achieved by replacing repeated byte sequences with the number of occurrences.
- If a byte occurs at least 4 consecutive times, the sequence is encoded with the byte followed by a special flag and the number of its occurrence. e.g.,
 - Uncompressed: ABCCCCCCCCDEFGGGHI
 - Compressed: ABC!4DEFGGGHI
- If 4Cs (CCCC) is encoded as C!0, and 5Cs as C!1, and so on, then it allows compression of 4 to 259 repetitions into 3 bytes only.

10



Run-Length Encoding (2)

- Problems:
 - Ineffective with text
 - Q: Is RLE useful with: “ccc”?
 - Need an escape byte
 - OK with text
 - Nahhhh with data
 - Solution?
- Used in “Kermit” and CCITT Group 3 fax compression.

11



Variable-Length Code & Unique Decipherability

- Key: Use less bits for frequently used symbols and more bits for less used symbols
- This approach obviously reduces the total data size.
- Problems:
 - How to assign codes?
 - Can we uniquely decipher the code?

Source Letter	Letter Probability	Binary Codeword
a_1	0.40	0
a_2	0.15	1
a_3	0.15	00
a_4	0.10	01
a_5	0.10	10
a_6	0.05	11
a_7	0.04	000
a_8	0.01	001

You cannot decode the code uniquely!!

12



Prefix Code

- The idea is to translate symbols of input data into variable-length codes.
- We wish to encode each symbol into a sequence of 0's and 1's so that no code of the symbol is the prefix of the code of any other symbol -- the **prefix condition**.
- Hence, no special delimiter is needed.
- With the prefix property, we are able to **uniquely** decode a sequence of 0,1 into a sequence of characters.

Source Letter	Letter Probability	Binary Codeword
a_1	0.40	010
a_2	0.15	011
a_3	0.15	00
a_4	0.10	100
a_5	0.10	101
a_6	0.05	110
a_7	0.04	1110
a_8	0.01	1111

13



Huffman Code

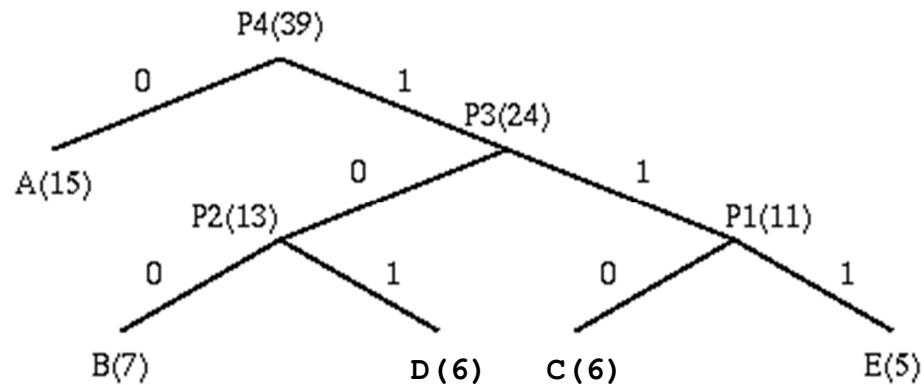
- A simple method that generates a kind of prefix code - Huffman code.
- Used in many compression programs, e.g., pkzip, arj.
- Algorithm (**bottom-up** approach):
 - Assume the probabilities (frequencies) of symbols used are known
 - Label each node with its correspondence probability and put all nodes into the candidate list.
 - Pick two nodes with smallest probabilities, create a parent to connect them and label this parent with the sum of probabilities of these two children.
 - Put the parent node into the candidate list.
 - Repeat the last two steps until one node (root) left in candidate list.
 - Assign 0 and 1 to left and right branches of the tree in the candidate list.

14

Huffman Code (2)

- Example 1: 5 symbols are used in the source

Symbol	A	B	C	D	E
Count	15	7	6	6	5
Code	0	100	110	101	111

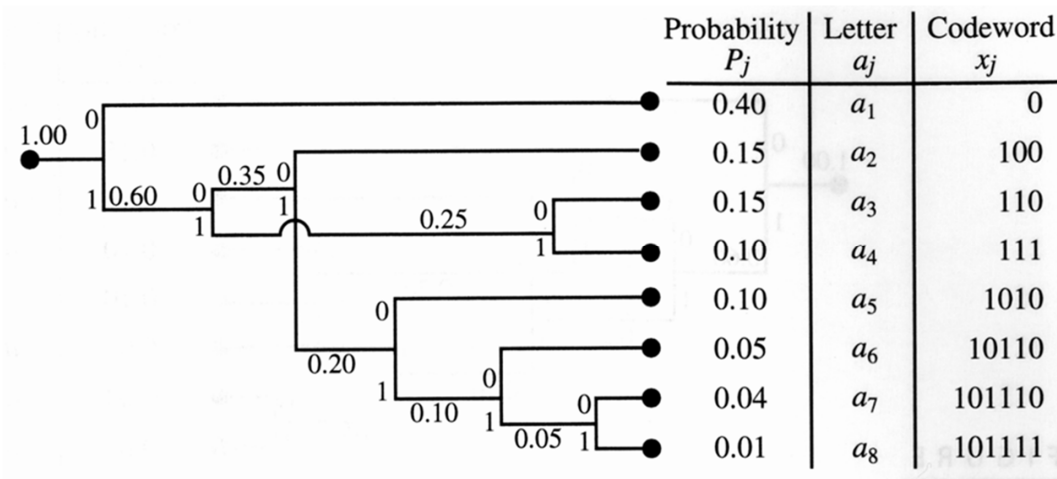


- Code assignment may not be unique.

15

Huffman Code (3)

- More example 2:



16



Huffman Coding (Advantages)

- Decoding is trivial as long as the coding table is sent before the data. Overhead is relatively negligible if the data file is big.
- Unique Decipherable:
 - No code is a prefix to any other code. Forward scanning is never ambiguous. Good for decoder.
- Average bit per symbol \bar{l} (expected no of code symbols per source symbol) for previous examples

Ex 1: $\bar{l} = [15*1 + (7+6+6+5)*3] / 39 = 2.231$

Ex 2: $\bar{l} = 0.40*1 + (0.15+0.15+0.10)*3 + 0.10*4 + 0.05*5 + (0.04+0.01)*6 = 2.55$

Optimal for instantaneous coding (code without buffering) ¹⁷

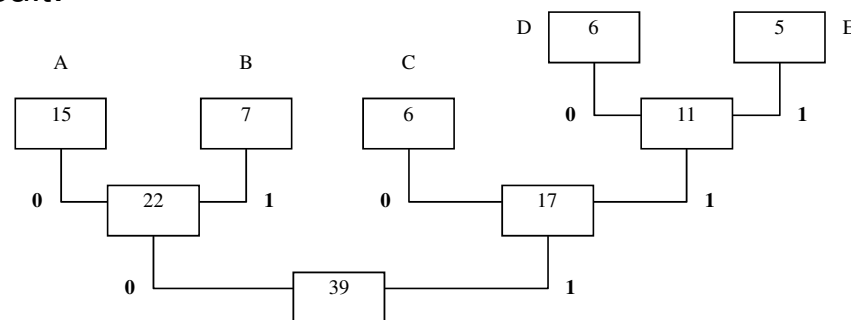


Huffman Code (Disadvantages)

- A table is needed that lists each input symbol and its corresponding code.
- Need to know the character frequency distribution in advance => need two passes over the data.
- More seriously, it does not explore the coherence between symbols. You cannot group a set of symbols and output one single code for them, e.g. pattern “the” is usually used in English.

Shannon-Fano Algorithm

- Similar to Huffman code
- Algorithm (top-down approach):
 - 1. Sort symbols according to their probabilities.
 - 2. Recursively divide symbols into 2 half-sets, each with roughly equal sum of probabilities.
- Result:



19

Shannon-Fano Algorithm (2)

Symbol	Count	Code	Bits used
A	15	00	30
B	7	01	14
C	6	10	12
D	6	110	18
E	5	111	15

- Average bit per symbol

$$\bar{l} = [(15+7+6)*2 + (6+5)*3] / 39 = 2.282$$

- How good we can compress?
- Let's measure it with the lower bound of the average bit per symbol.

20



How small we can compress?

- Entropy is a measure of uncertainty or randomness.
- According to Shannon (father of information theory), the entropy of an information source S with the probability distribution $\{p_j\}$ is defined as:

$$H(\{p_j\}) = -\sum_j p_j \log_2 p_j$$

– where p_j is the probability that symbol S_j in S will occur.

- So what?
- The average codeword length \bar{l} of a binary code is always **at least as great as the source entropy** calculated, i.e.

$$\bar{l} \geq H$$

21



Entropy

- The entropy is the lower bound for *lossless compression*: when the probability of each symbol is fixed, each symbol should be represented at least with H bits on average.
- Proof: l_j be the length of code word for symbol S_j

$$\begin{aligned}\bar{l} - H &= \sum_j p_j (l_j + \log_2 p_j) \\ &= \sum_j p_j \log_2 (2^{l_j} p_j) \geq \log_2 e \sum_j p_j \left(1 - \frac{2^{-l_j}}{p_j}\right) \\ &= \log_2 e \sum_j (p_j - 2^{-l_j}) \\ &= \log_2 e \left(1 - \sum_j 2^{-l_j}\right) \geq \log_2 e (1 - 1) = 0\end{aligned}$$

Mathematical fact:

$$\log_D(x) \geq \log_D(e) \left(1 - \frac{1}{x}\right)$$

Kraft inequality:

$$\sum_j 2^{-l_j} \leq 1$$

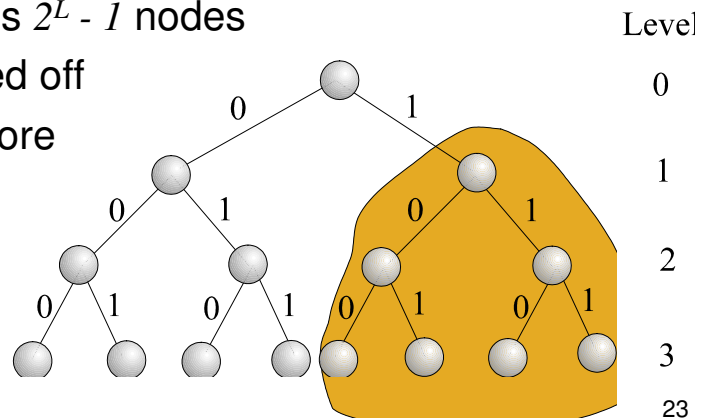
22

Kraft Inequality

- Given a binary code with word lengths $\{l_j\}$, we may embed it in a binary tree. (e.g. Huffman tree)
- Assign to each codeword of length l_j a node on level l_j to serve as its terminal nodes.
- This prunes from the tree the entire subtree that previously stemmed from this node.
- A L -level tree contains $2^L - 1$ nodes
- 2^{L-l_j} nodes are pruned off
- Since there are no more than 2^L nodes, so

$$\sum_j 2^{L-l_j} \leq 2^L$$

$$\sum_j 2^{-l_j} \leq 1$$



Entropy

- Q: How about an image in which half of the pixels are white, another half are black?
 - Ans: $p(\text{white}) = 0.5$, $p(\text{black}) = 0.5$, so entropy is 1.
- Let check Example 2 in the slides of Huffman code

$$\{p_j\} = \{0.15, 0.4, 0.15, 0.1, 0.1, 0.05, 0.04, 0.01\}$$

$$H = -0.15\log_2(0.15) - 0.4\log_2(0.4) - 0.15\log_2(0.15) - 0.1\log_2(0.1) - 0.1\log_2(0.1) - 0.05\log_2(0.05) - 0.04\log_2(0.04) - 0.01\log_2(0.01) = 2.483$$

$$\bar{l} = 0.40*1 + (0.15+0.15+0.10)*3 + 0.10*4 + 0.05*5 + (0.04+0.01)*6 = 2.55$$
- Again, $\bar{l} \geq H$



Universal Loseless Compression

- Isn't Huffman code optimal? Why bother with others?
- By observation, there are still gaps between entropy and the average length of Huffman codes. Why?
- **Problem 1**: The statistics have to be known in advance
- **Problem 2**: Group of symbols cannot be coded as one codeword to further reduce average code length.
- Huffman code is optimal as an instantaneous code (no buffering is allowed).
- A coding scheme is **universal** if, it succeeds in compressing that data down to the entropy rate of source despite having no a priori knowledge of the source statistics.

25



Lempel-Ziv Algorithms

- Proposed by A. Lempel and J. Ziv.
- Variants:
 - LZ77, proposed in 1977, used in pkzip, gzip
 - LZ78, proposed in 1978
 - LZW, improved by Terry A. Welch in 1984, used in compress, GIF. It is patented by Unisys Corporation.
 - LZY, developed by Yakoo in 1992
- Probably the most widely-used loseless compression methods.
- **All** approach H (entropy bound) when input data size is large.
- No data statistics is needed in advance.
- Universal

26



LZ Algorithms (2)

- Motivation:
 - Huffman code cannot encode multiple input symbols by a single codeword. Hence, a lot of patterns (e.g. “the”, “of”, “an” in English are regular patterns) are frequently seen. Can we do something on them?
- May be we can build a dictionary of all frequently seen pattern and encode them with the table index.
- But, how to build the dictionary if we have no knowledge?
- LZ methods build the dictionary (or table or tree) on-the-fly as the input symbols are encoded.
- Let’s illustrate the idea by introducing LZ78 and then LZW.

27



LZ78

- Let’s consider an input source with only two symbols “a” and “b”. $S = \{a, b\}$
- The basic idea is to construct a LZ78 tree. (“dictionary”)
- Since there are only two alphabets, the LZ78 tree is binary in this case.
- The tree starts as a single root node “R”. It grows as more input data enter.
- Let’s further assume the input data stream is
a a a b a b b a a a a b a b a a b a a a b a a b
- LZ78 will break the stream into phrases, each will then be encoded by a codeword (“dictionary index”)
a, a a, b, a b, b a, a a a, a b a, b a a, b a a b, a a a b, a a b
- How? Let’s animate the process!

28

LZ78 (2)

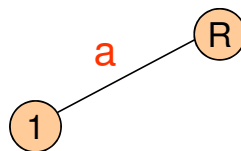
R

a a a b a b b a a a a b a b a a b a a b a a a b a a b



29

LZ78 (2)



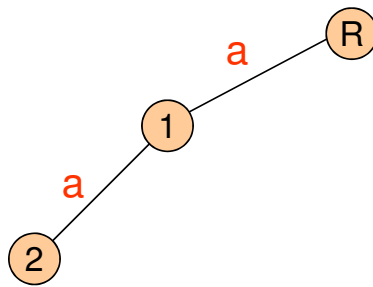
a, a a b a b b a a a a b a b a a b a a b a a a b a a b



output: R, a

30

LZ78 (2)



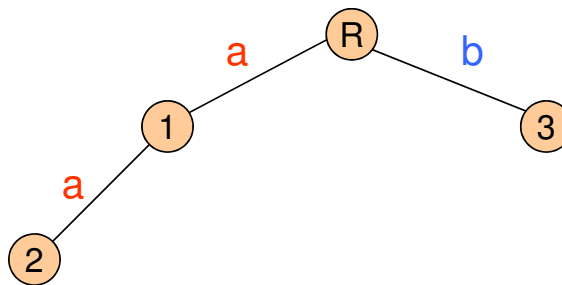
a, a a, b a b b a a a a b a b a a b a a b a a a b a a b



output: R, a, 1, a,

31

LZ78 (2)



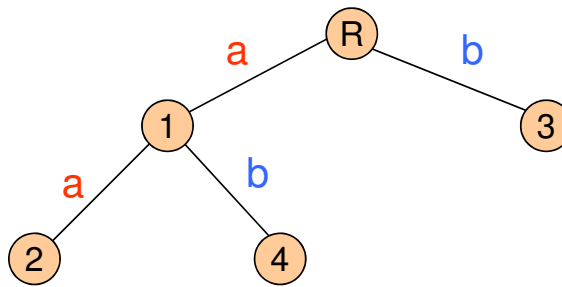
a, a a, b, a b b a a a a a b a b a a b a a b a a a b a a b



output: R, a, 1, a, R, b,

32

LZ78 (2)



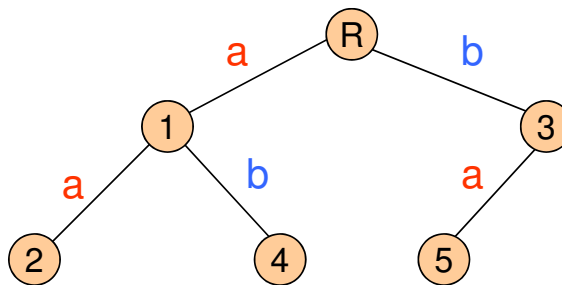
a, a a, b, a b, b a a a a b a b a a b a a b a a a b a a b



output: R, a, 1, a, R, b, 1, b,

33

LZ78 (2)



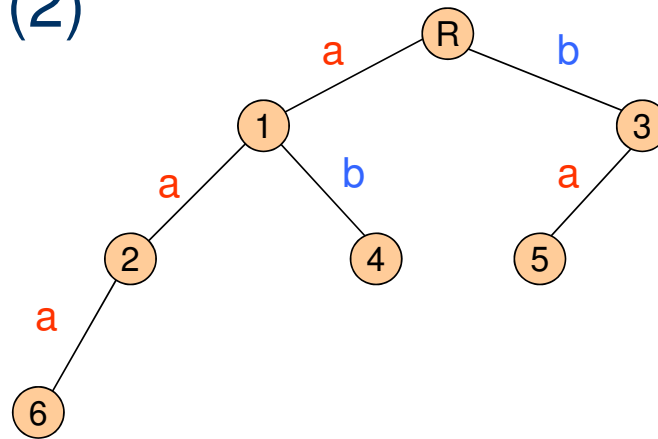
a, a a, b, a b, b a, a a a a b a b a a b a a b a a a b a a b



output: R, a, 1, a, R, b, 1, b, 3, a,

34

LZ78 (2)



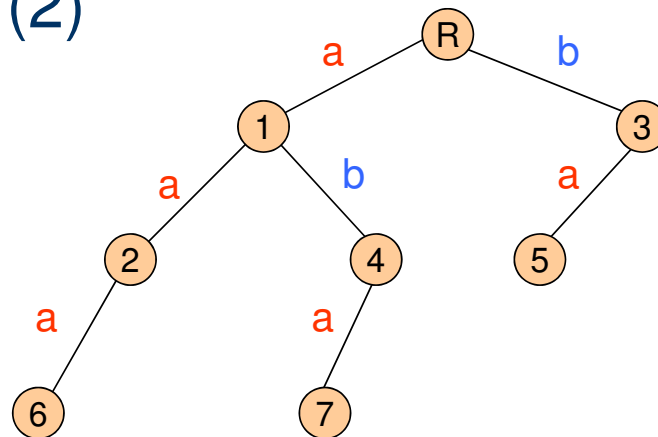
a, a a, b, a b, b a, a a a, a b a b a a b a a a b a a b



output: R, a, 1, a, R, b, 1, b, 3, a, 2, a,

35

LZ78 (2)



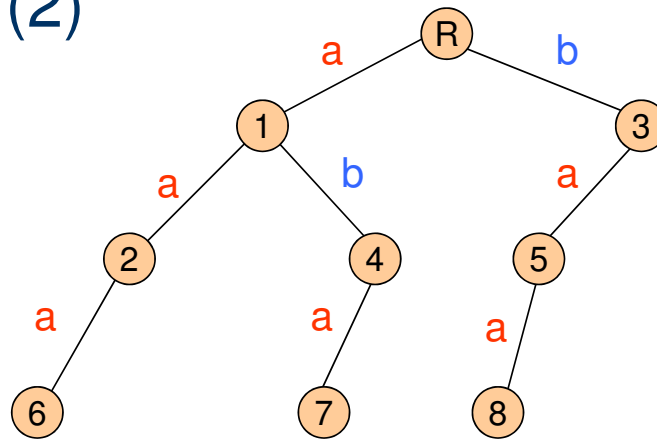
a, a a, b, a b, b a, a a a, a b a, b a a b a a b a a a b a a b



output: R, a, 1, a, R, b, 1, b, 3, a, 2, a, 4, a,

36

LZ78 (2)



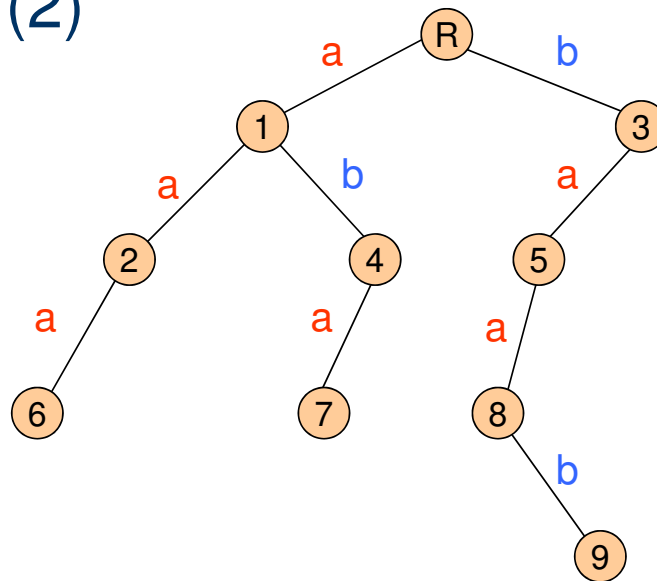
a, a a, b, a b, b a, a a a, a b a, b a a, b a a b a a a b a a b



output: R, a, 1, a, R, b, 1, b, 3, a, 2, a, 4, a, 5, a,

37

LZ78 (2)



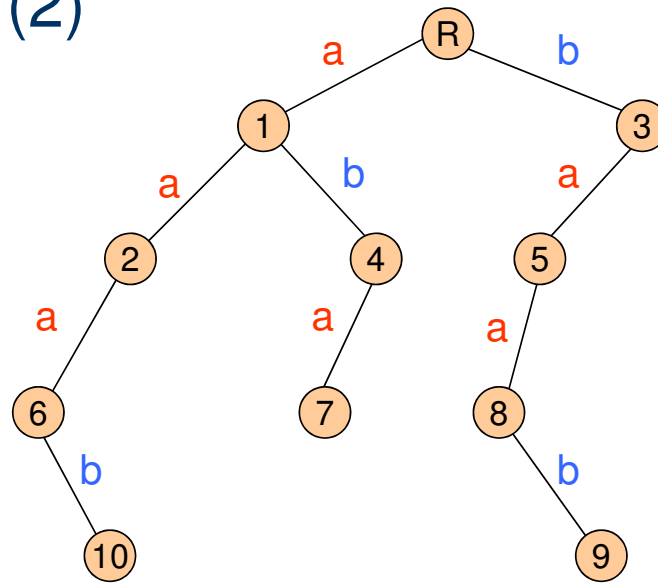
a, a a, b, a b, b a, a a a, a b a, b a a, b a a b, a a a b a a b



output: R, a, 1, a, R, b, 1, b, 3, a, 2, a, 4, a, 5, a, 8, b,

38

LZ78 (2)



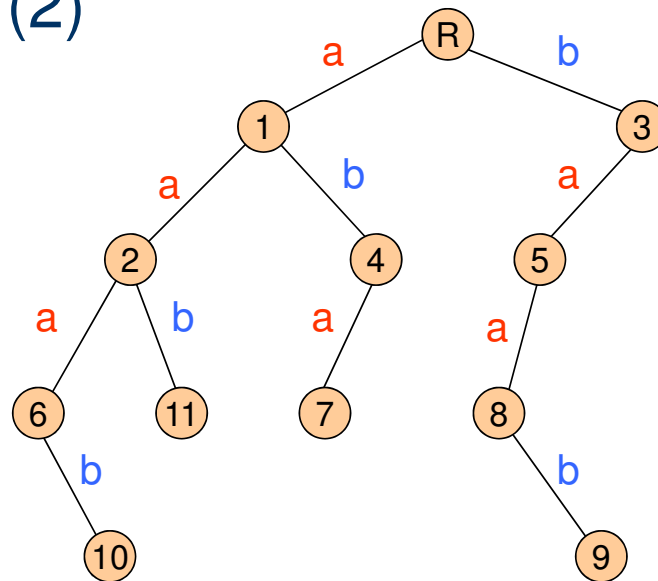
a, a a, b, a b, b a, a a a, a b a, b a a, b a a b, a a a b, a a b



output: R, a, 1, a, R, b, 1, b, 3, a, 2, a, 4, a, 5, a, 8, b, 6, b,

39

LZ78 (2)



a, a a, b, a b, b a, a a a, a b a, b a a, b a a b, a a a b, a a b,



output: R, a, 1, a, R, b, 1, b, 3, a, 2, a, 4, a, 5, a, 8, b, 6, b, 2, b,

40



LZ78 (3)

- Each time the phrase is constructed by looking up the LZ78 tree find the maximal match from the root node.
- Output the terminal node that matches the input string.
- Output the last symbol in the input string.
- What do you find?
- The matched string will become longer and longer as more symbols are entered.
- It does not work well for short input string like the example
- Asymptotically, the compression ratio approaches H
- Previous slides show how it is encoded, how about decoding?
- Have we output the LZ78 tree to the decoder?
- Yes, we **implicitly** sent the LZ78 tree

41



LZ78 (4)

R

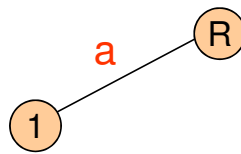
DECODER SIDE:

input: R, a, 1, a, R, b, 1, b, 3, a, 2, a, 4, a, 5, a, 8, b, 6, b, 2, b,
↑
output:

42

LZ78 (4)

DECODER SIDE:



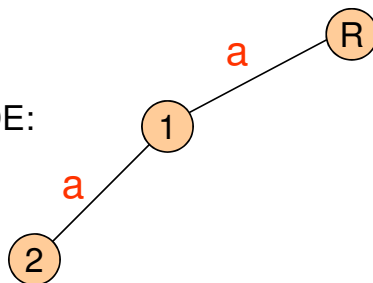
input: R, **a**, 1, **a**, R, **b**, 1, **b**, 3, **a**, 2, **a**, 4, **a**, 5, **a**, 8, **b**, 6, **b**, 2, **b**,

output: **a**

43

LZ78 (4)

DECODER SIDE:



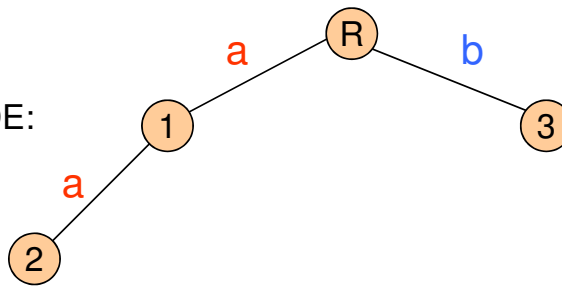
input: R, **a**, 1, **a**, R, **b**, 1, **b**, 3, **a**, 2, **a**, 4, **a**, 5, **a**, 8, **b**, 6, **b**, 2, **b**,

output: **a a a**

44

LZ78 (4)

DECODER SIDE:



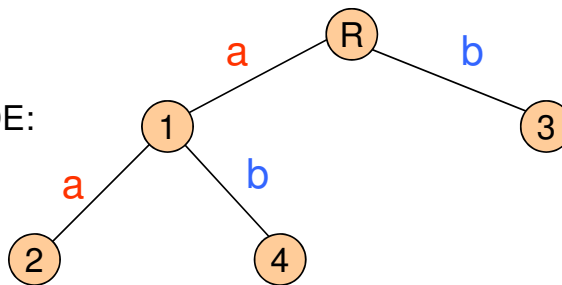
input: R, a, 1, a, R, b, 1, b, 3, a, 2, a, 4, a, 5, a, 8, b, 6, b, 2, b,

output: a a a b

45

LZ78 (4)

DECODER SIDE:



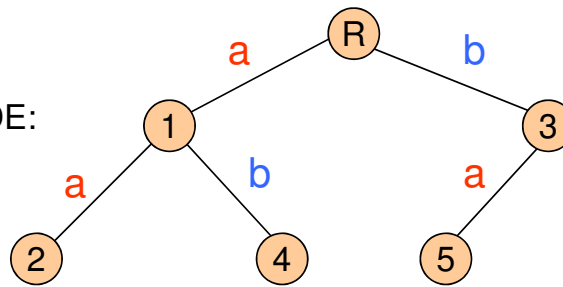
input: R, a, 1, a, R, b, 1, b, 3, a, 2, a, 4, a, 5, a, 8, b, 6, b, 2, b,

output: a a a b a b

46

LZ78 (4)

DECODER SIDE:



input: R, a, 1, a, R, b, 1, b, 3, a, 2, a, 4, a, 5, a, 8, b, 6, b, 2, b,

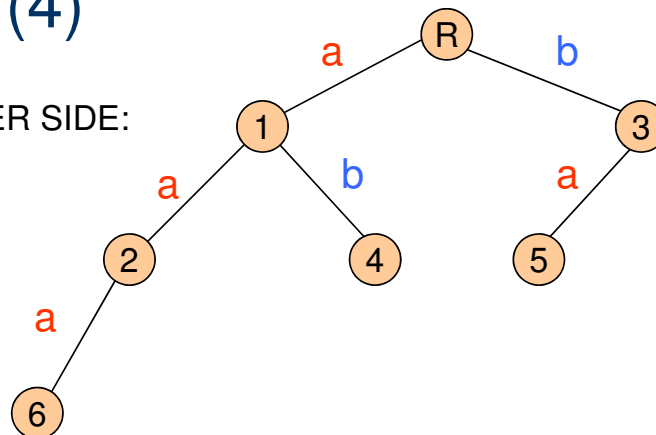


output: a a a b a b b a

47

LZ78 (4)

DECODER SIDE:



input: R, a, 1, a, R, b, 1, b, 3, a, 2, a, 4, a, 5, a, 8, b, 6, b, 2, b,

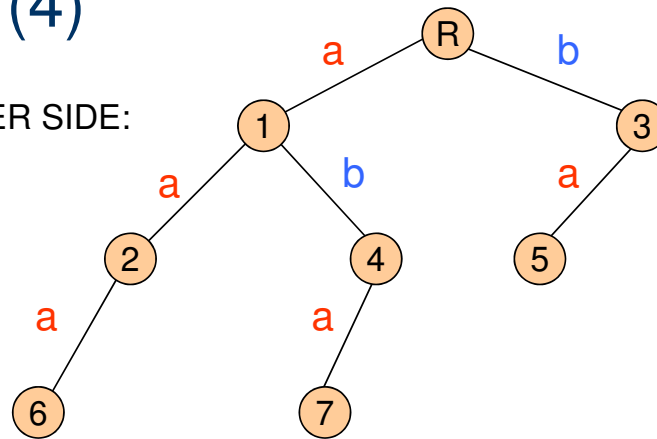


output: a a a b a b b a a a a

48

LZ78 (4)

DECODER SIDE:

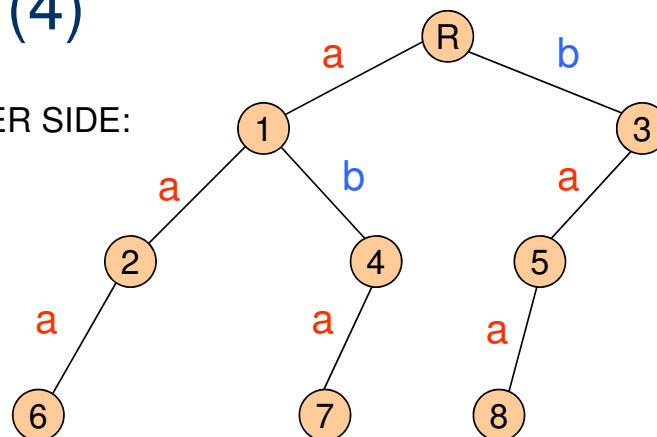


input: R, a, 1, a, R, b, 1, b, 3, a, 2, a, 4, a, 5, a, 8, b, 6, b, 2, b,
 output: a a a b a b b a a a a b a

49

LZ78 (4)

DECODER SIDE:

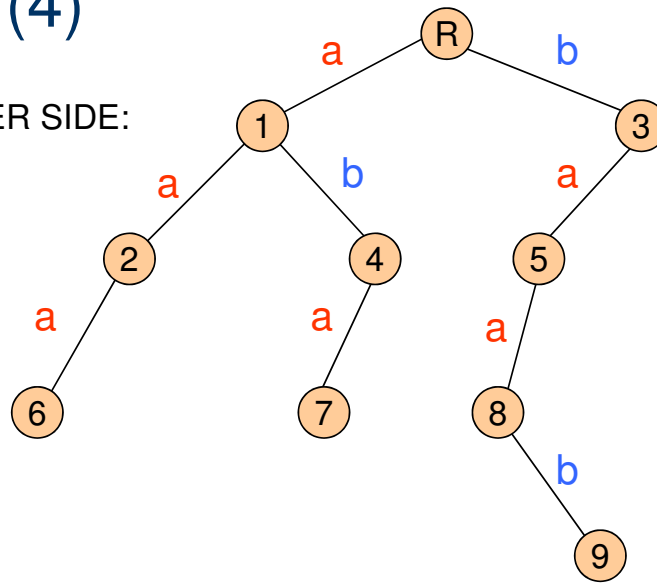


input: R, a, 1, a, R, b, 1, b, 3, a, 2, a, 4, a, 5, a, 8, b, 6, b, 2, b,
 output: a a a b a b b a a a a b a b a a

50

LZ78 (4)

DECODER SIDE:



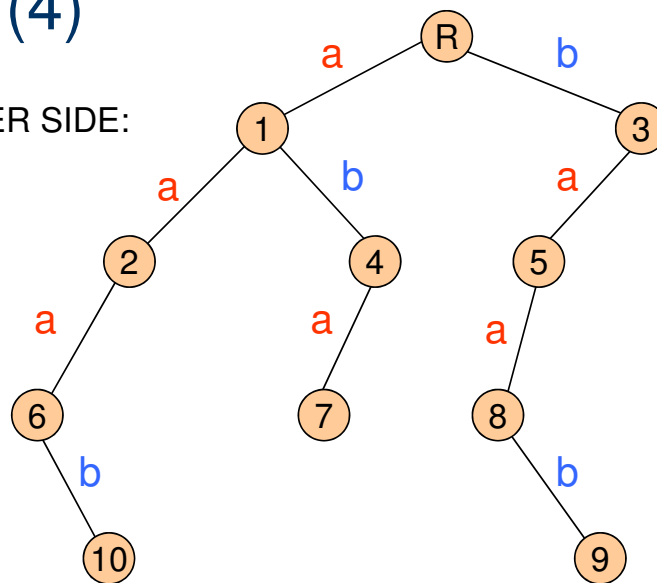
input: R, a, 1, a, R, b, 1, b, 3, a, 2, a, 4, a, 5, a, 8, b, 6, b, 2, b,

output: a a a b a b b a a a a b a b a a b a a b

51

LZ78 (4)

DECODER SIDE:



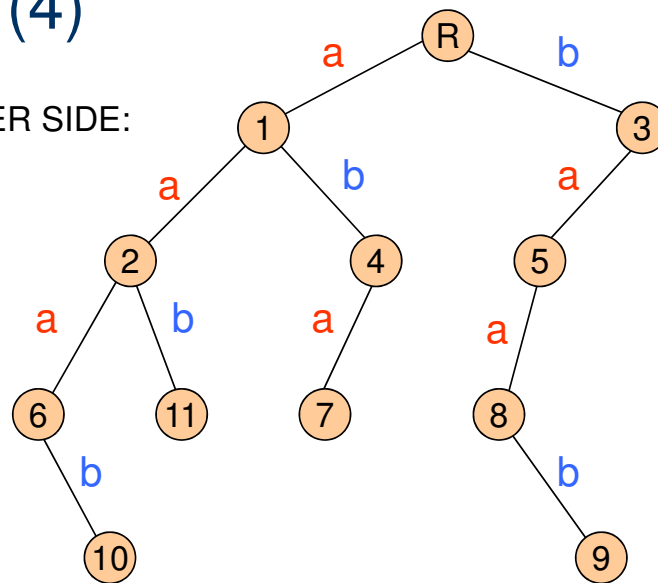
input: R, a, 1, a, R, b, 1, b, 3, a, 2, a, 4, a, 5, a, 8, b, 6, b, 2, b,


output: a a a b a b b a a a a b a b a a b a a b a a a b

52

LZ78 (4)

DECODER SIDE:



input: R, a, 1, a, R, b, 1, b, 3, a, 2, a, 4, a, 5, a, 8, b, 6, b, 2, b, 

output: a a a b a b b a a a a b a b a a b a a b a a a b a a b

53

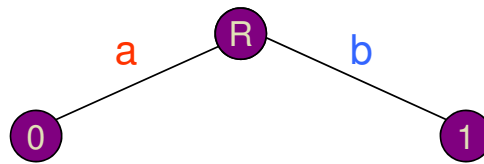
LZW

- In fact, LZ78 is already very good.
- We send the extra symbol in uncompressed form.
- Can we do something even better?
- This motivates the development of LZW
- The initial LZW tree contains root, a-descendant and b-descendant, i.e. 3 nodes instead of 1 in LZ78
- Encoder updates the LZW tree by adding the node corresponding to one-symbol extension of current phrase (Look ahead)
- The extra symbol is not part of current phrase
- The decoder will deduce this mystery extra symbol
- Let's reuse the previous example

54

LZW (2)

ENCODER SIDE:



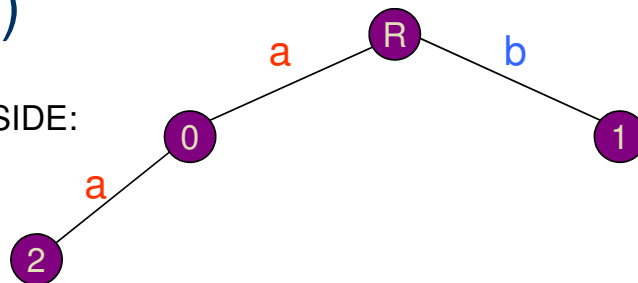
a a a b a b b a a a a b a b a a b a a b a a a b a a b



55

LZW (2)

ENCODER SIDE:



a a a b a b b a a a a b a b a a b a a b a a a b a a b

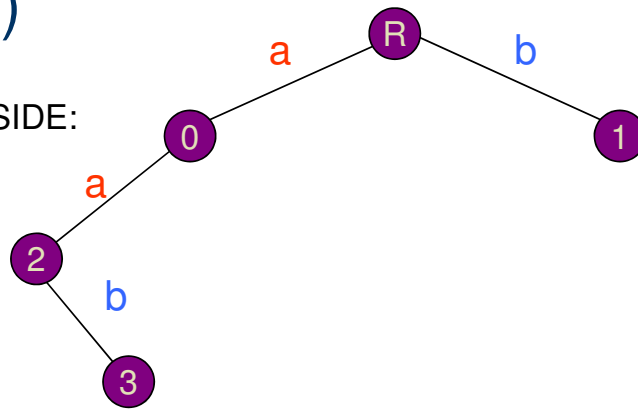


output: 0,

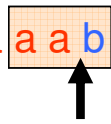
56

LZW (2)

ENCODER SIDE:



a a a b a b b a a a a b a b a a b a a a b a a b

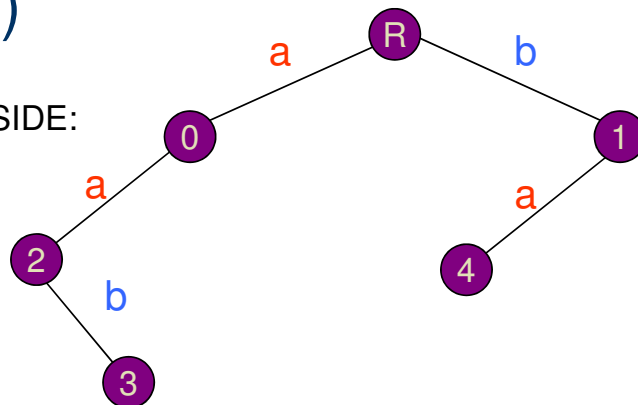


output: 0, 2,

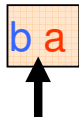
57

LZW (2)

ENCODER SIDE:



a a a b a b b a a a a b a b a a b a a a b a a b

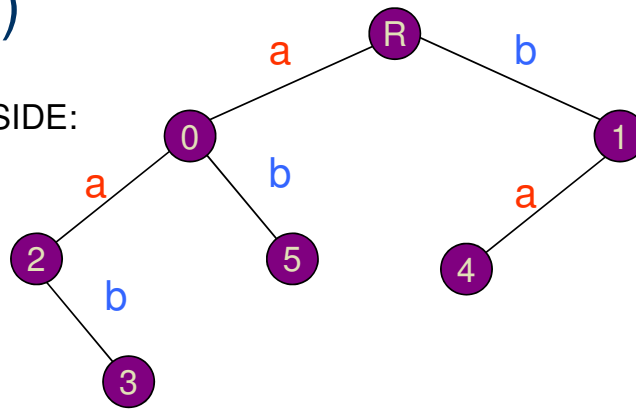


output: 0, 2, 1,

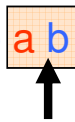
58

LZW (2)

ENCODER SIDE:



a a a b a b a a a a b a b a a b a a a b a a b

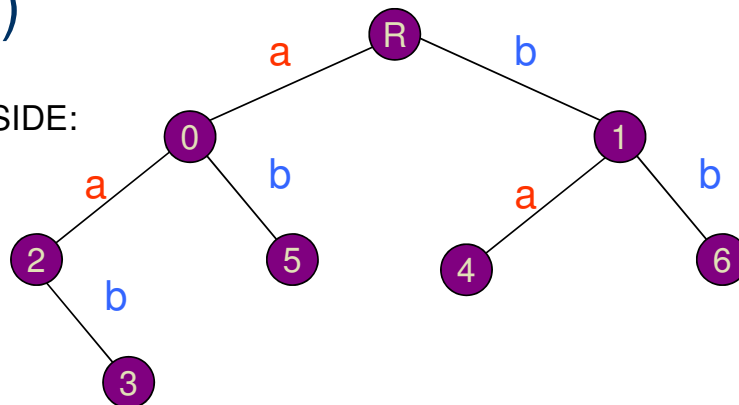


output: 0, 2, 1, 0,

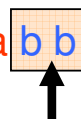
59

LZW (2)

ENCODER SIDE:



a a a b a b b a a a a b a b a a b a a a b a a b

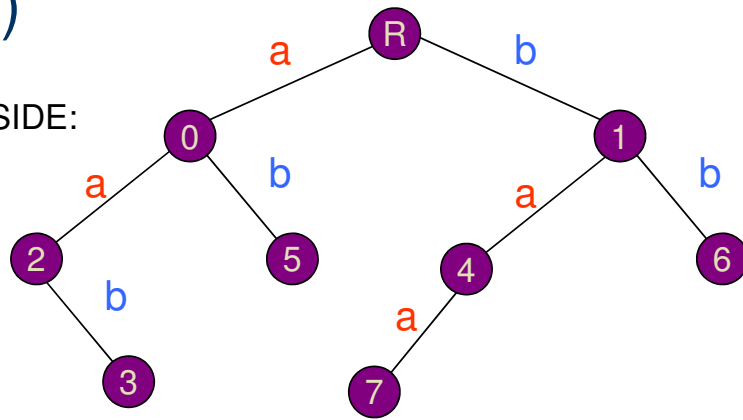


output: 0, 2, 1, 0, 1

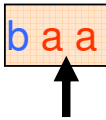
60

LZW (2)

ENCODER SIDE:



a a a b a b b a a a a b a b a a b a a a b a a b

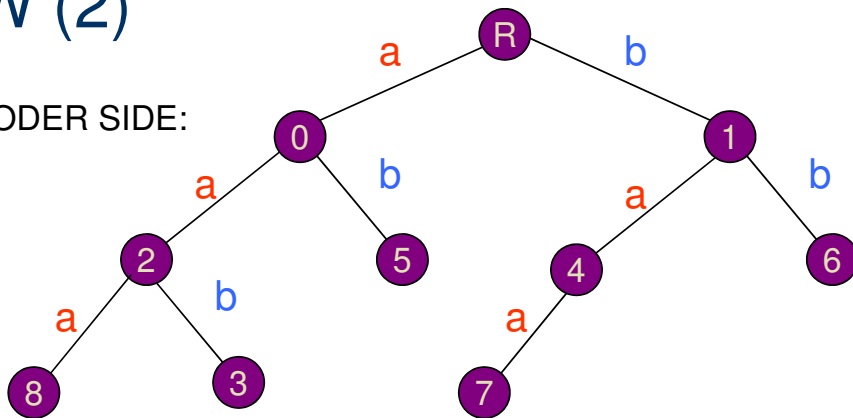


output: 0, 2, 1, 0, 1, 4

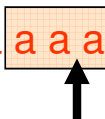
61

LZW (2)

ENCODER SIDE:



a a a b a b b a a a a b a b a a b a a a b a a b

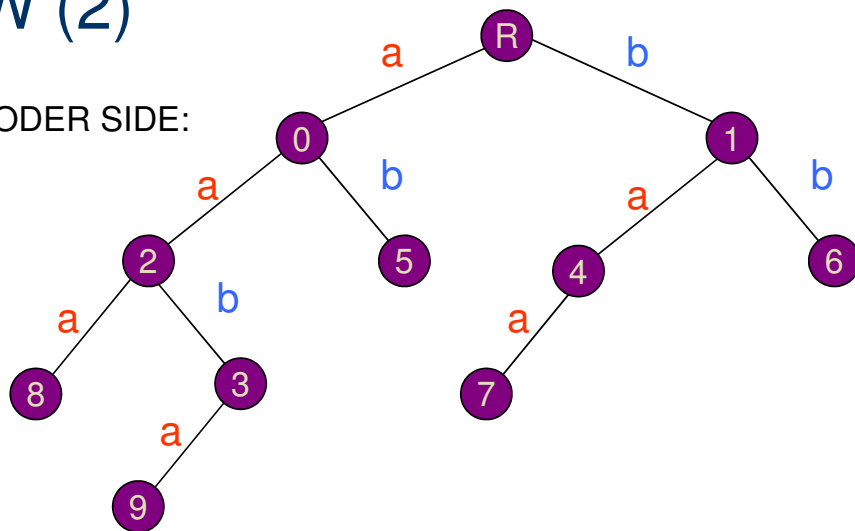


output: 0, 2, 1, 0, 1, 4, 2

62

LZW (2)

ENCODER SIDE:



a a a b a b b a a a a b a b a a b a a b a a b

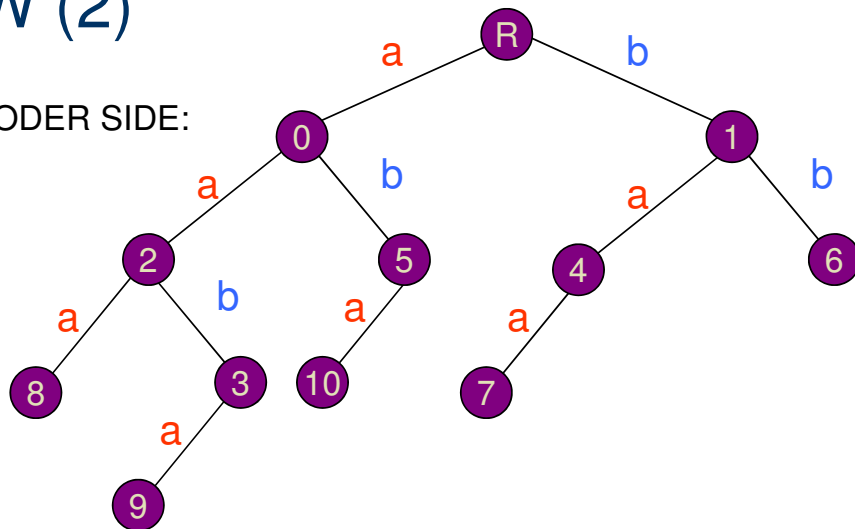


output: 0, 2, 1, 0, 1, 4, 2, 3

63

LZW (2)

ENCODER SIDE:



a a a b a b b a a a a b a b a b a a b a a b

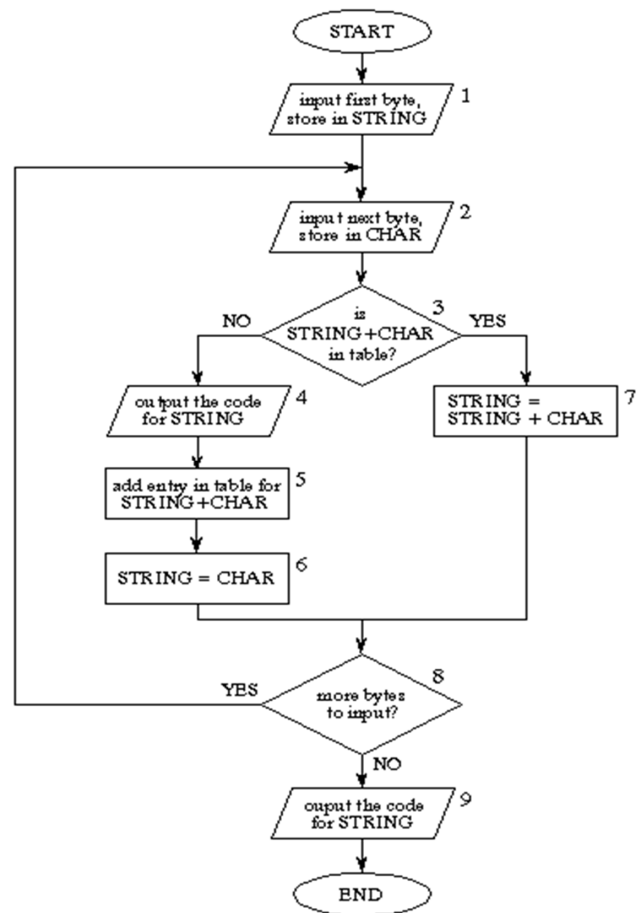


output: 0, 2, 1, 0, 1, 4, 2, 3, 5

64

LZW (2)

- In fact, the algorithm of the encoder is very simple.

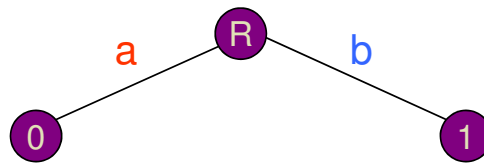


LZW (3)

- OK, try to encode the rest yourself.
- Note the output only contains codewords, no symbol is sent in uncompressed form.
- Let's see how can we decode. This will be a little bit tricky.
- Again we start with a LZW tree containing 3 nodes.
- Let's use a string variable "Last string" to memorize the last decoded string

LZW (4)

DECODER SIDE:



input: 0, 2, 1, 0, 1, 4, 2, 3, 5

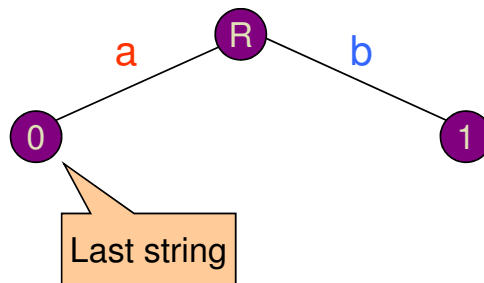


output:

67

LZW (4)

DECODER SIDE:



input: 0, 2, 1, 0, 1, 4, 2, 3, 5



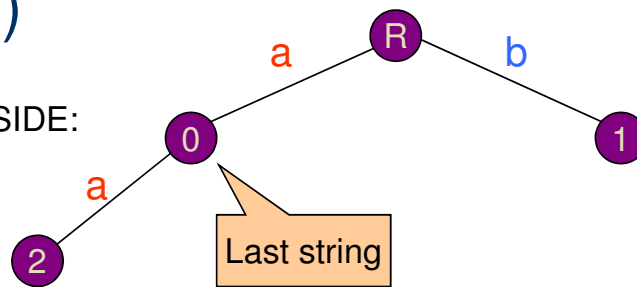
output: **a**

Last string = **a**

68

LZW (4)

DECODER SIDE:



What! Where is node 2?

Don't panic

There can be only one situation it does not exist

The first symbol of the string represented by "2" must be **a**, otherwise we should receive "1".

Moreover, this first **a** must also serve as the "look-ahead" symbol when the encoder constructs this new node. So the string must be **aa**

input: 0, 2, 1, 0, 1, 4, 2, 3, 5

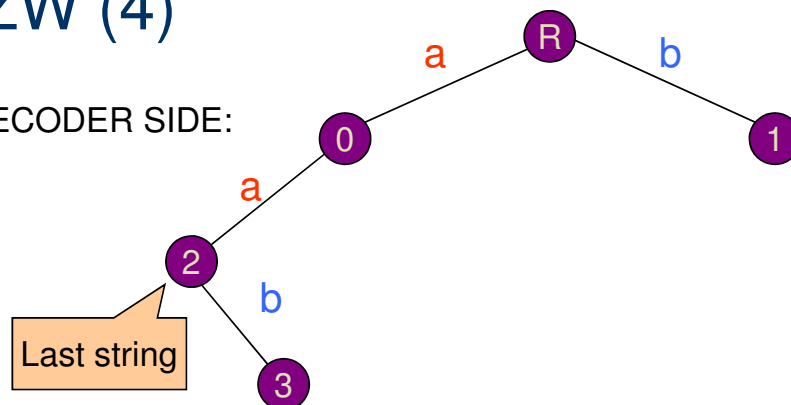
output: **a a a**

Last string = **a**

69

LZW (4)

DECODER SIDE:



Then we receive 1, so the encoded string is **b**.

The encoder must try to look for last string+**b** before. So we can insert node 3.

input: 0, 2, 1, 0, 1, 4, 2, 3, 5

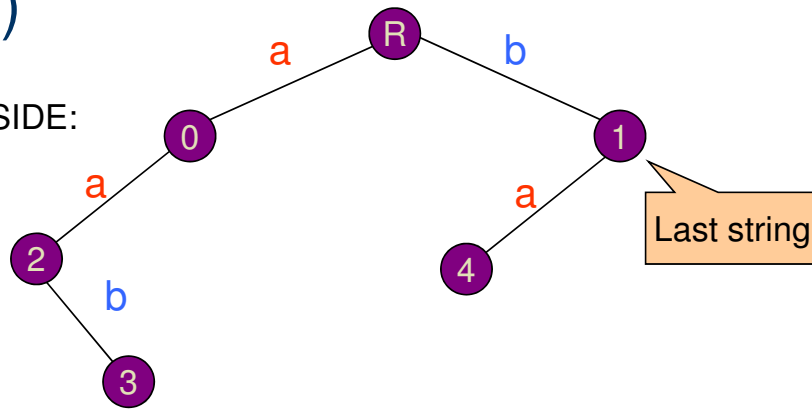
output: **a a a b**

Last string = **a a**

70

LZW (4)

DECODER SIDE:



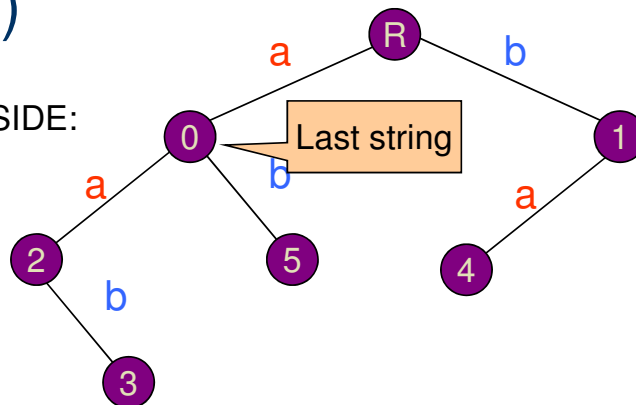
input: 0, 2, 1, 0, 1, 4, 2, 3, 5

output: a a a b a Last string = b

71

LZW (4)

DECODER SIDE:



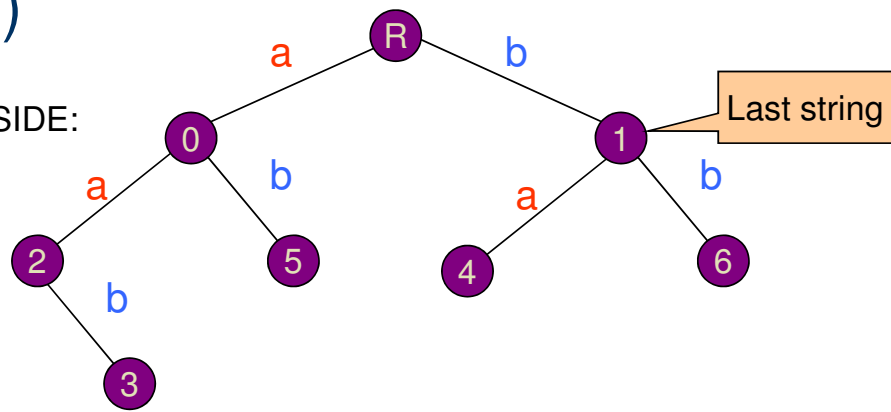
input: 0, 2, 1, 0, 1, 4, 2, 3, 5

output: a a a b a b Last string = a

72

LZW (4)

DECODER SIDE:



input: 0, 2, 1, 0, 1, 4, 2, 3, 5

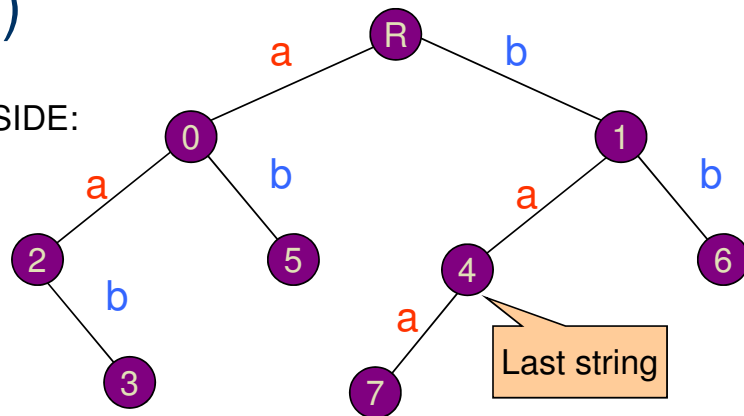
Last string = b

output: a a a b a b b a

73

LZW (4)

DECODER SIDE:



input: 0, 2, 1, 0, 1, 4, 2, 3, 5

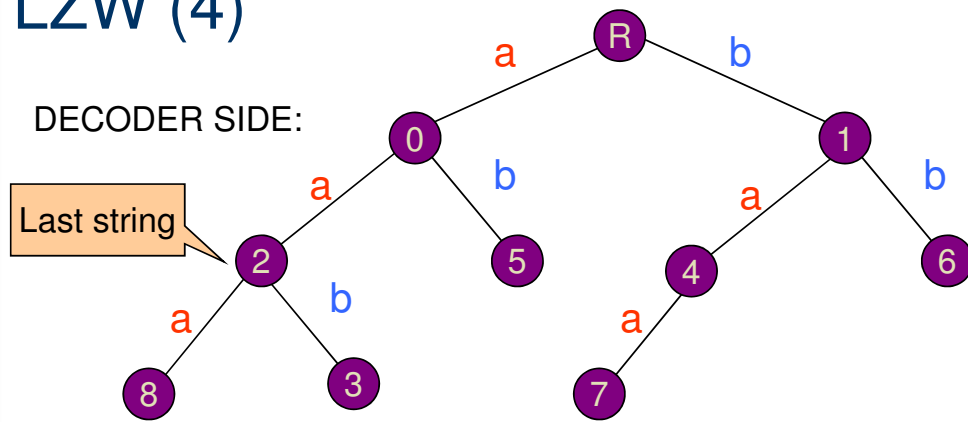
Last string = b a

output: a a a b a b b a a a

74

LZW (4)

DECODER SIDE:



input: 0, 2, 1, 0, 1, 4, 2, 3, 5



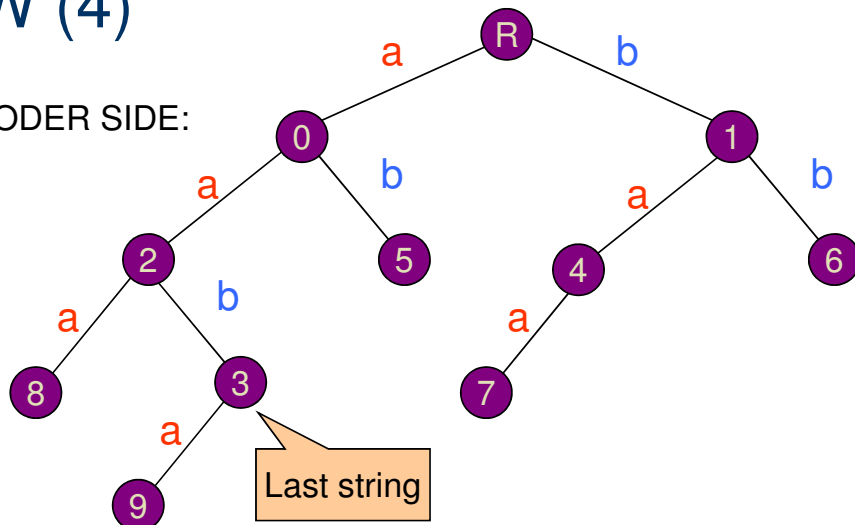
Last string = a a

output: a a a b a b b a a a a a b

75

LZW (4)

DECODER SIDE:



input: 0, 2, 1, 0, 1, 4, 2, 3, 5



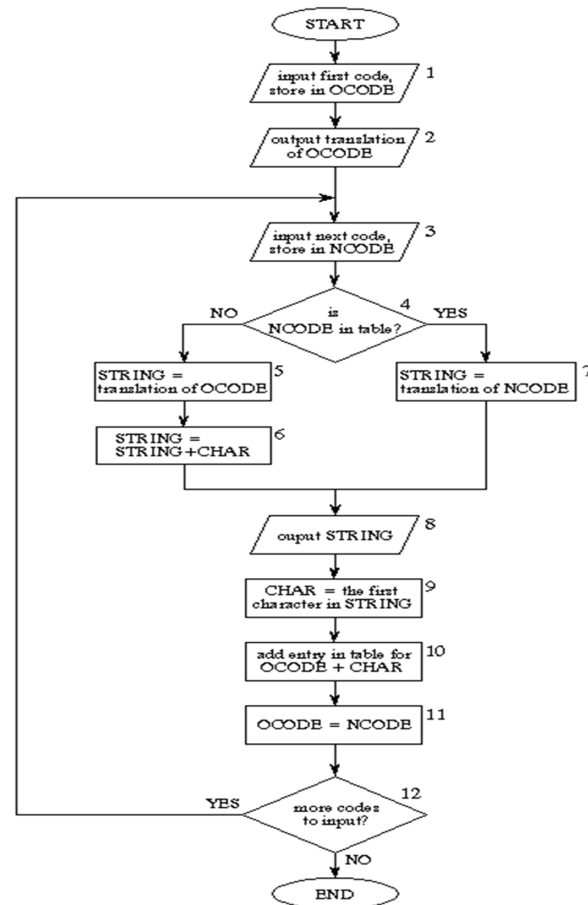
Last string = a a b

output: a a a b a b b a a a a a b a b

76

LZW (5)

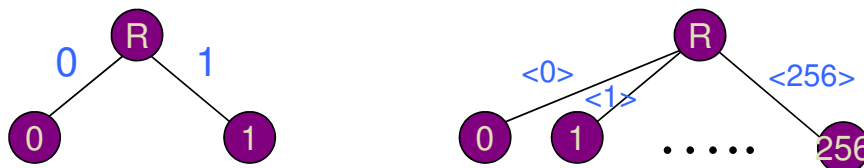
- It seems the decoder is very tricky.
- But the algorithm is very simple again.



77

Implementing LZW

- Instead of handling bit stream, real applications usually handle byte stream, i.e. there are 256 alphabets in the source. $S = \{<0>, \dots, a, b, \dots, z, A, B, \dots, Z, 0, 1, \dots, 9\}$
- Implication: binary tree -> 256-ary tree



- But the algorithm is exactly the same.
- 12 bits are used to represent a codeword => 4096-entry table is used to store the 256-ary tree.
- We start with a 4k dictionary, of which entries 0-255 refer to individual bytes, and entries 256-4095 refer to substrings.

78



LZW: Pros and Cons

- LZW (advantages)
 - Effective at exploiting:
 - character repetition redundancy
 - high-usage pattern redundancy
 - ** but not on positional redundancy
 - An adaptive algorithm
 - One-pass procedure
- LZW (disadvantages)
 - Message must be sufficient long for effective compression