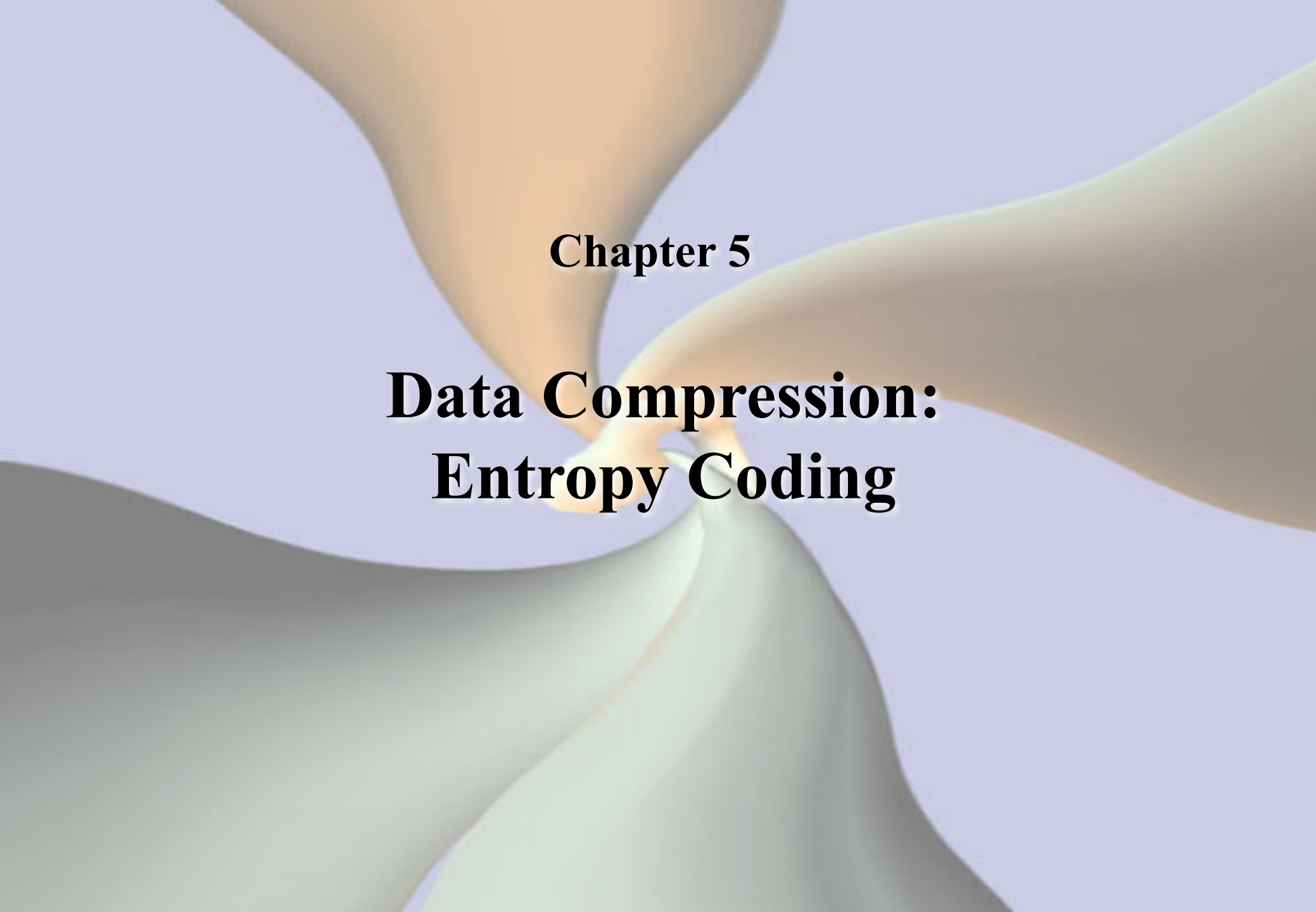


Chapter 5

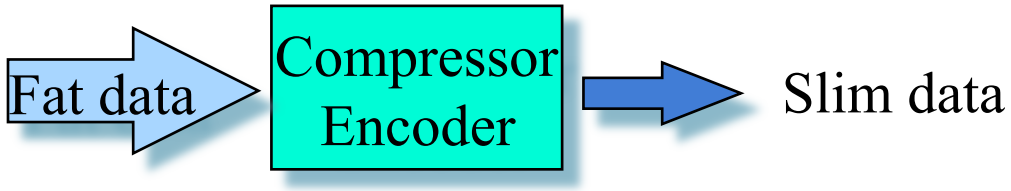
Data Compression: Entropy Coding

Storage Size for Media Types

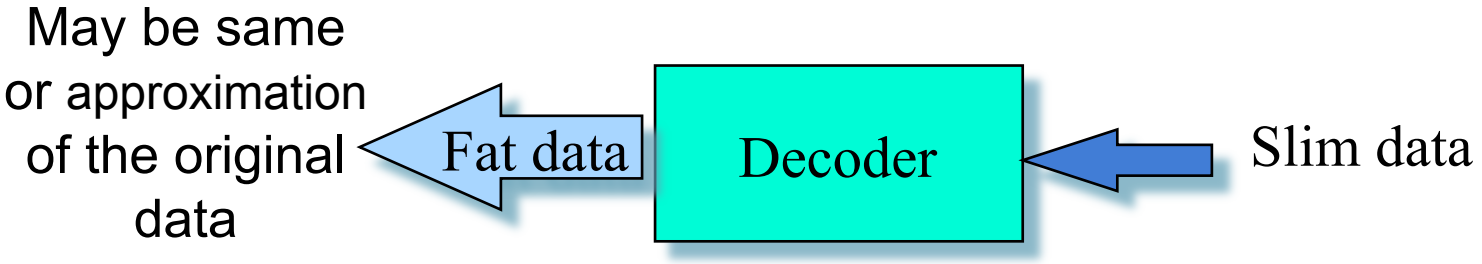
	Text	Image	Audio	Video
Object Type	-ASCII -EBCDIC -BIG5 -GB	-Bitmapped graphics -Still photos -Faxes	-Noncoded stream of digitized audio or voice	-Digital video at 25-30 frames/s
Size and Bandwidth	2KB per page	-Simple: 1MB per image -Detailed (color) 7.5 MB per image	Voice/Phone 8kHz/8 bits (mono) 8-44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



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)

Redundancy (2)

- ***Character Distribution***
  - Some characters are used more frequently than others.
  - In English text, ‘e’ and *space* occur most often.
  - Example: “abacabacabacabacabacabacabac”
  - 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 => “--..” codeword
    - Morse’s principle: assign short code strings to frequently occurring letters and longer code to letters that occur less often.

Redundancy (3)

- ***Character Repetition***
  - When a string of repetitions of a single character occurs, e.g., “aaaaaaaaaaaaaabc”
  - 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. $\theta = "th"$

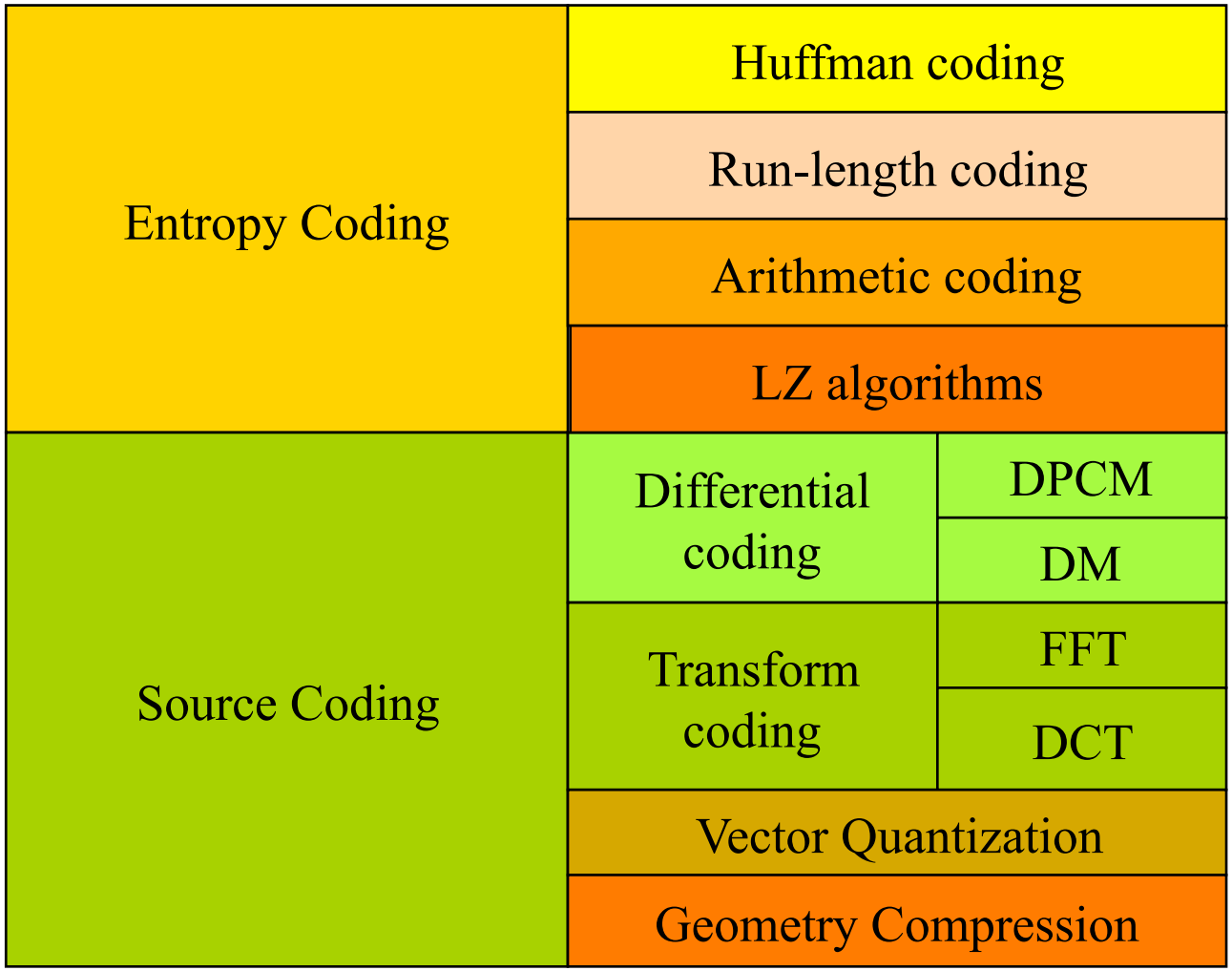
Redundancy (4)

- ***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.

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.

Categorization



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.

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.

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!!

delimiter

某一個codeword是另一codeword的prefix

Prefix Code 無 prefix condition

- 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

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.

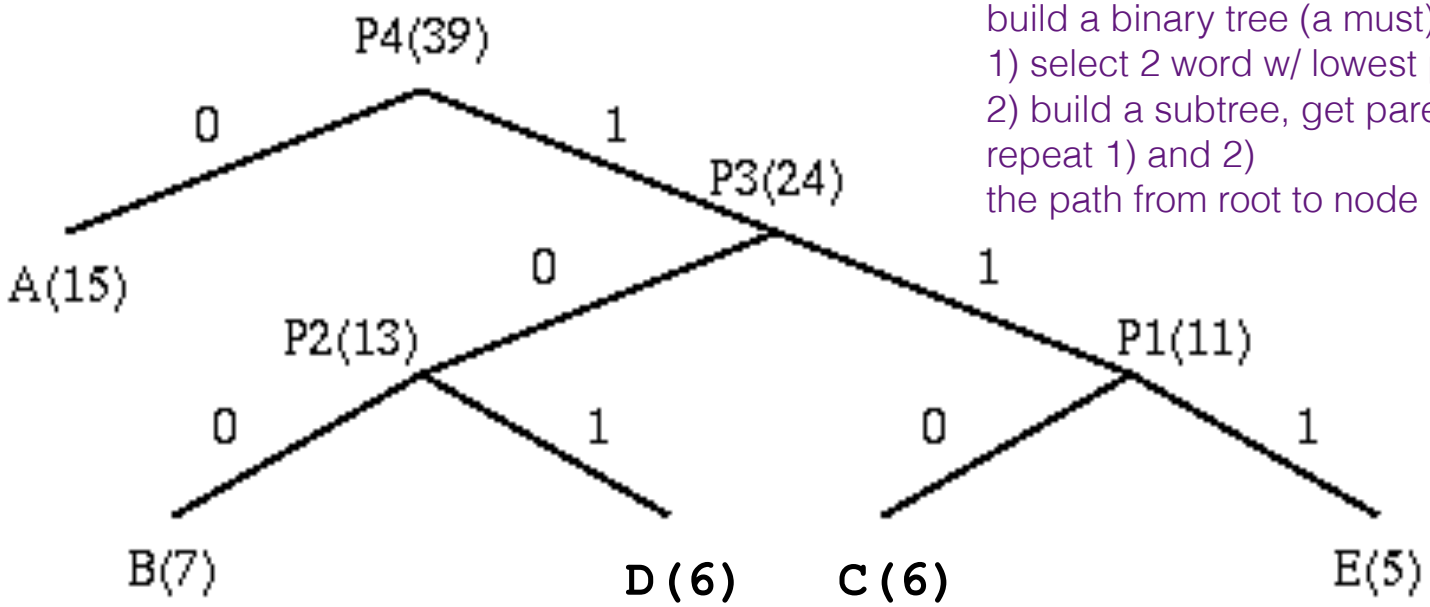
Huffman Code (2)

all char must be leave node
=> so no prefix conflict

愈唔balance
performance愈好

- 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

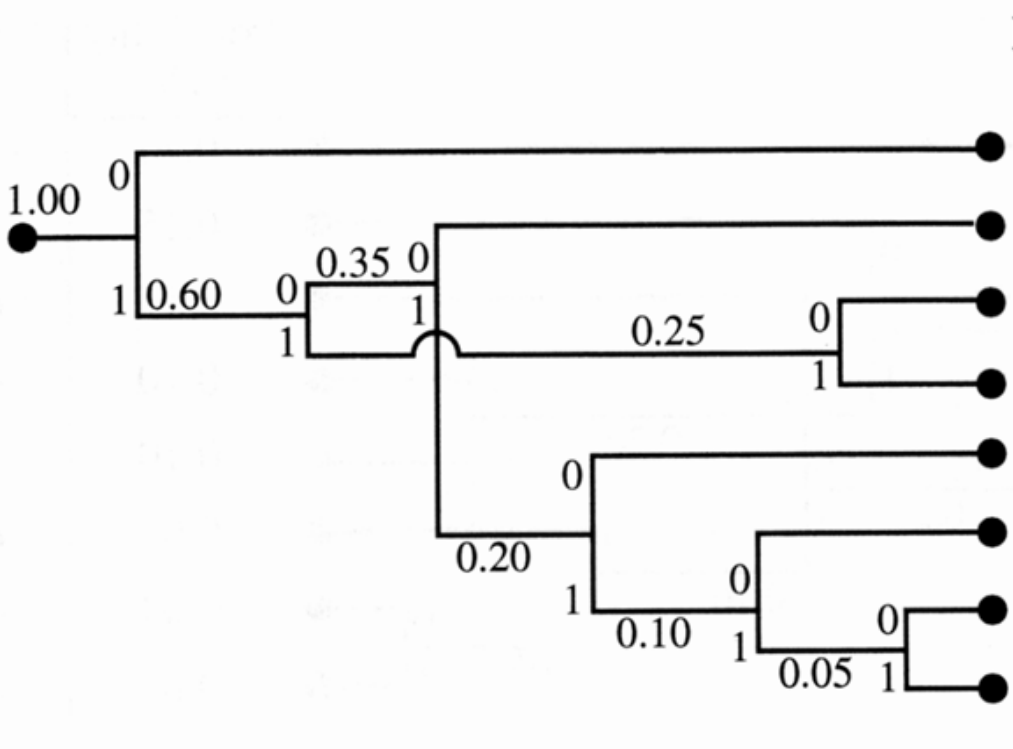


build a binary tree (a must)
1) select 2 word w/ lowest propability
2) build a subtree, get parent P1
repeat 1) and 2)
the path from root to node is the codeword

- Code assignment may not be unique.

Huffman Code (3)

- More example 2:



Probability P_j	Letter a_j	Codeword x_j
0.40	a_1	0
0.15	a_2	100
0.15	a_3	110
0.10	a_4	111
0.10	a_5	1010
0.05	a_6	10110
0.04	a_7	101110
0.01	a_8	101111

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: $\bar{l} = 0.40*1 + (0.15+0.15+0.10)*3 + 0.10*4 + 0.05*5$

  $+ (0.04+0.01)*6 = 2.55$

  要one-by-one encode => at least 1 char 1 codeword
  cannot more than 1 char to give out 1 codeword

  Optimal for instantaneous coding (code without buffering)

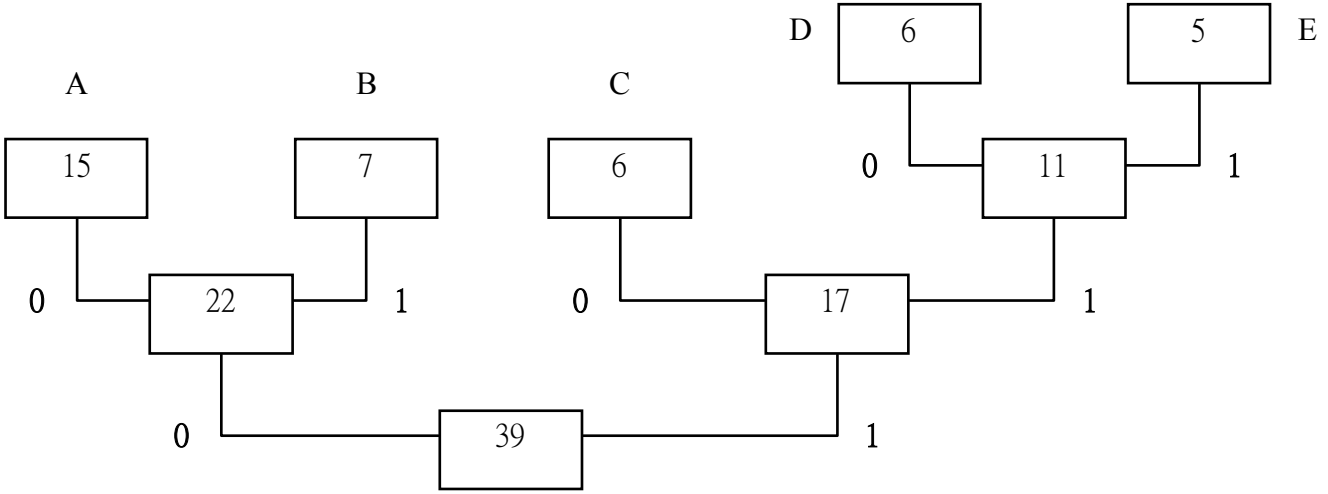
Huffman Code (Disadvantages)

- 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:

sum of probabilities



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.

How small we can compress?

信息量

- 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.

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

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

$$\bar{l} - H = \sum_j p_j (l_j + \log_2 p_j)$$

$$= \sum_j p_j \log_2 (2^{l_j} p_j) \geq \log_2 e \sum_j p_j (1 - \frac{2^{-l_j}}{p_j})$$

$$= \log_2 e \sum_j (p_j - 2^{-l_j})$$

$$= \log_2 e (1 - \sum_j 2^{-l_j}) \geq \log_2 e (1-1) = 0$$

Mathematical fact:

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

Kraft inequality:

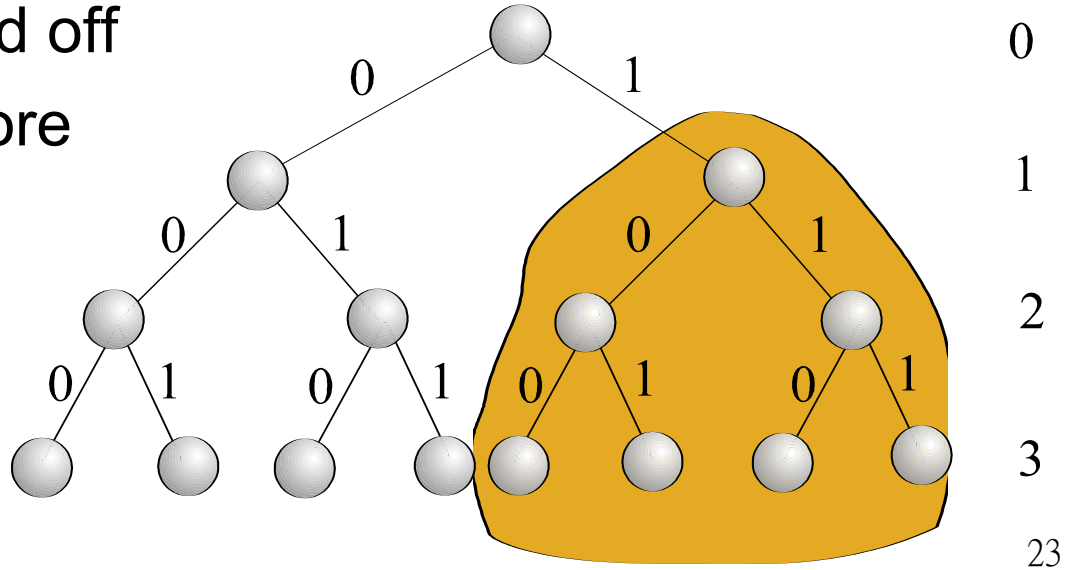
$$\sum_j 2^{-l_j} \leq 1$$

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

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

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



Entropy

- Q: How about an image in which half of the pixels are white, another half are black?
  - Ans: p(white) = 0.5, p(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, $\bar{l} \geq H$

Arithmetic Coding

- Huffman coding uses integer number (k) of bits for each symbol.
- k is at least 1.
- Sometimes groups of symbols repeat frequently, but Huffman code cannot utilize this redundancy.
- Arithmetic coding works by representing a sequence of symbols by an interval of real numbers between 0 and 1.
- As the sequence becomes longer, the interval needed to represent it becomes smaller, and the number of bits needed to represent the interval increases.
- The interval is like this [L, R) where the left bound L is inclusive and the right bound R is exclusive.
- Find a real number within the interval to represent that interval. What value? The most convenient solution: L

Arithmetic Coding (2)

Algorithm:

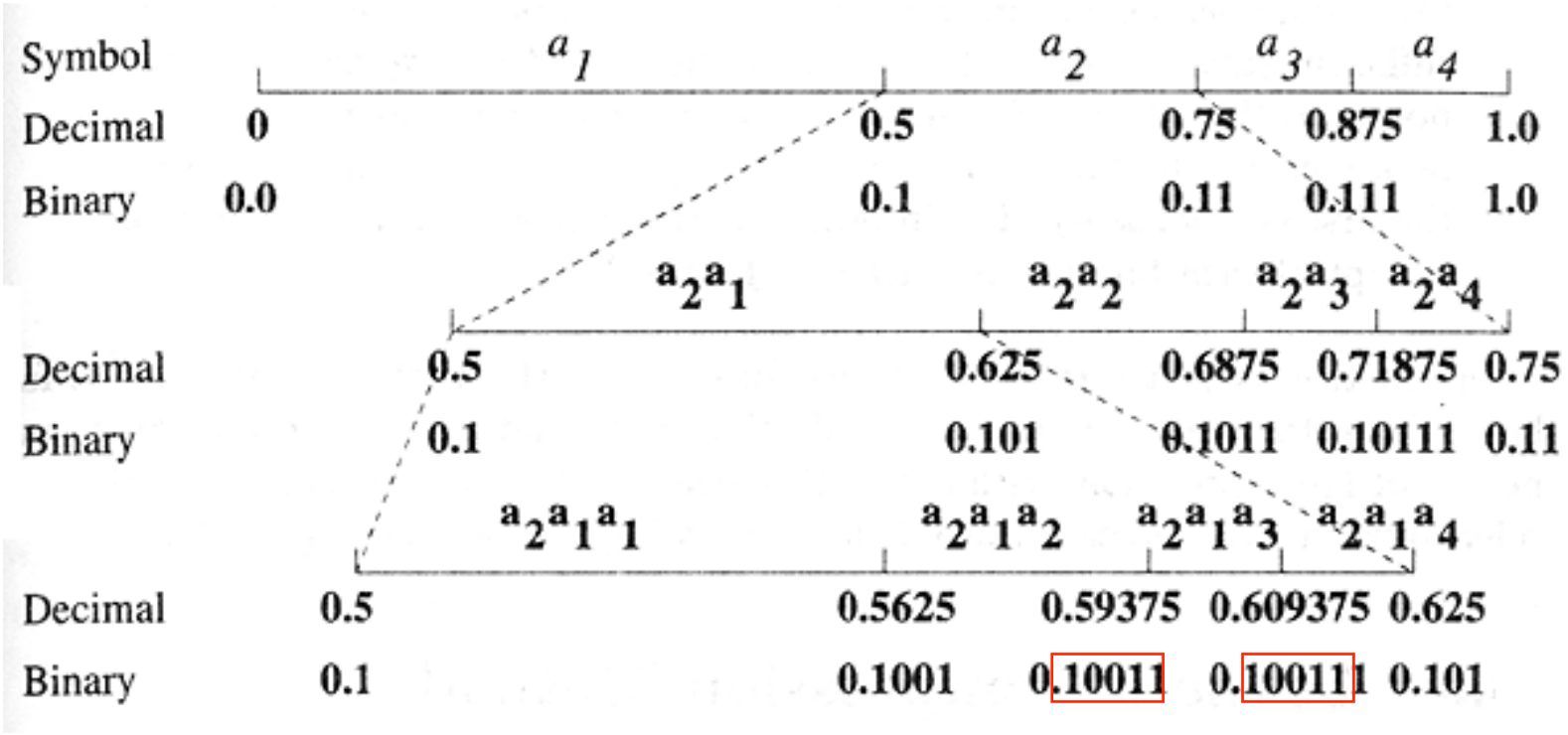
- Suppose the alphabet contains $\{a_i\}$, i=1, …, M, and probabilities $p(a_i) = p_i$
- Each symbol is assigned with an interval in [0,1]
- Read the first symbol, let L and R be the lower and upper bound the interval representing this symbol
- Repeat
  - Express L and R in binary representation, e.g.
    L = .01101
    R= .010111
  - output their common binary prefix that are not previously output, e.g. 01 (in previous example)
  - Read the next symbol.
  - Update L and R to the next subinterval.
- Until no more symbol
- Output the rest of the binary bits of L

Arithmetic Coding (3)

Example:

Encode $a_2\, a_1\, a_3$

$p_1 = 0.5,\quad p_2 = 0.25,\ p_3 = 0.125,\ p_4 = 0.125$



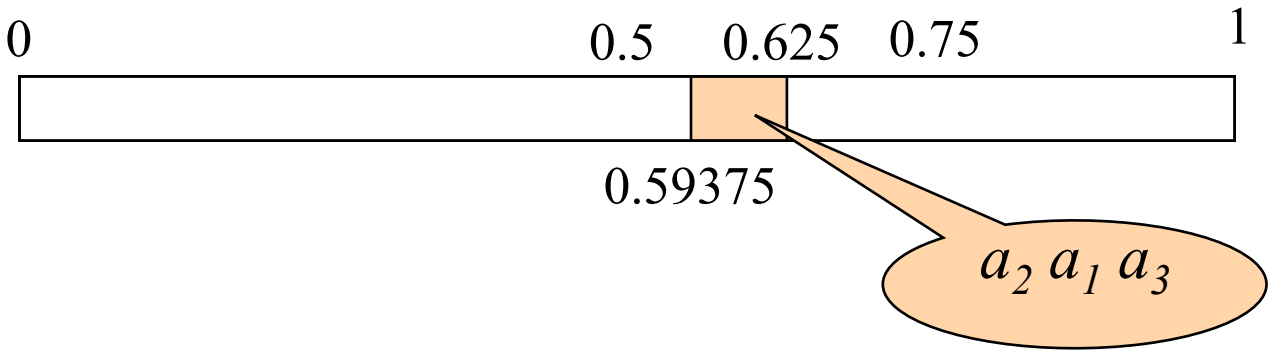
Output: 1 0 011

Arithmetic Coding (4)

At decoder side:

Received Bit	Interval	Output
1	[0.5, 1)	-
0	[0.5, 0.75)	a_2
0	[0.5, 0.625)	a_1
1	[0.5625, 0.625)	-
1	[0.59375, 0.625)	a_3

"End of bitstream"



Arithmetic Coding (5)

- Number of bits of codeword determined by the size of interval.
- Less-occurred sequence is assigned a longer codeword as its corresponding interval is tiny.
- The code length approaches optimal encoding as the sequence gets longer.
- Used in JBIG standard (a fax coding)

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.

Lempel-Ziv Algorithms

- Proposed by A. **L**empel and J. **Z**iv.
- 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

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. 做個索引 ID
- 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.

LZ78

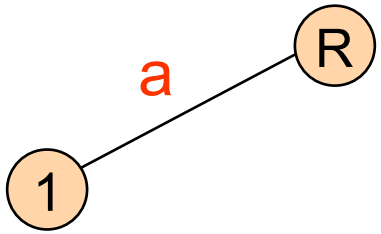
- Let's consider an input source with only two symbols "a" and "b". S = {a, b}
- The basic idea is to construct a LZ78 tree. ("dictionary")
- Since there are only two alphabets, the LZ78 tree is binary in this case.
- The tree starts as a single root node "R". It grows as more input data enter.
- Let's further assume the input data stream is
  a a a b a b b a a a a a b a b a 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!

LZ78 (2)

R

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

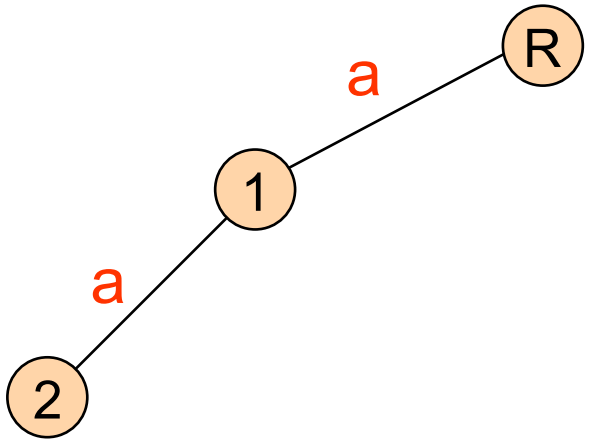
LZ78 (2)



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

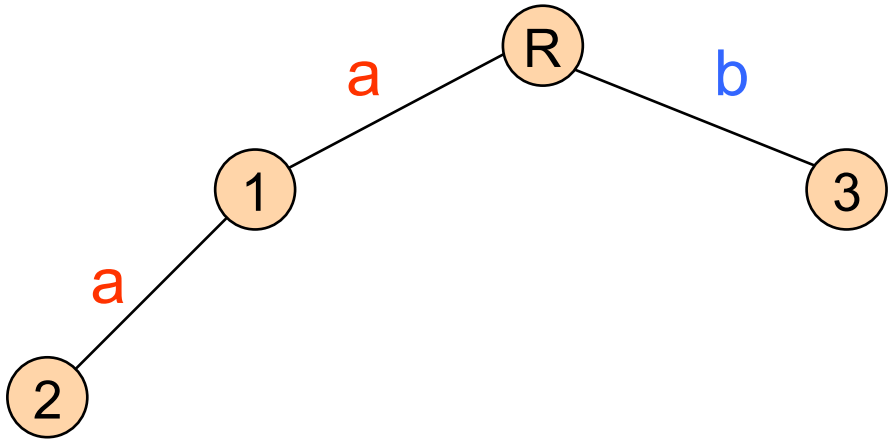
LZ78 (2)



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,

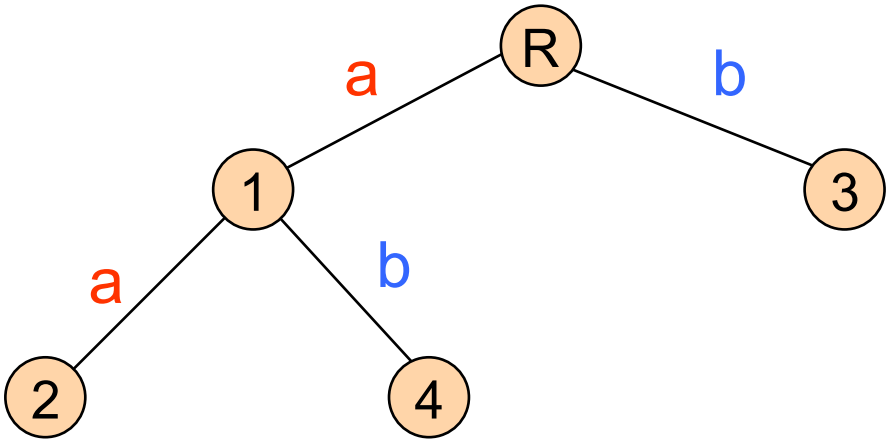
LZ78 (2)



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,

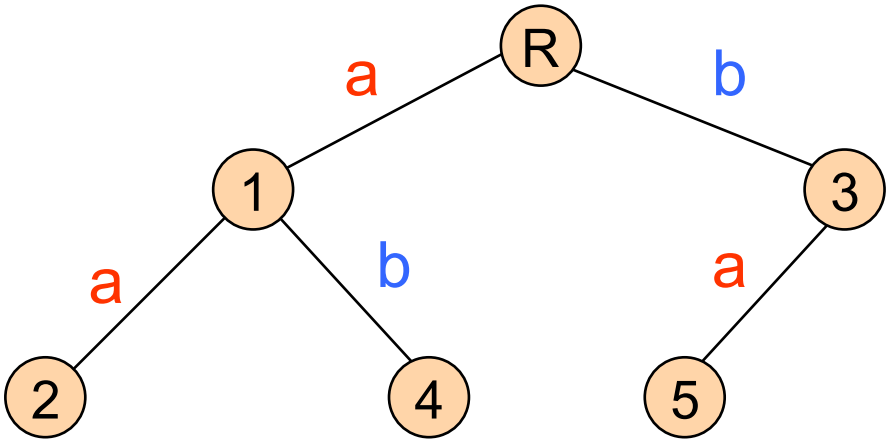
LZ78 (2)



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,

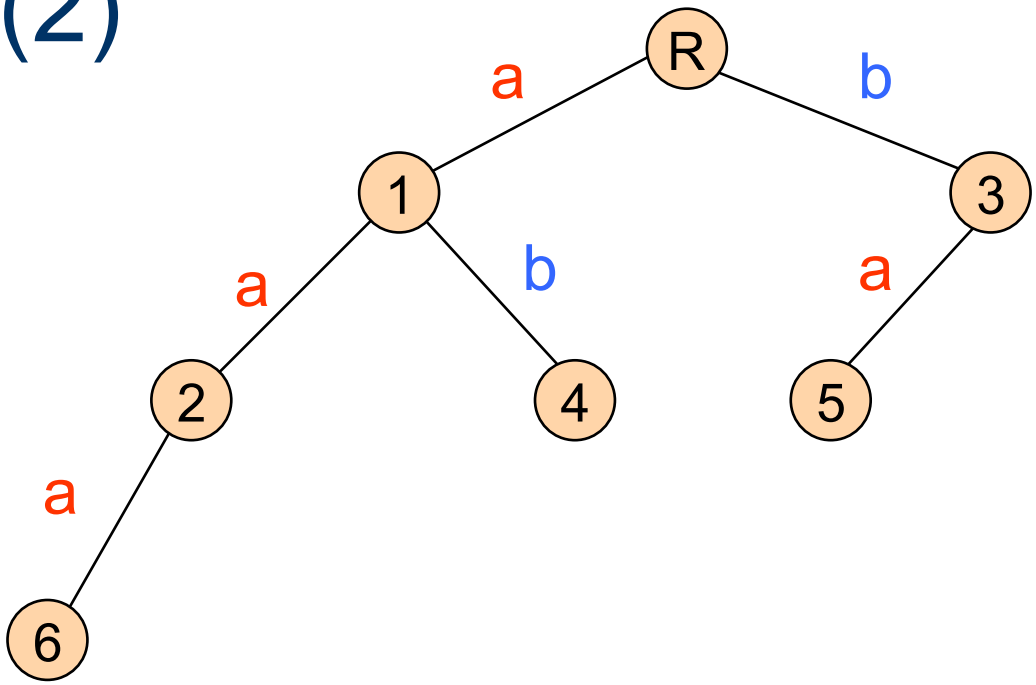
LZ78 (2)



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,

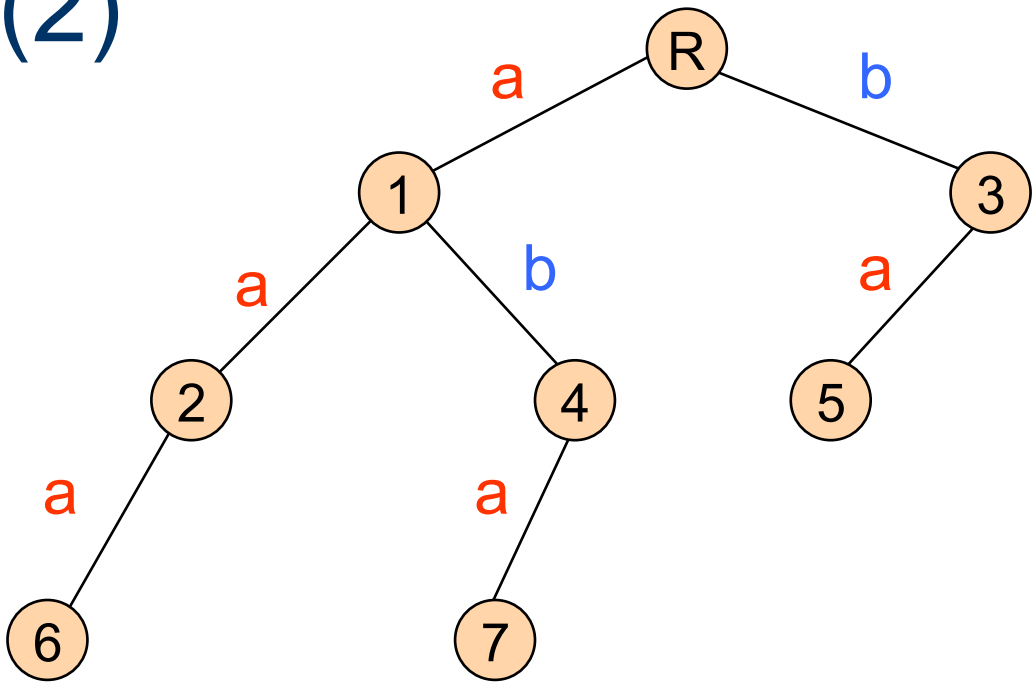
LZ78 (2)



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,

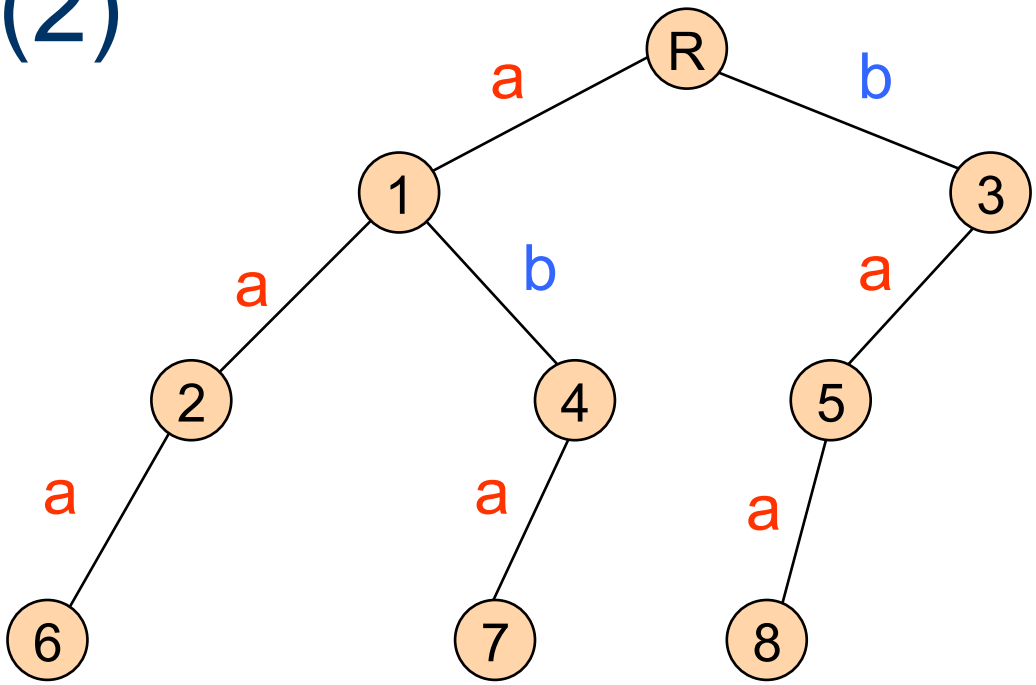
LZ78 (2)



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,

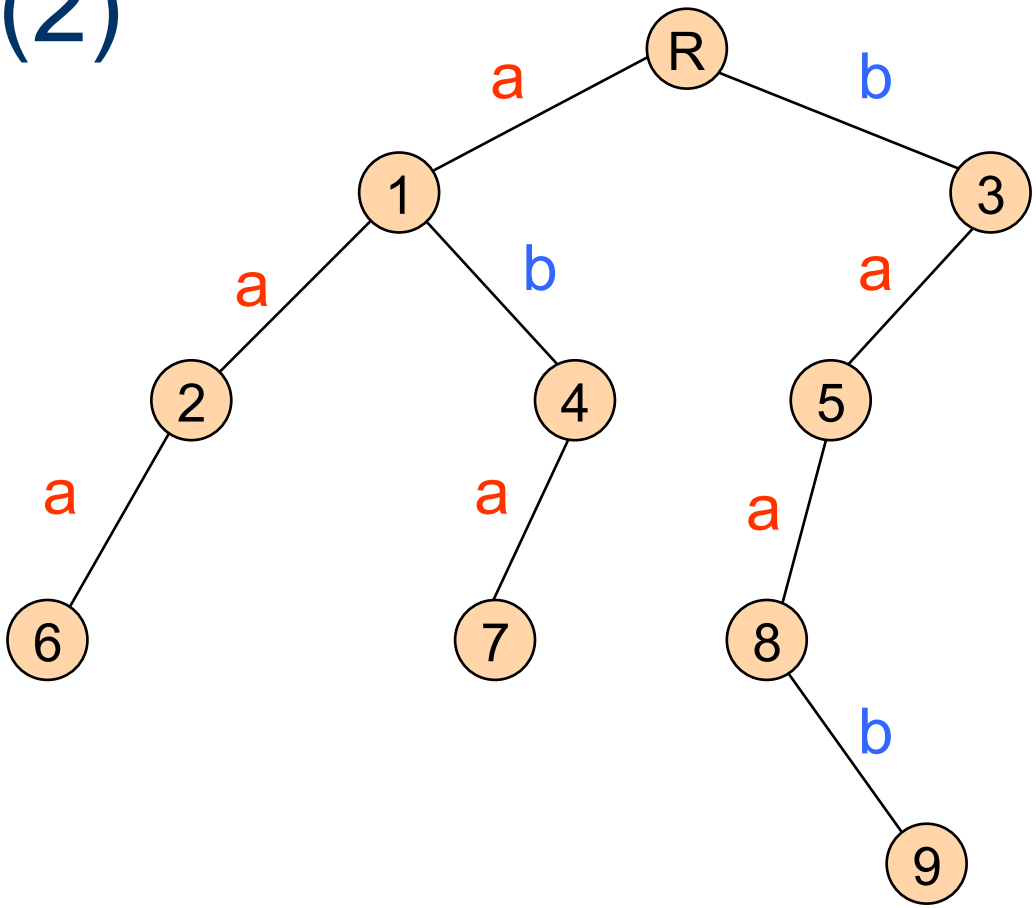
LZ78 (2)



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,

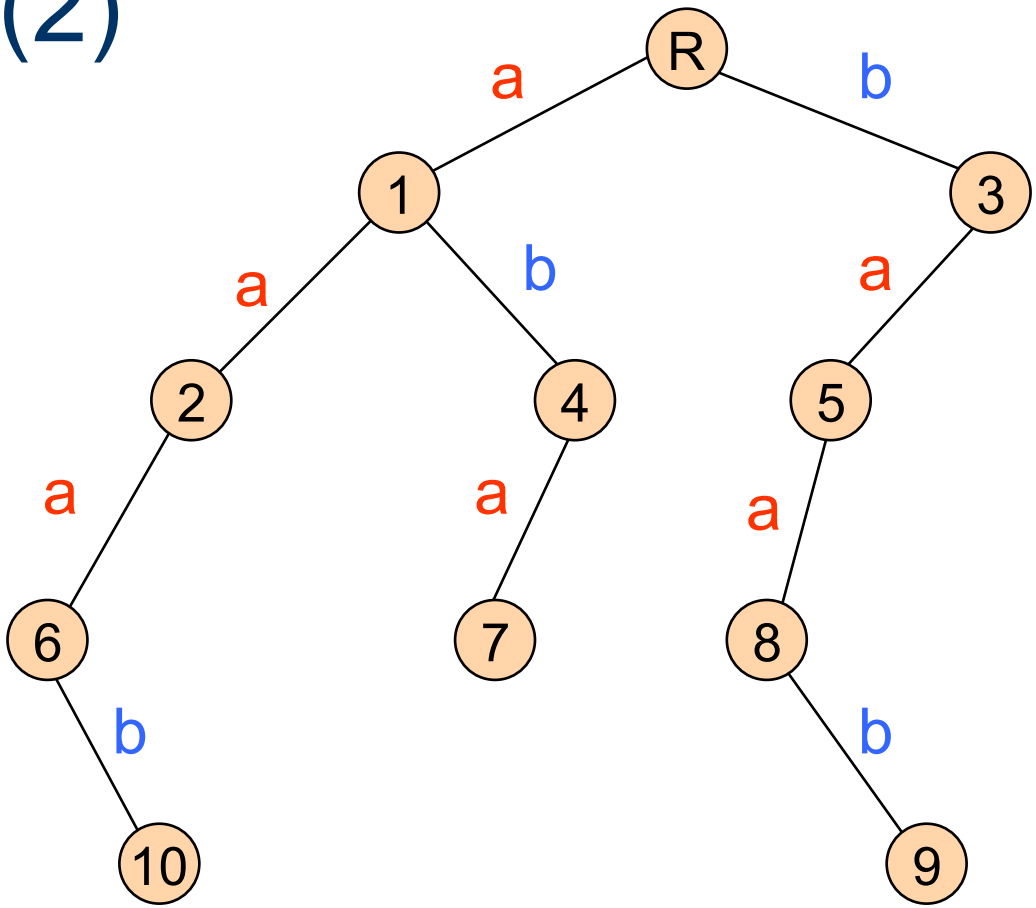
LZ78 (2)



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,

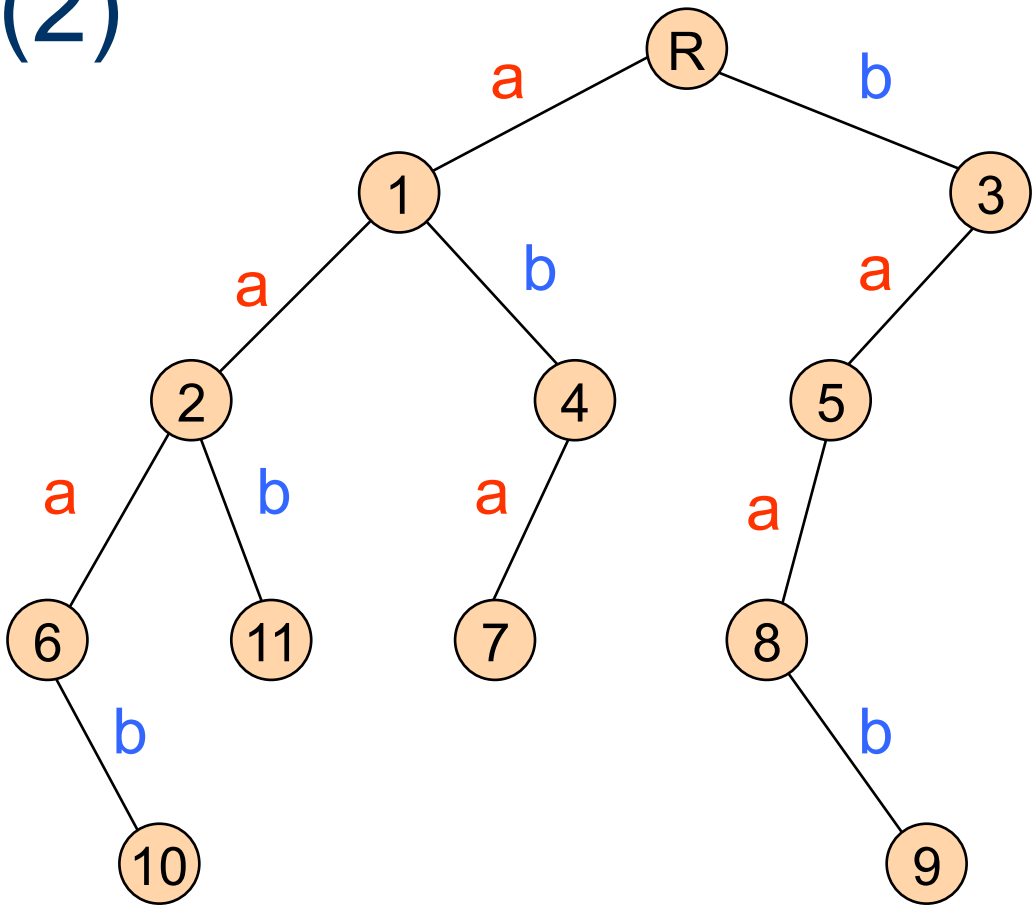
LZ78 (2)



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,

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,

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

LZ78 (4)

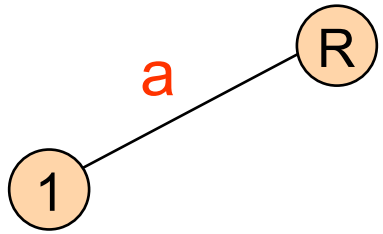
DECODER SIDE:

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

output:

LZ78 (4)

DECODER SIDE:

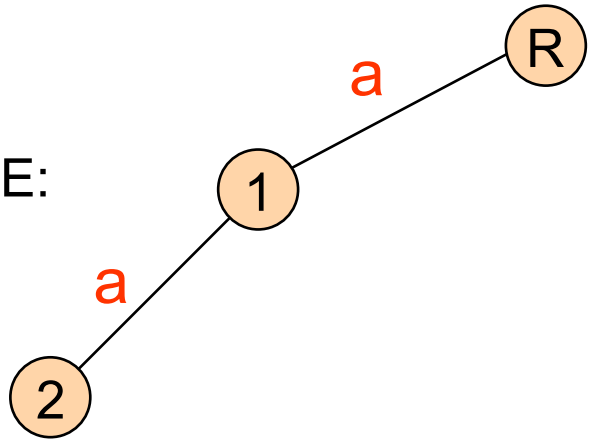


input: R, a, 1, a, R, b, 1, b, 3, a, 2, a, 4, a, 5, a, 8, b, 6, b, 2, b,

output: a

LZ78 (4)

DECODER SIDE:

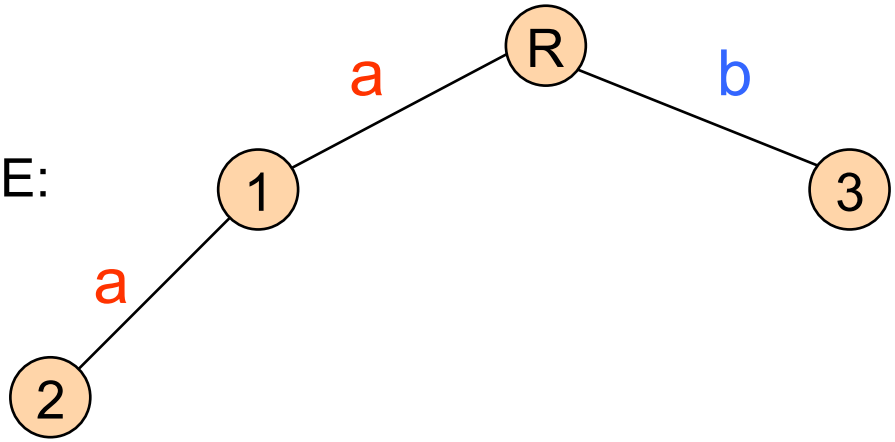


input: R, a, 1, a, R, b, 1, b, 3, a, 2, a, 4, a, 5, a, 8, b, 6, b, 2, b,

output: a a a

LZ78 (4)

DECODER SIDE:

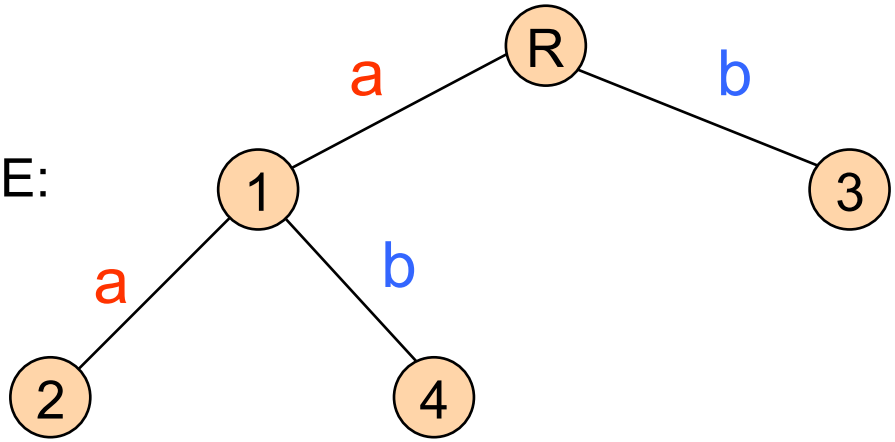


input: R, a, 1, a, R, b, 1, b, 3, a, 2, a, 4, a, 5, a, 8, b, 6, b, 2, b,

output: a a a b

LZ78 (4)

DECODER SIDE:

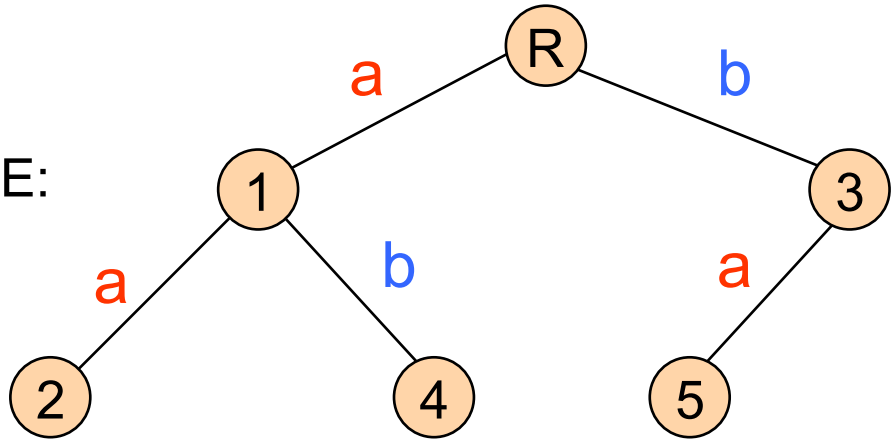


input: R, a, 1, a, R, b, 1, b, 3, a, 2, a, 4, a, 5, a, 8, b, 6, b, 2, b,

output: a a a b a b

LZ78 (4)

DECODER SIDE:

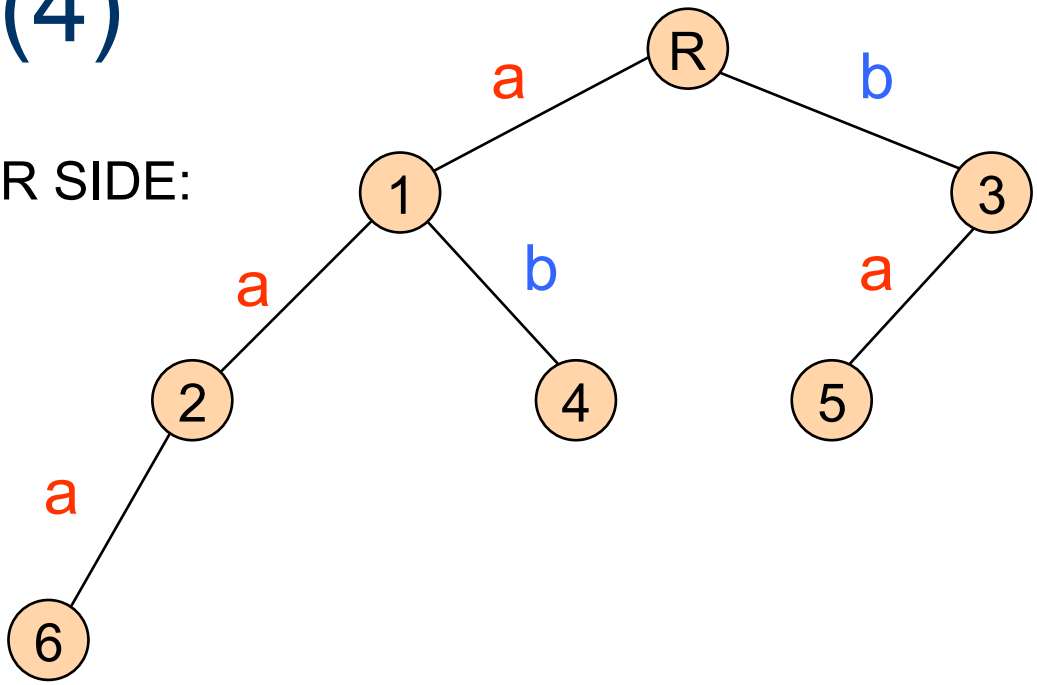


input: R, a, 1, a, R, b, 1, b, 3, a, 2, a, 4, a, 5, a, 8, b, 6, b, 2, b,

output: a a a b a b b a

LZ78 (4)

DECODER SIDE:

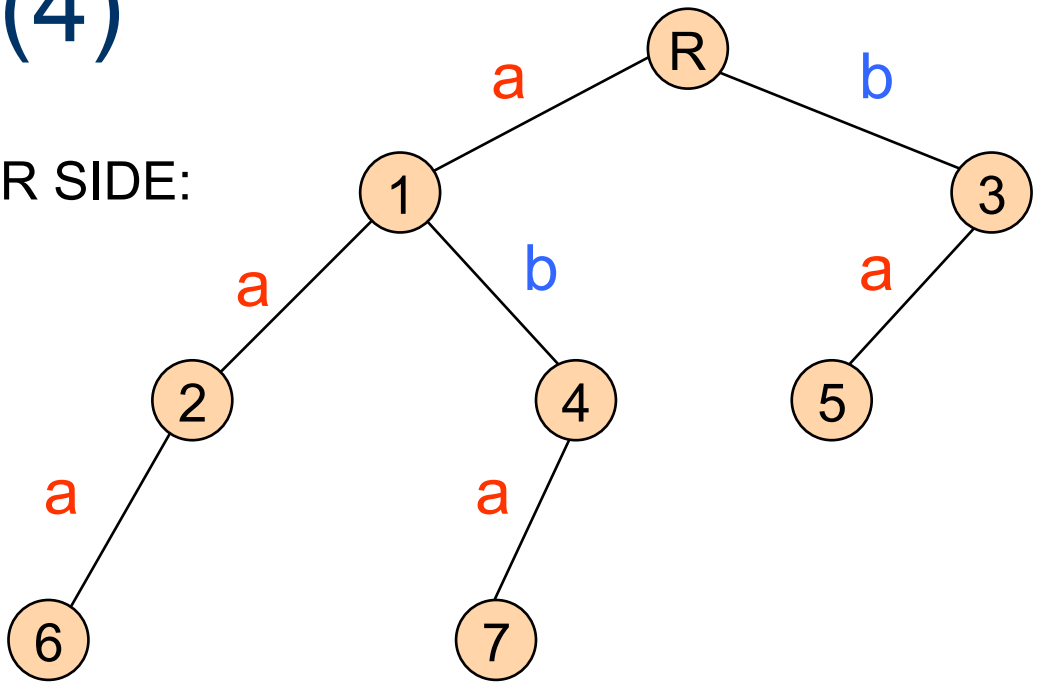


input: R, a, 1, a, R, b, 1, b, 3, a, 2, a, 4, a, 5, a, 8, b, 6, b, 2, b,

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

LZ78 (4)

DECODER SIDE:

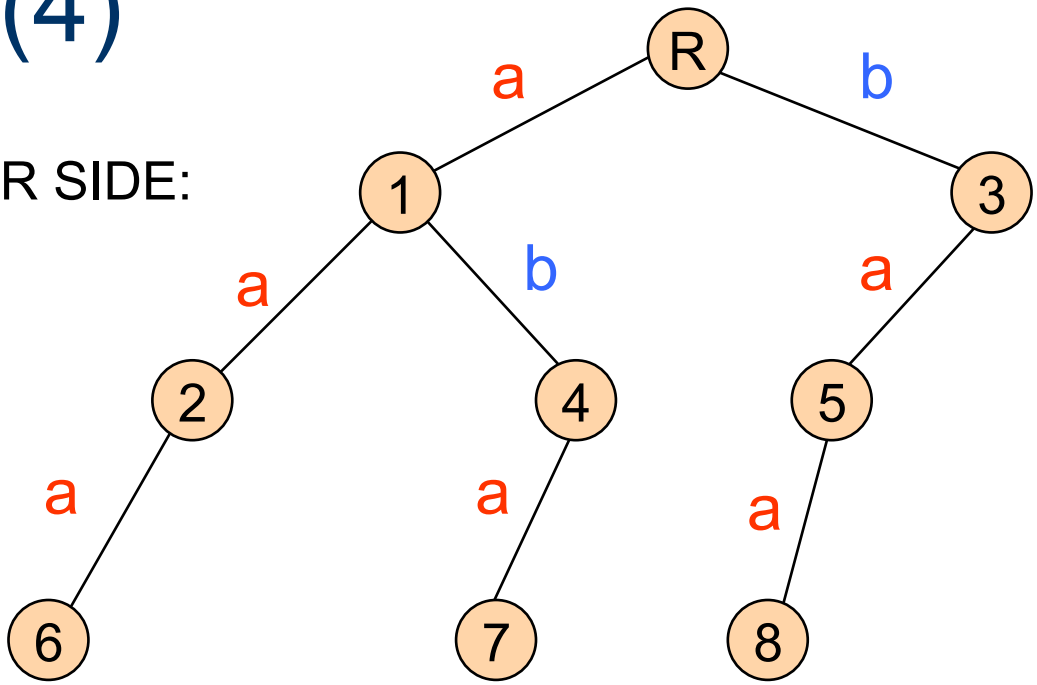


input: R, a, 1, a, R, b, 1, b, 3, a, 2, a, 4, a, 5, a, 8, b, 6, b, 2, b,

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

LZ78 (4)

DECODER SIDE:

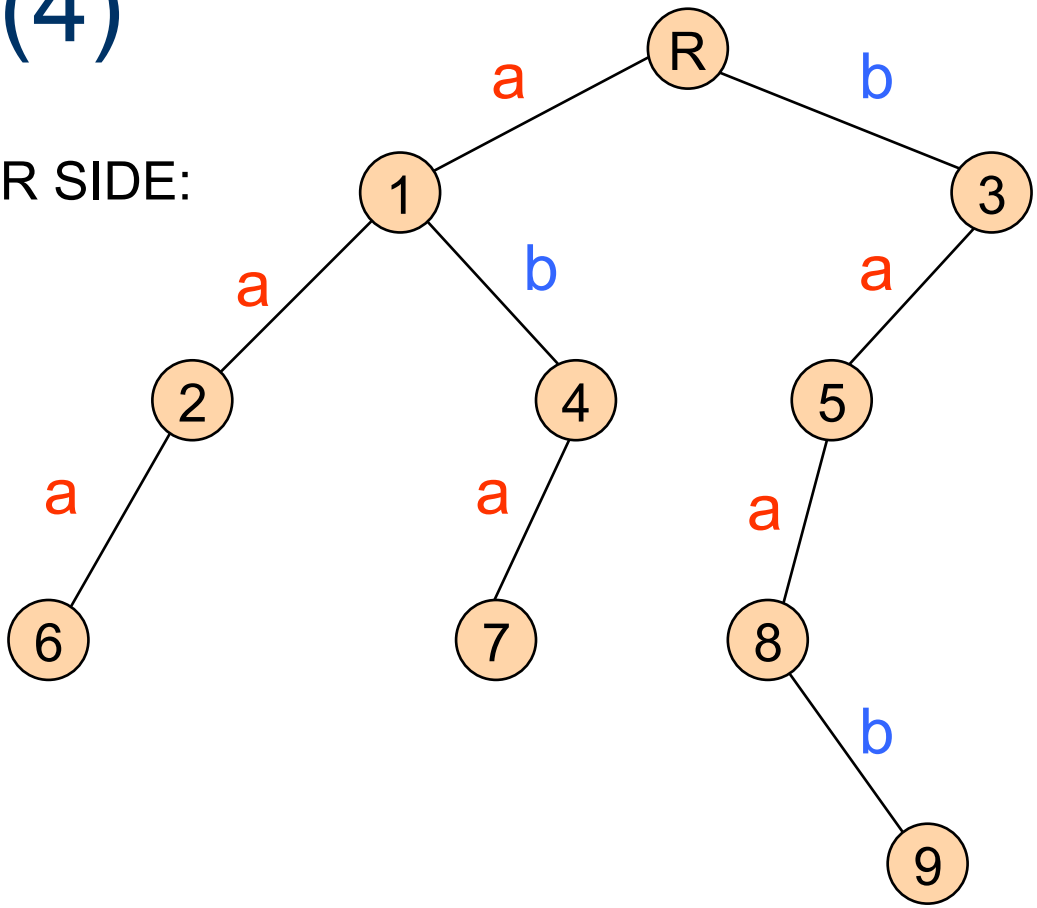


input: R, a, 1, a, R, b, 1, b, 3, a, 2, a, 4, a, 5, a, 8, b, 6, b, 2, b,

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

LZ78 (4)

DECODER SIDE:

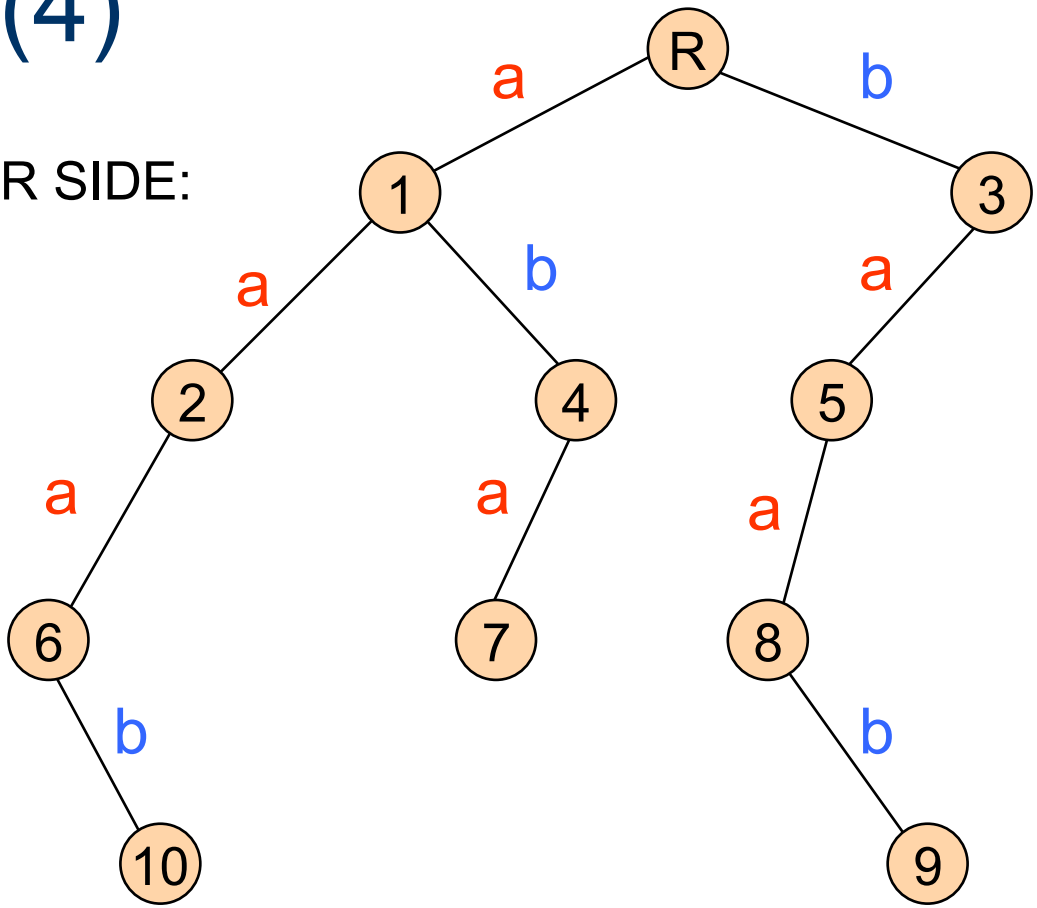


input: R, a, 1, a, R, b, 1, b, 3, a, 2, a, 4, a, 5, a, 8, b, 6, b, 2, b,

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

LZ78 (4)

DECODER SIDE:

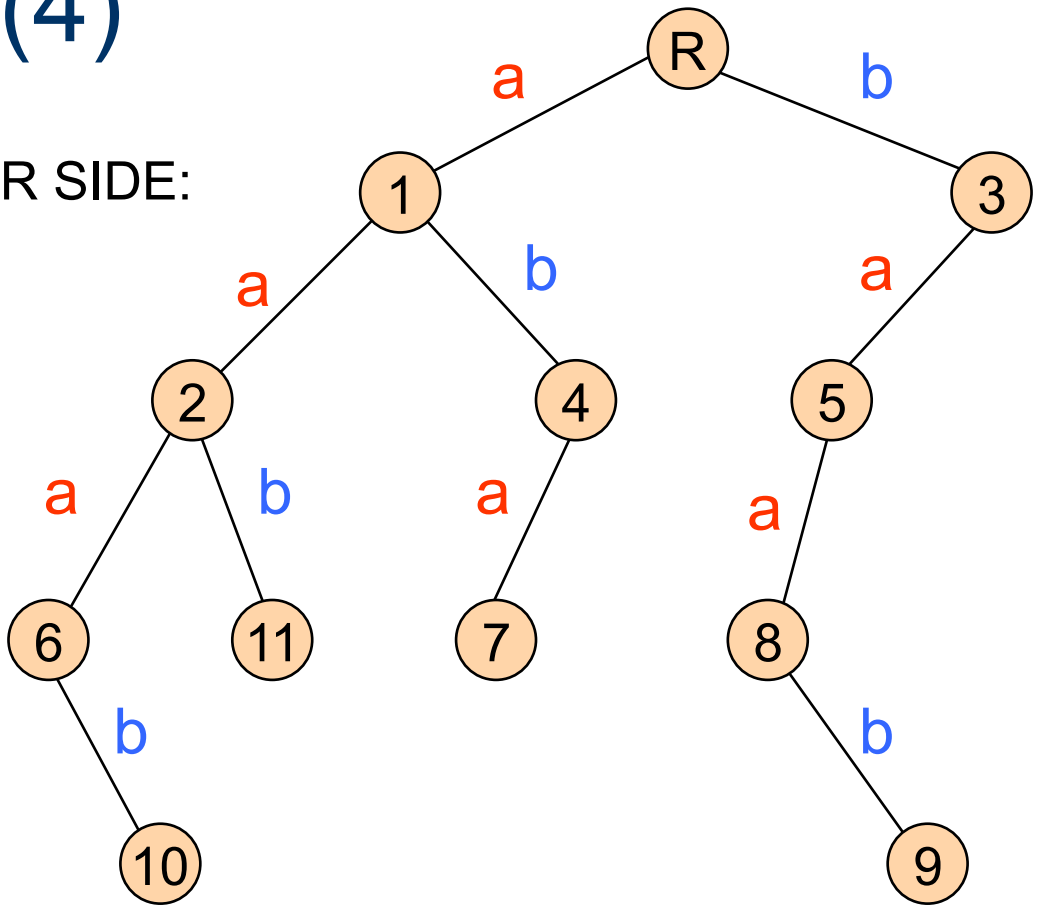


input: R, a, 1, a, R, b, 1, b, 3, a, 2, a, 4, a, 5, a, 8, b, 6, b, 2, b,

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

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,

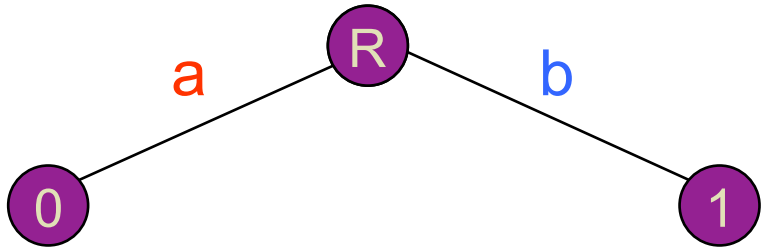
output: a a a b a b b a a a a a b a b a a b a a b a a a b a a b

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

LZW (2)

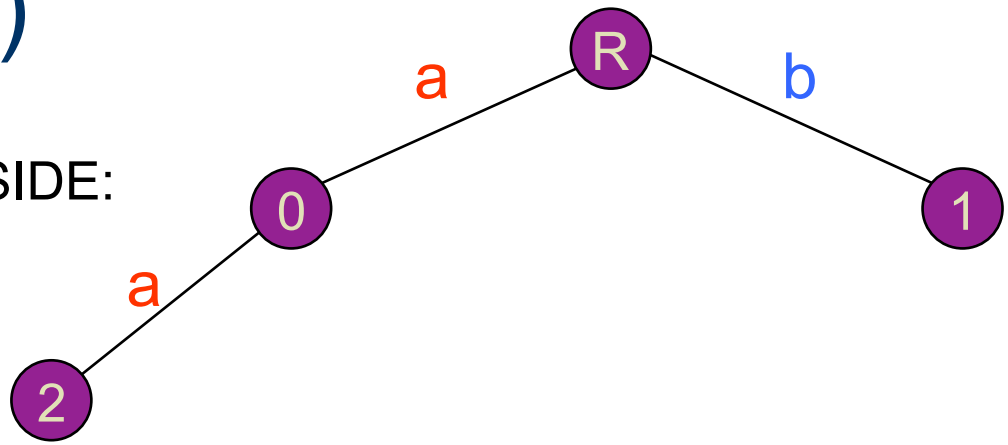
ENCODER SIDE:



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

LZW (2)

ENCODER SIDE:

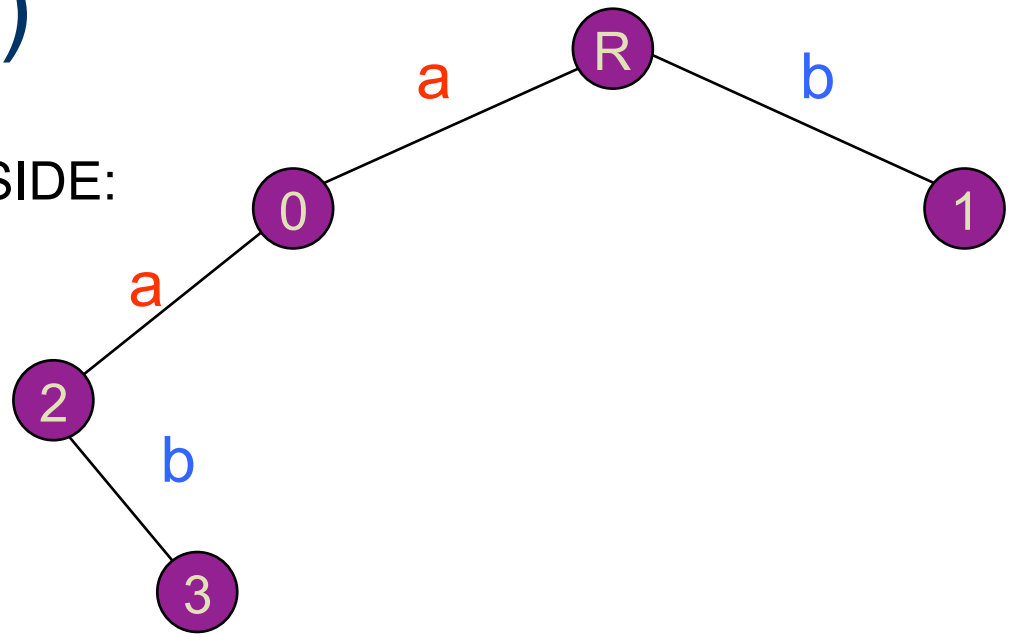


a a a b a b b a a a a a b a b a a b a a b a a a b a a b

output: 0,

LZW (2)

ENCODER SIDE:

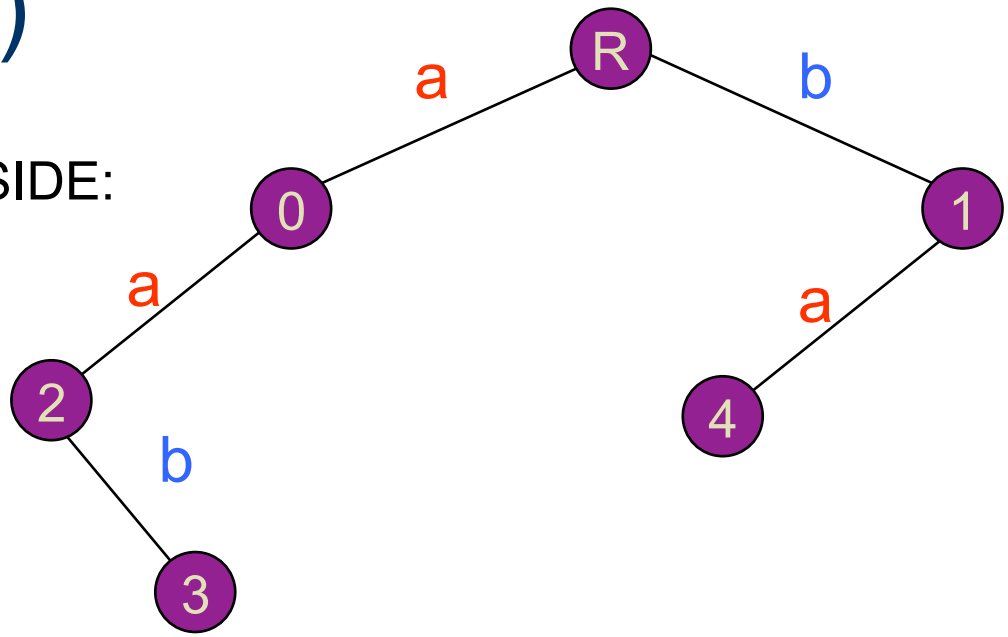


a a a b a b b a a a a a b a b a a b a a b a a a b a a b

output: 0, 2,

LZW (2)

ENCODER SIDE:

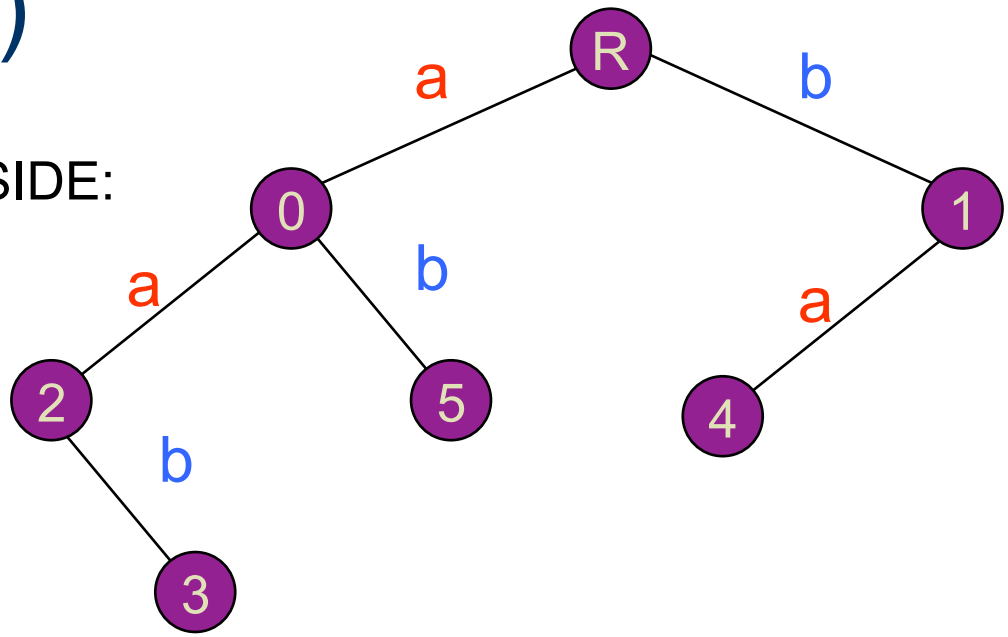


a a a b a b b a a a a a b a b a a b a a b a a a b a a b

output: 0, 2, 1,

LZW (2)

ENCODER SIDE:

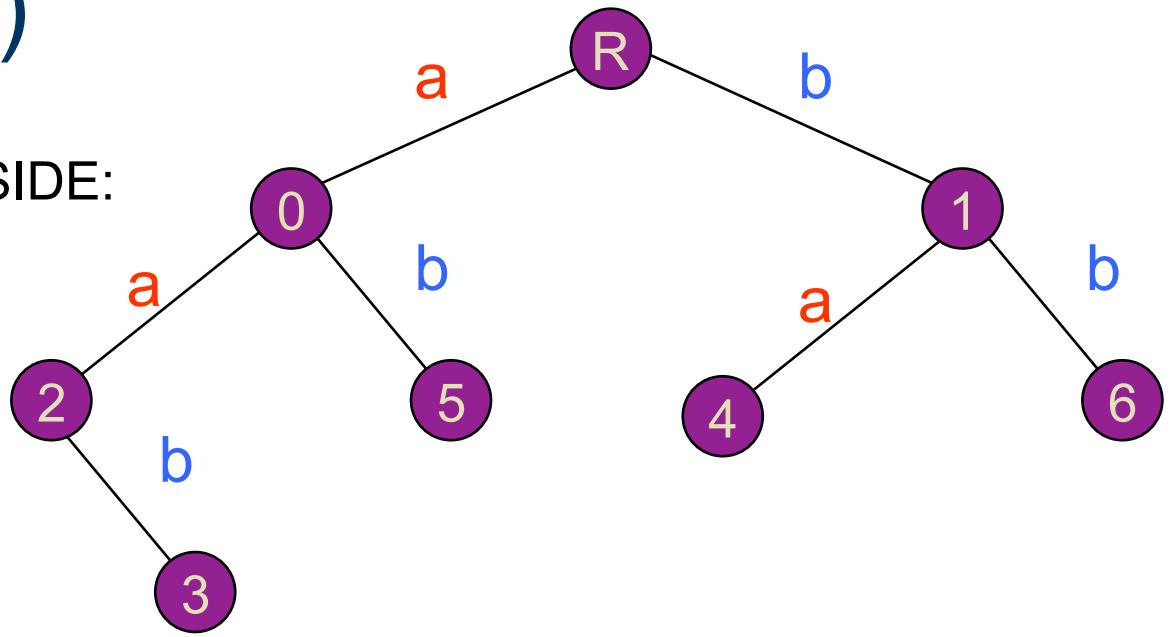


a a a b a b b a a a a a b a b a a b a a b a a a b a a b

output: 0, 2, 1, 0,

LZW (2)

ENCODER SIDE:

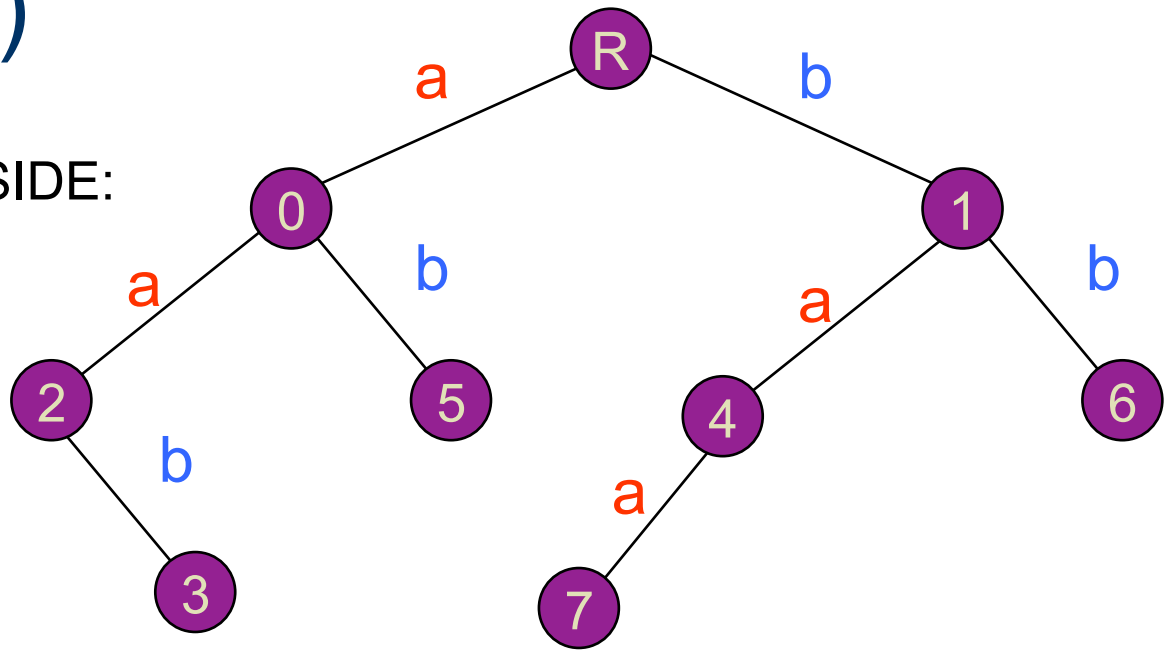


a a a b a b b a a a a a b a b a a b a a b a a a b a a b

output: 0, 2, 1, 0, 1

LZW (2)

ENCODER SIDE:

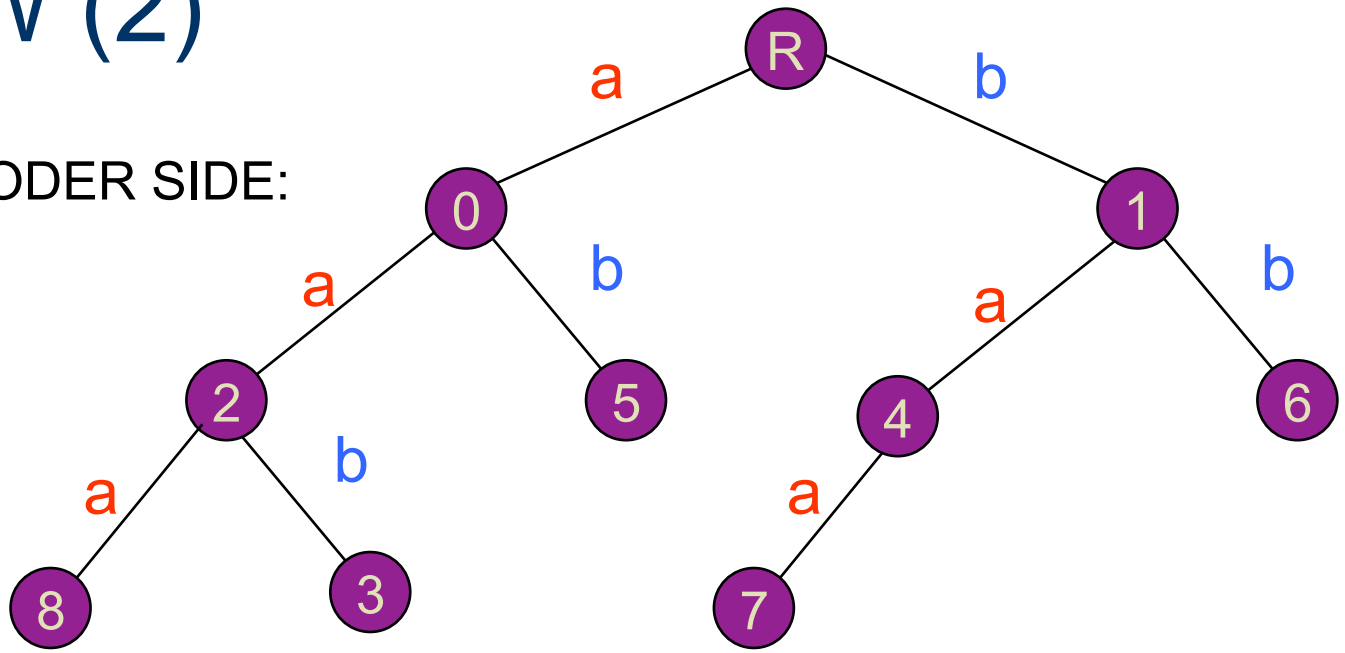


a a a b a b b a a a a a b a b a a b a a b a a a b a a b

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

LZW (2)

ENCODER SIDE:

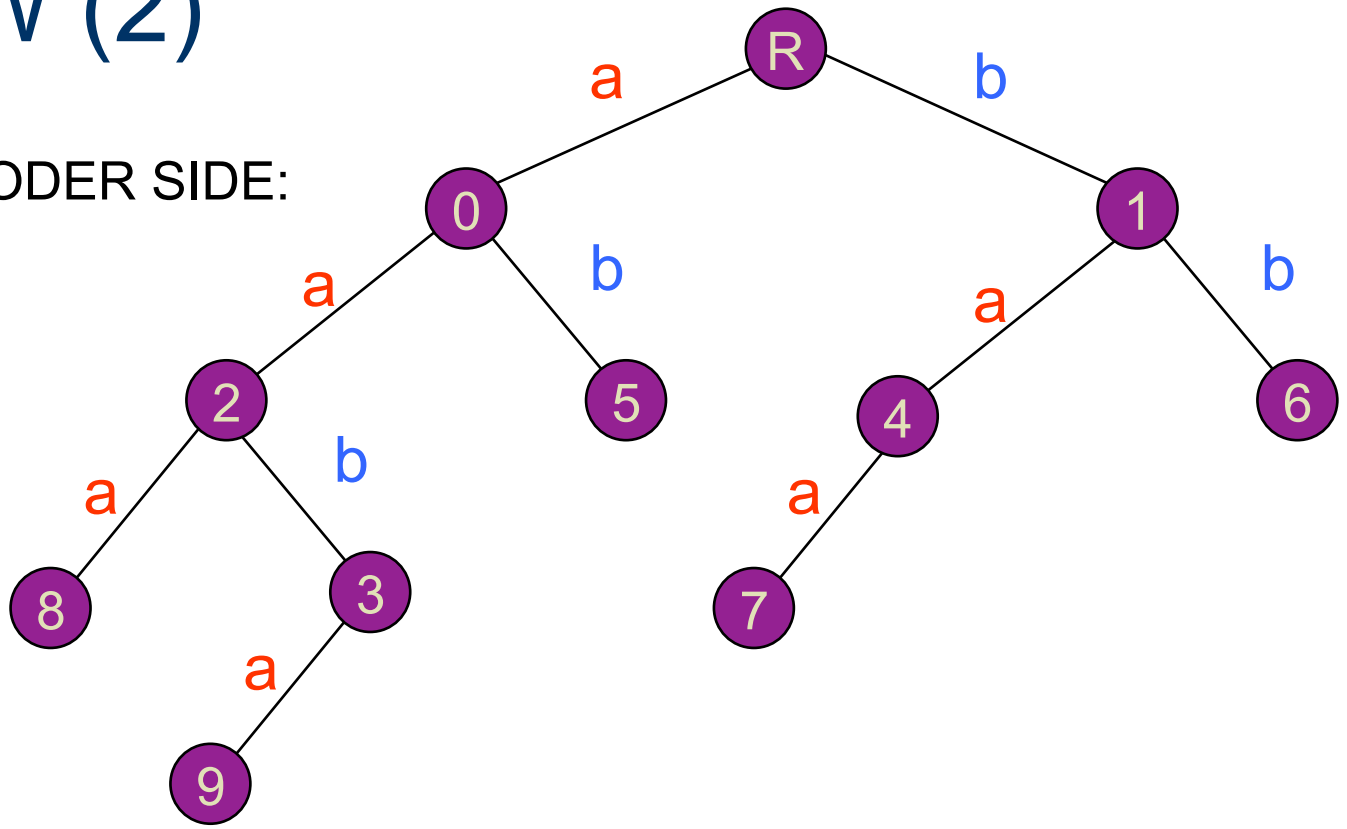


a a a b a b b a a a a a b a b a a b a a b a a a b a a b

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

LZW (2)

ENCODER SIDE:

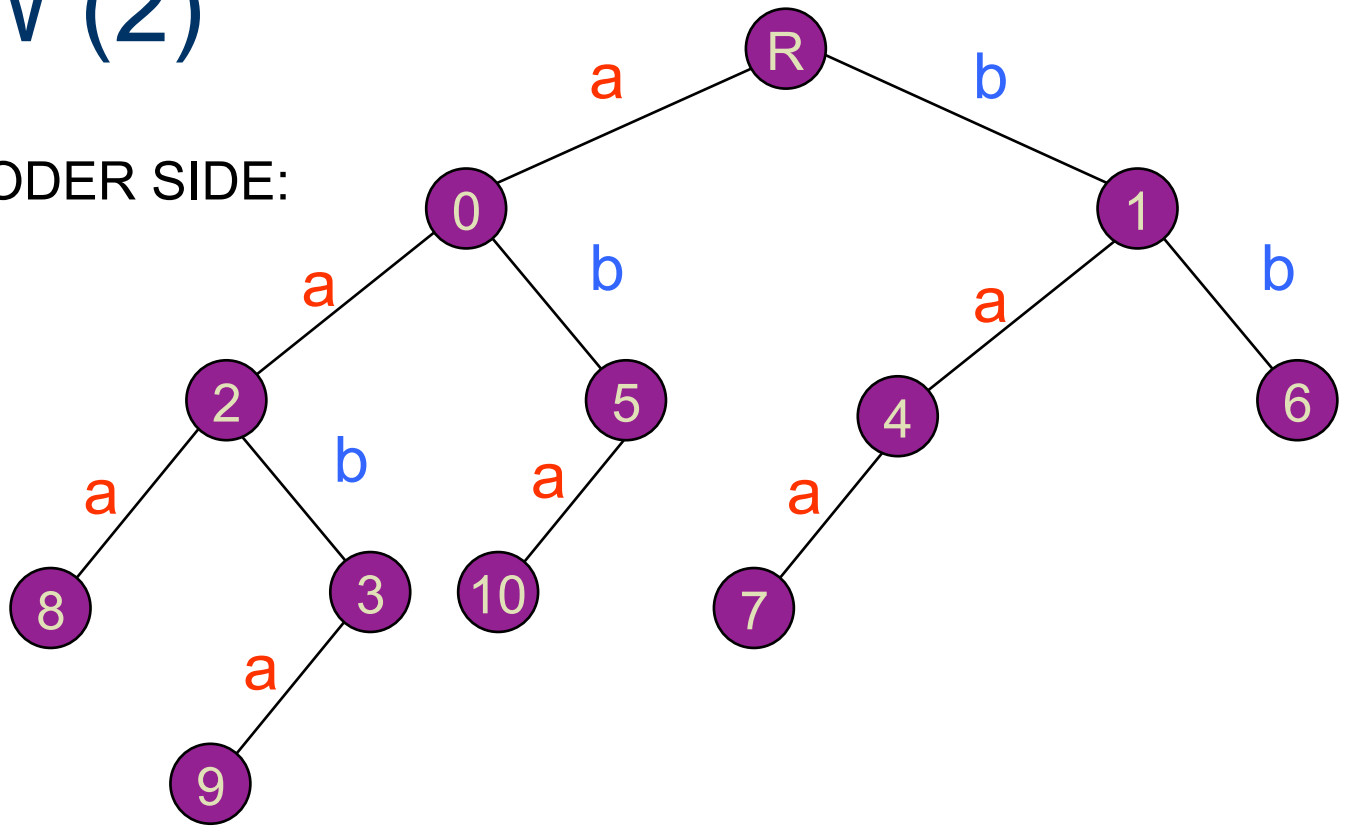


a a a b a b b a a a a a b a b a a b a a b a a a b a a b

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

LZW (2)

ENCODER SIDE:

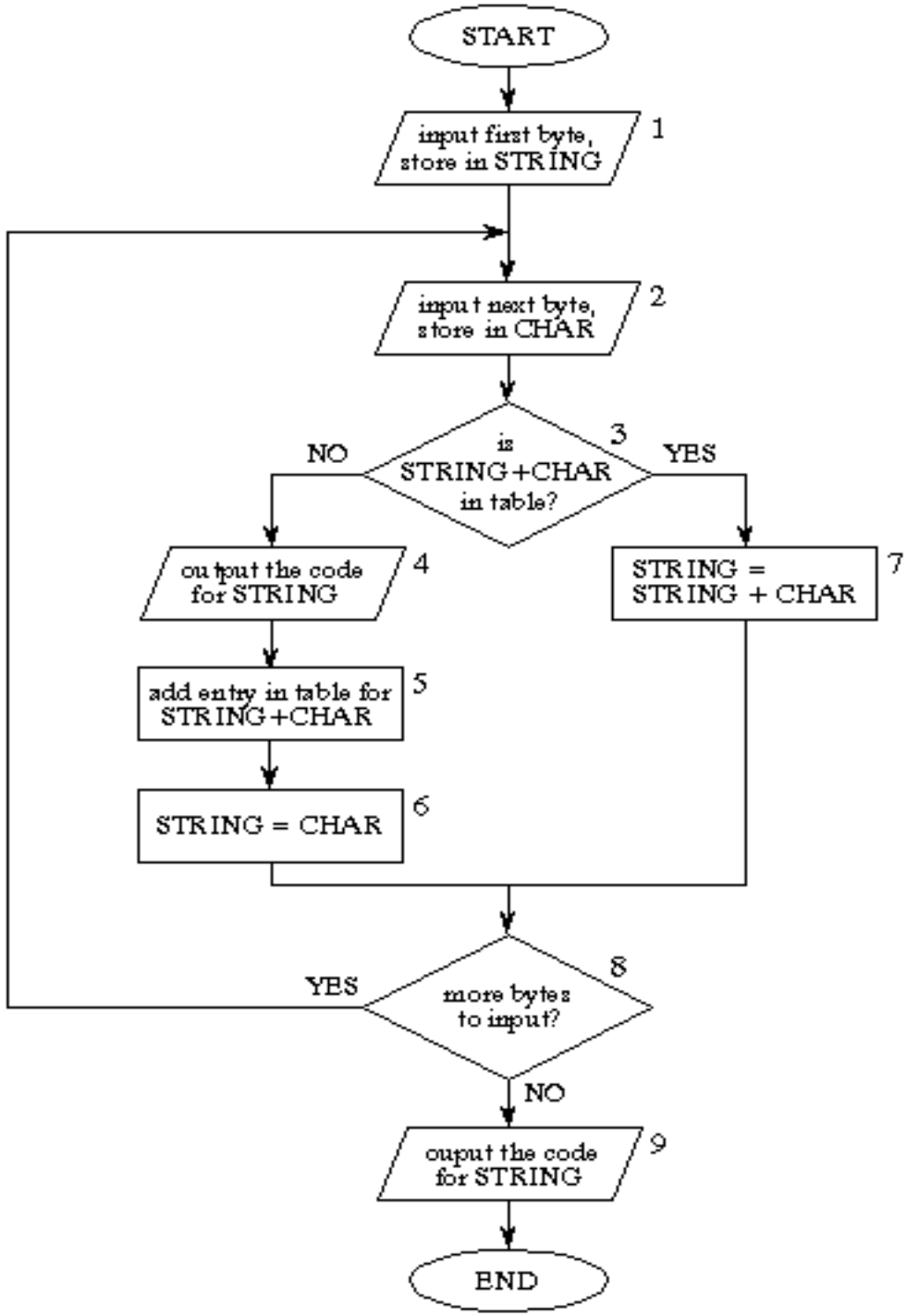


a a a b a b b a a a a a b a b a a b a a b a a a b a a b

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

LZW (2)

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

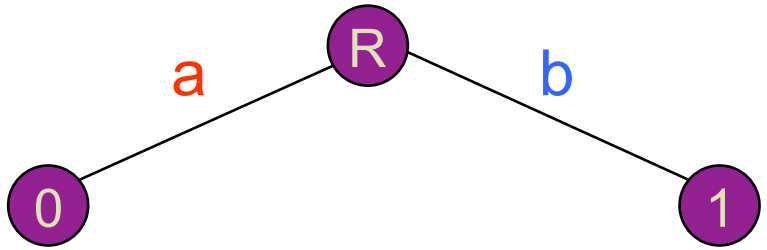


LZW (3)

- 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:

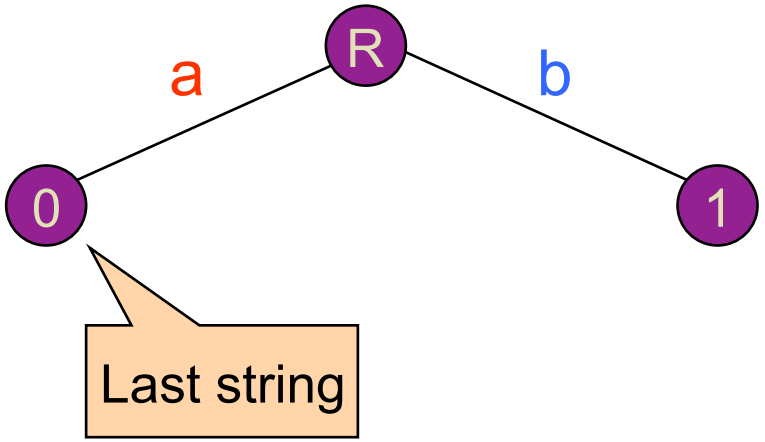


input: 0, 2, 1, 0, 1, 4, 2, 3, 5

output:

LZW (4)

DECODER SIDE:



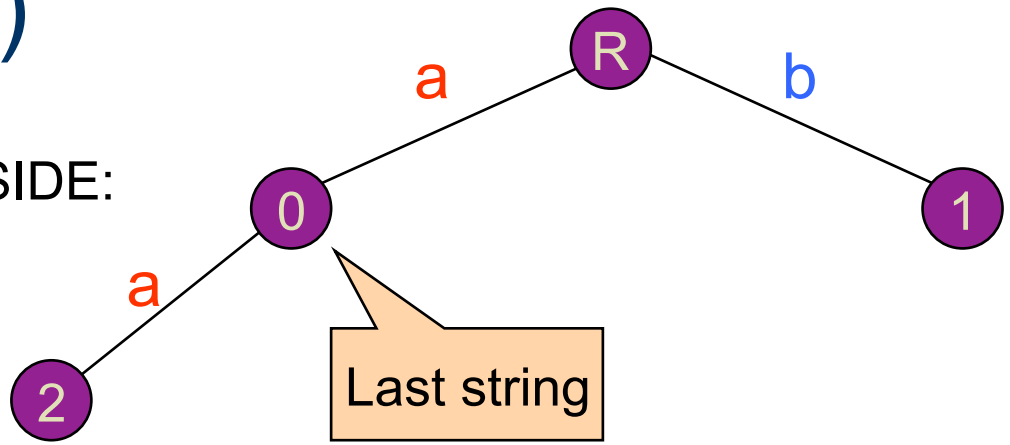
input: 0, 2, 1, 0, 1, 4, 2, 3, 5

output: a

Last string = a

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

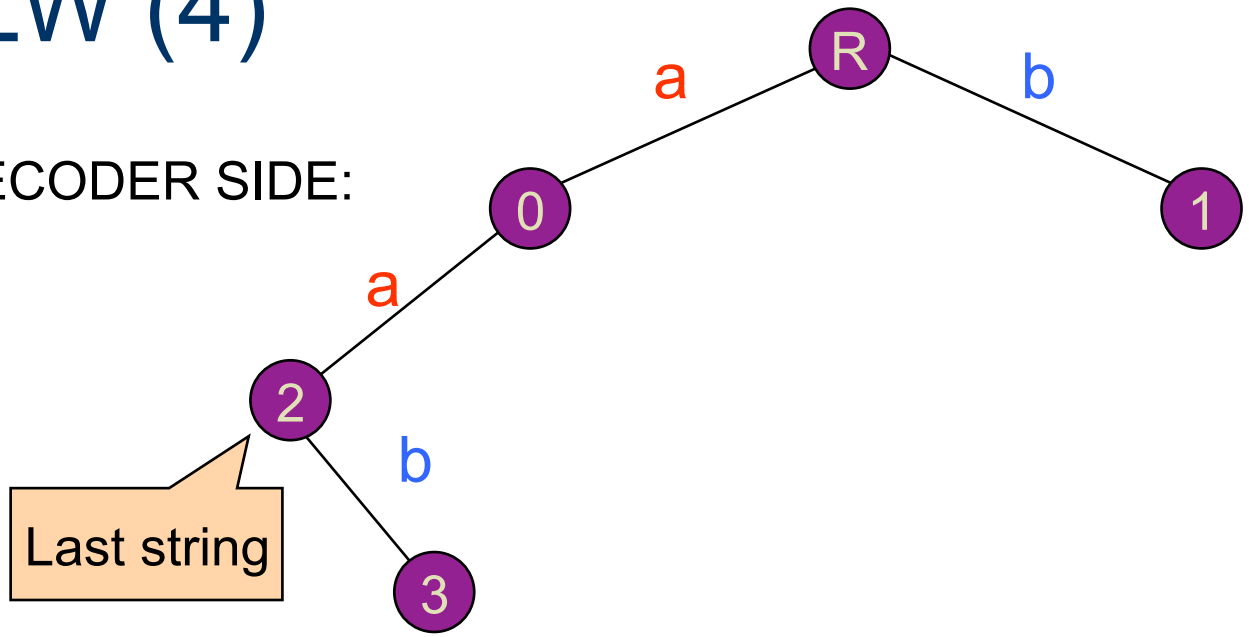
input: 0, 2, 1, 0, 1, 4, 2, 3, 5

output: a a a

Last string = a

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.

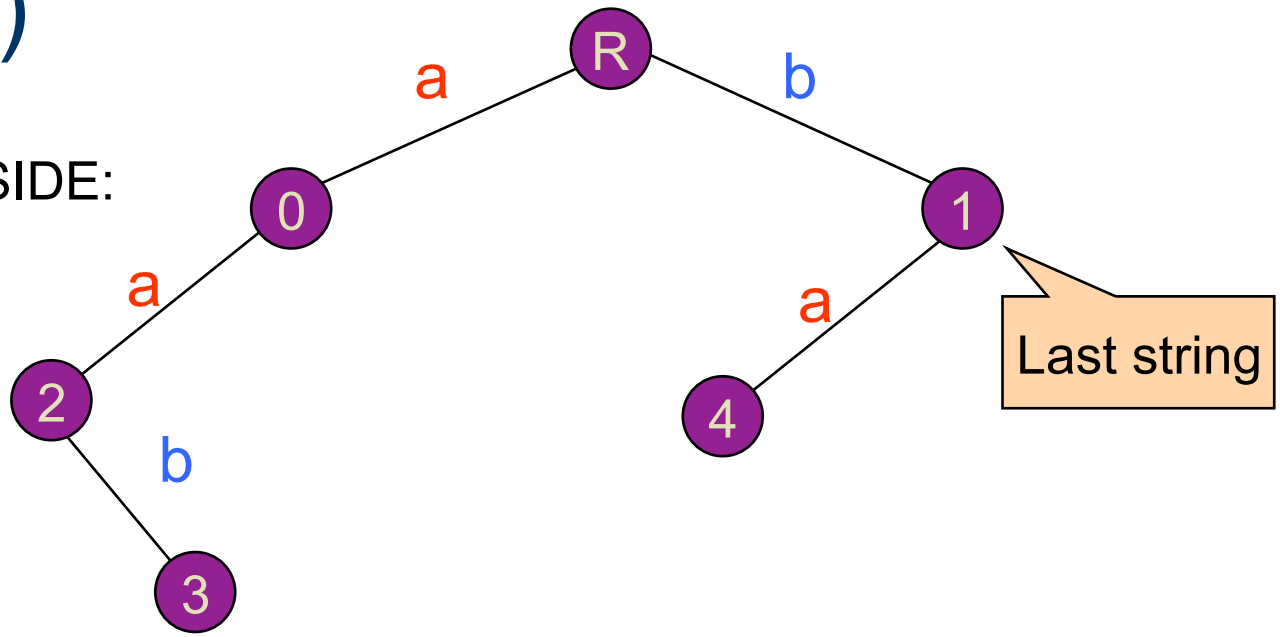
input: 0, 2, 1, 0, 1, 4, 2, 3, 5

output: a a a b

Last string = a a

LZW (4)

DECODER SIDE:



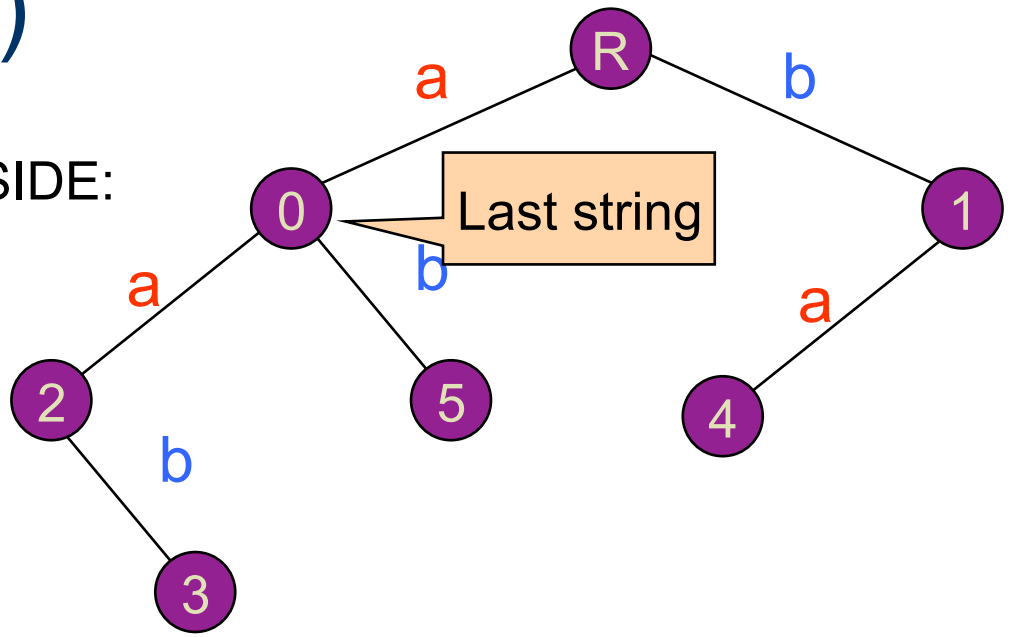
input: 0, 2, 1, 0, 1, 4, 2, 3, 5

output: a a a b a

Last string = b

LZW (4)

DECODER SIDE:



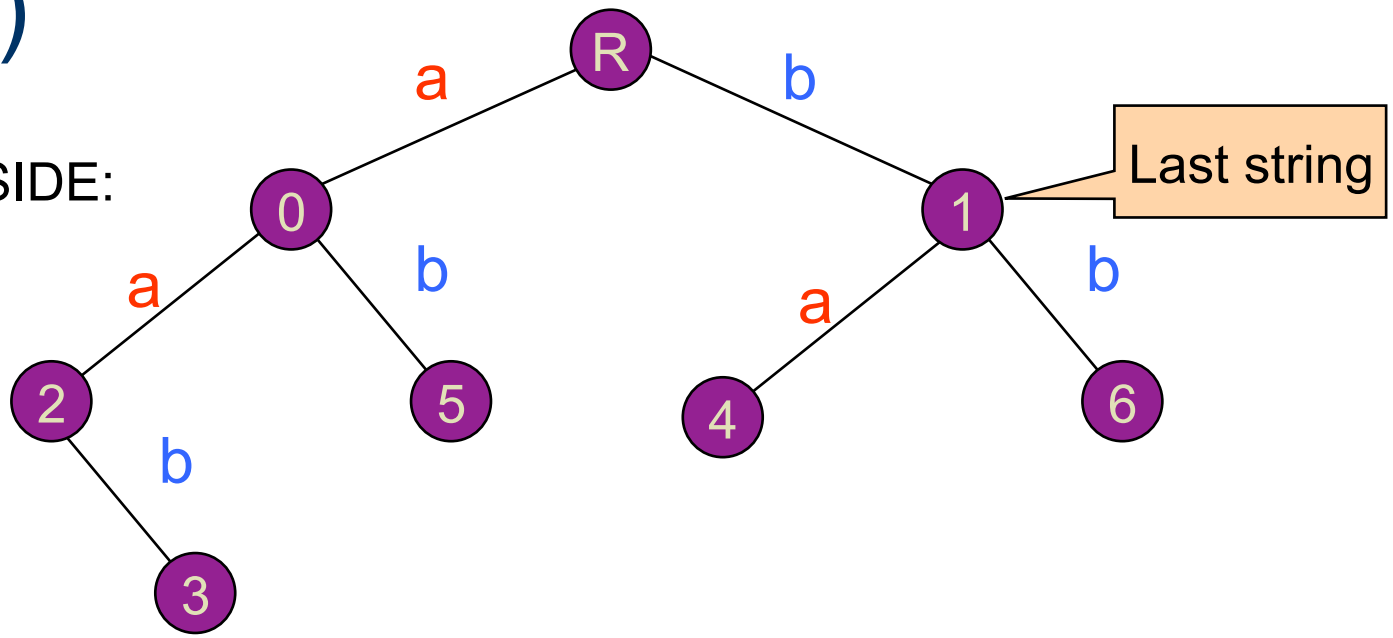
input: 0, 2, 1, 0, 1, 4, 2, 3, 5

output: a a a b a b

Last string = a

LZW (4)

DECODER SIDE:



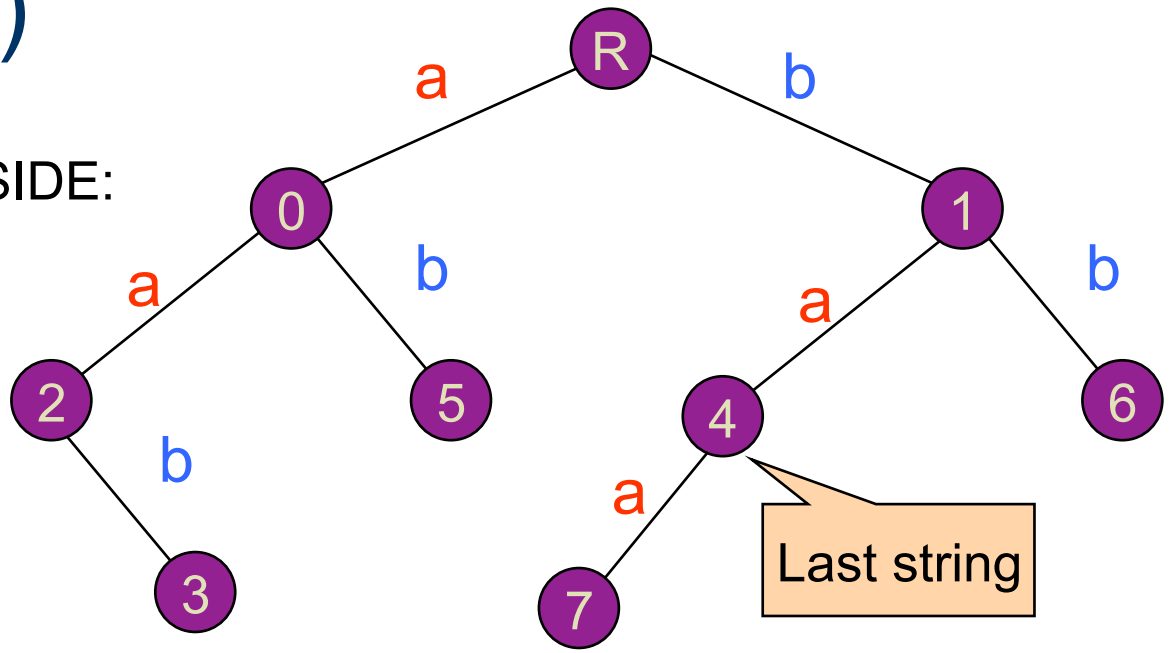
input: 0, 2, 1, 0, 1, 4, 2, 3, 5

output: a a a b a b b a

Last string = b

LZW (4)

DECODER SIDE:



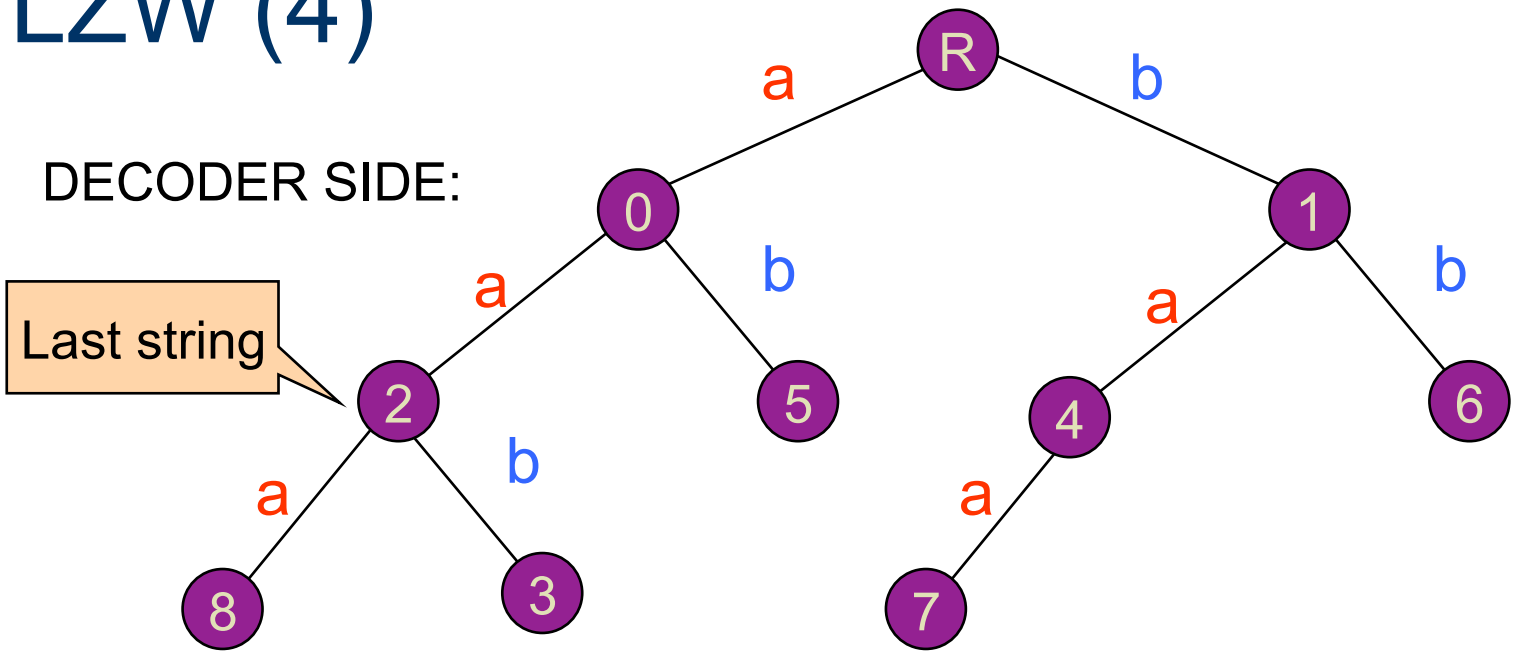
input: 0, 2, 1, 0, 1, 4, 2, 3, 5

Last string = b a

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

LZW (4)

DECODER SIDE:



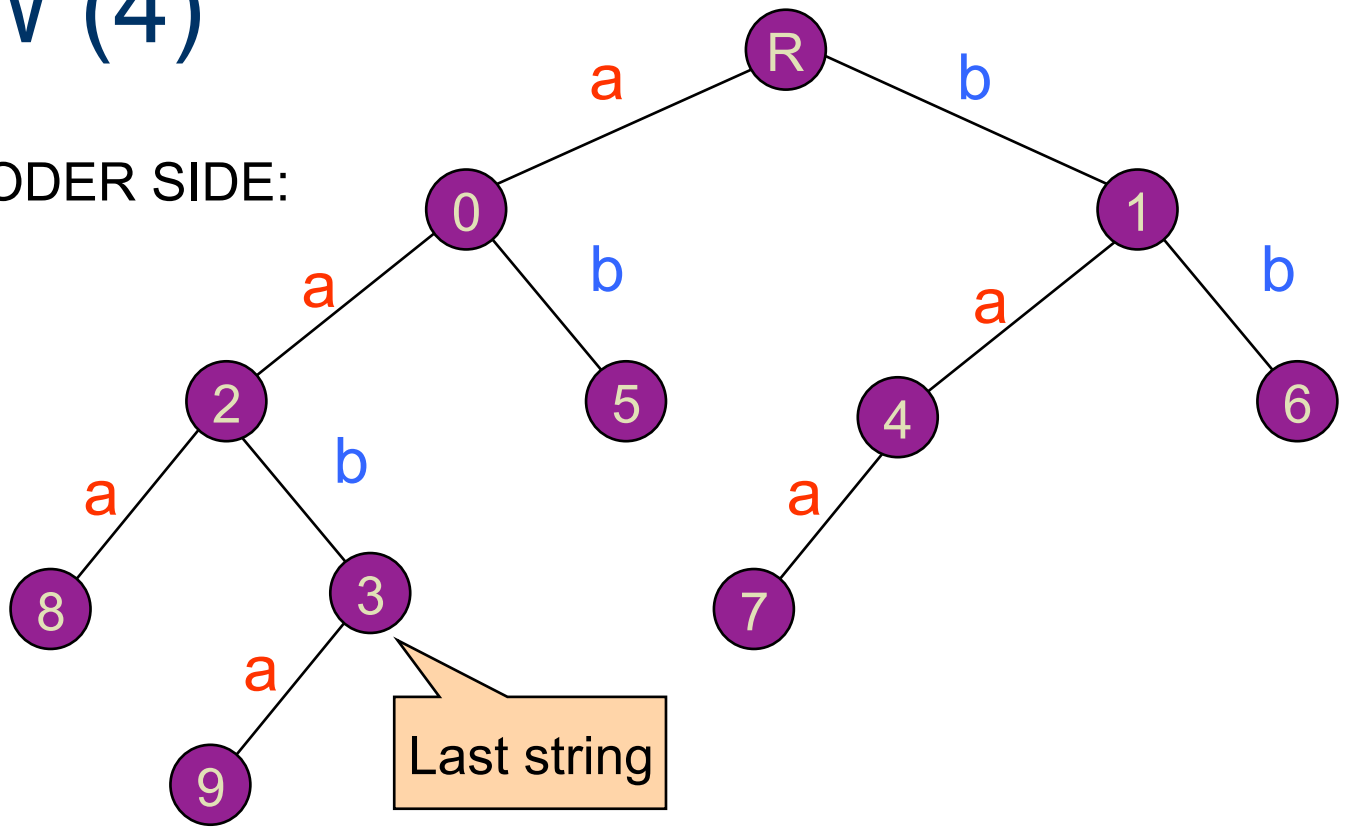
input: 0, 2, 1, 0, 1, 4, 2, 3, 5

Last string = a a

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

LZW (4)

DECODER SIDE:



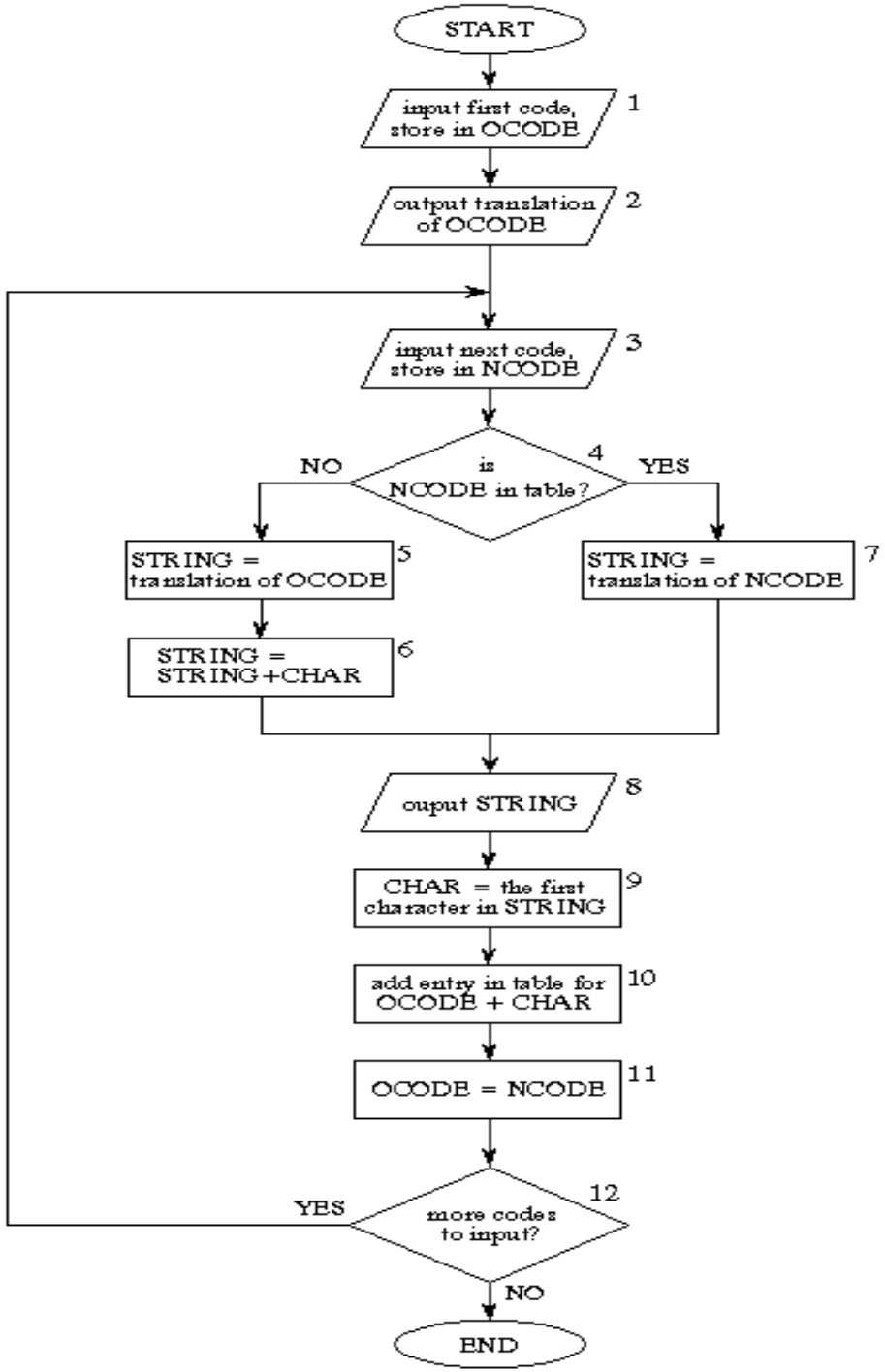
input: 0, 2, 1, 0, 1, 4, 2, 3, 5

Last string = a a b

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

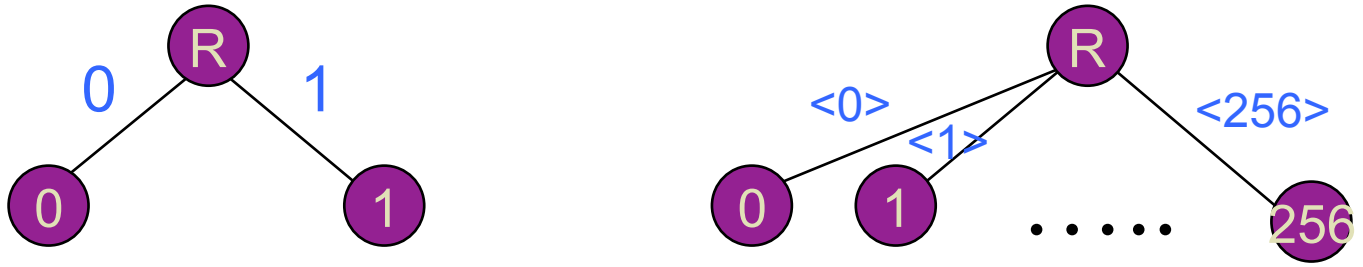
LZW (5)

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



Implementing LZW

- Instead of handling bit stream, real applications usually handle byte stream, i.e. there are 256 alphabets in the source. S = {<0>, …, a, b, … z, A, B, …, Z, 0, 1, … 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 starts with a 4k dictionary, of which entries 0-255 refer to individual bytes, and entries 256-4095 refer to substrings.

Implementing LZW (2)

- Same as the binary case
- Compression Rules:
  - Rule 1: whenever the pattern in the buffer is in the dictionary (table or LZW tree), we try to extend it.
  - Rule 2: whenever a pattern is added to the dictionary, we spill out the codeword of the pattern before the last extension.
- Let's try "^WED^WE^WEE^WEB^WET"

Example 1 (Compression)

^WED^WE^WEE^WEB^WET$	**Dict**	**Index**	**Output**	
^				
^W	^W	256	^	2c
WE	WE	257	W	2c
ED	ED	258	E	2c
D^	D^	259	D	2c
^W				seen at 256
^WE	^WE	260	256	3c from 256
E^	E^	261	E	2c
^W				seen at 256
^WE				seen at 260
^WEE	^WEE	262	260	4c from 260
E^				seen at 261
E^W	E^W	263	261	3c from 261
WE				seen at 257
WEB	WEB	264	257	3c from 257
B^	B^	265	B	2c
^W				seen at 256
^WE				seen at 260
^WET	^WET	266	260	4c from 260
T$			T	

Example 1

- 19 chars input reduced to 7 chars output plus 5 indices.
- Some compression is achieved. Usually no compression is achieved until a significant number of bytes (> 100) are read.
- Observation: 1 new entry is added to the dictionary for every codeword output.
- The compressor ***implicitly*** transmits the dictionary.
- What happen when the table is full?
- Restart to build another table.
- A hash table may be needed for fast searching of the string.

Example 1 (Decompression)

- “^WED{256}E{260}{261}{257}B{260}T” is received at the decoder side
- While scanning for decoding, the dictionary is rebuilt. When an index is read, the corresponding dictionary entry is retrieved to splice in position.
- The decoder deduces what the buffer of the encoder has gone through and rebuild the dictionary.
- Decompression is generally faster than compression. Why?
- Because searching during compression is time-consuming. No searching is required in decompression.

Example 1 (Decompression)

```
^WED{256}E{260}{261}{257}B{260}T$
^
^W
 WE
  ED
   D{256}
    {256}E
         E{260}
          {260}{261}
               {261}{257}
                    {257}B
                         B{260}
                          {260}T
                               T$
```

Insert Dict	Index	Output
		^
^W	256	W
WE	257	E
ED	258	D
D^	259	^W
^WE	260	E
E^	261	^WE
^WEE	262	E^
E^W	263	WE
WEB	264	B
B^	265	^WE
^WET	266	T

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