

# Algorithms for Data Compression

**[CLRS] – chap 16.3  
(plus some material not in  
Cormen)**

# Outline

- The Data compression problem
- Techniques for **lossless compression**:
  - Based on **codewords**
    - Huffman codes
  - Based on **dictionaries**
    - Lempel-Ziv, Lempel-Ziv-Welch

# The Data Compression Problem

- Compression: **transforming the way information is represented**
- Compression saves:
  - space (external storage media)
  - time (when transmitting information over a network)
- Types of compression:
  - **Lossless**: the compressed information can be decompressed into the original information
    - Examples: **zip**
  - **Lossy**: the decompressed information differs from the original, but ideally in an insignificant manner
    - Examples: **jpeg** compression

# Lossless compression

- The basic principle for lossless compression is to **identify and eliminate *redundant* information**
- Techniques used for codification:
  - Codewords
  - Dictionaries

# Codewords

- Each character is represented by a codeword (an unique binary string)
  - **Fixed-length codes**: all characters are represented by codewords of the same length (example: ASCII code)
  - **Variable-length codes**: frequent characters get short codewords and infrequent characters get longer

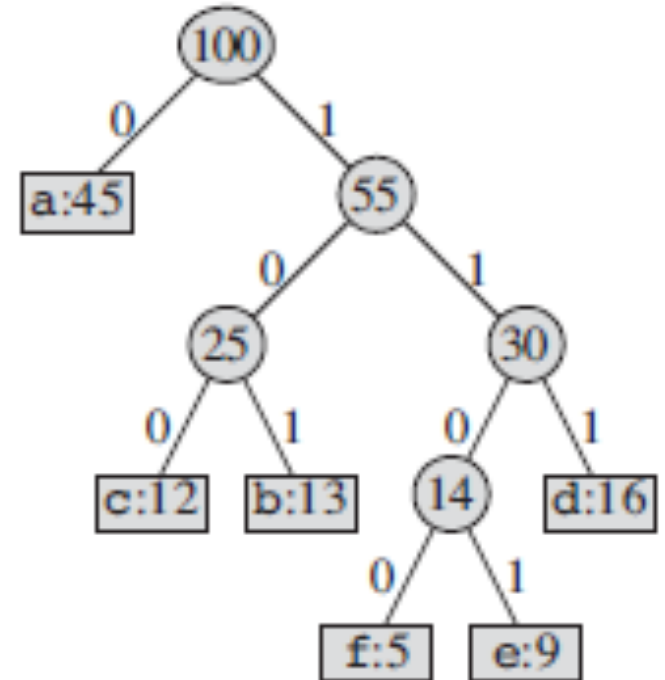
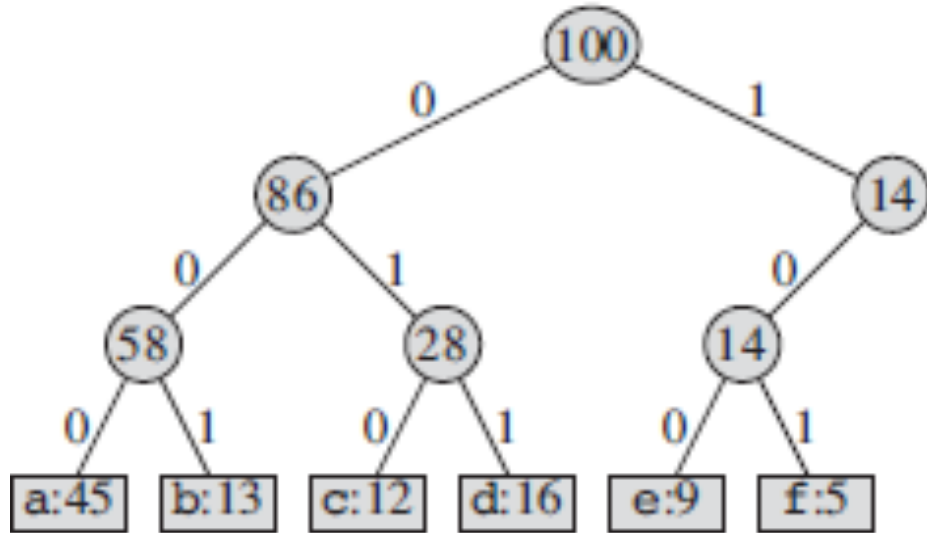
	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

# Prefix Codes

- A code is called a *prefix code* if no codeword is a prefix of any other codeword (actually “prefix-free codes” would be a better name)
- This property is important for being able to decode a message in a simple and unambiguous way:
  - We can match the compressed bits with their original characters as we decompress bits in order
  - Example: 0 0 1 0 1 1 10 1 is unambiguously decoded into aabe (assuming codes from previous table)

# Representation of Prefix Codes

- A binary tree whose leaves are the given characters. The codeword for a character is the simple path from the root to that character, where 0 means “go to the left child” and 1 means “go to the right child.”



# Constructing the optimal prefix code

- Given a tree  $T$  corresponding to a prefix code, we can compute the number of bits  $B(T)$  required to encode a file.
- For each character  $c$  in the alphabet  $C$ , let the attribute  $c.freq$  denote the frequency of  $c$  in the file and let  $d_T(c)$  denote the depth of  $c$ 's leaf in the tree.
- The number of bits  $B(T)$  required to encode a file is the Cost of the tree:

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c)$$

- $B(T)$  should be minimal !

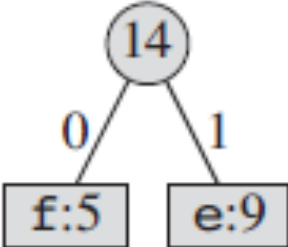


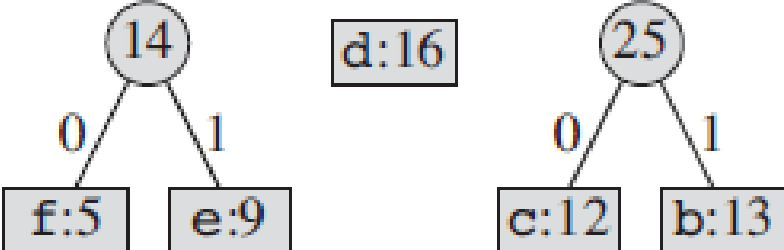
# Huffmann algorithm for constructing optimal prefix codes

- The principle of Huffman's algorithm is following:
- Input data: frequencies of the characters to be encoded
- The binary tree is built bottom->up
- We have a forest of trees that are united until one single tree results
- Initially, each character is its own tree
- Repeatedly find the two root nodes with lowest frequencies, create a new root with these nodes as its children, and give this new root the sum of its children frequencies

# Example - Huffman

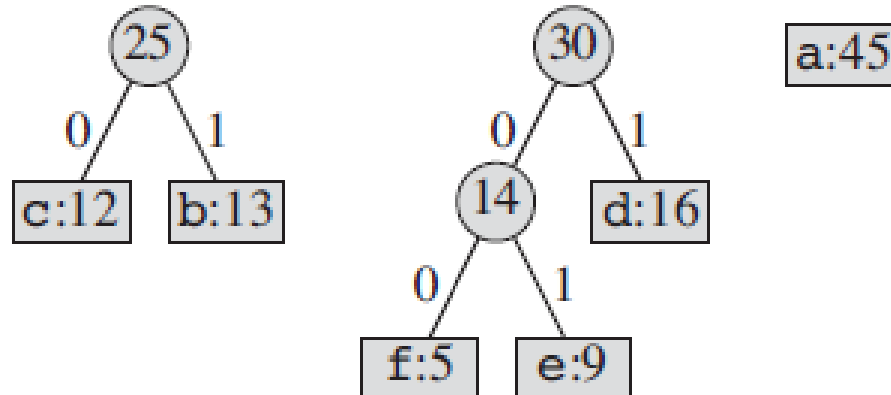
Step1: f:5 e:9 c:12 b:13 d:16 a:45

Step2: c:12 b:13 14 d:16 a:45  


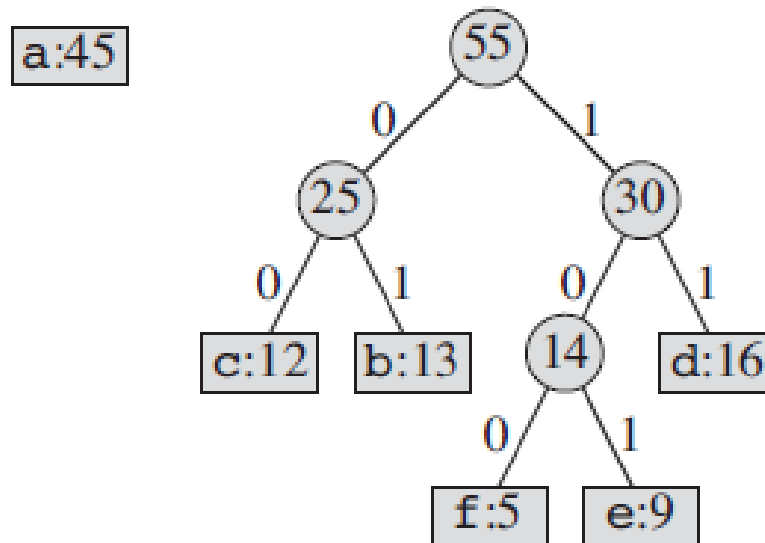
Step3: 14 d:16 25 a:45  


# Example – Huffman (cont)

Step 4:

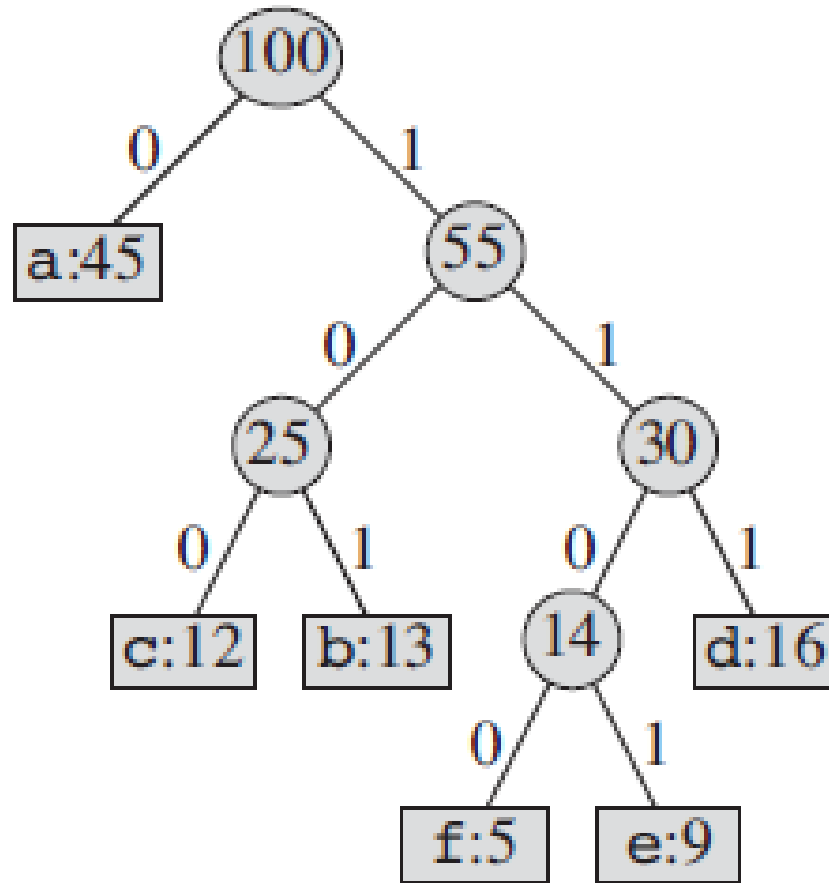


Step 5:



# Example – Huffman (final)

Step 6:



*Procedure* BUILD-HUFFMAN-TREE(*char*, *freq*, *n*)

*Inputs:*

- *char*: an array of *n* uncompressed characters.
- *freq*: an array of *n* character frequencies.
- *n*: the sizes of the *char* and *freq* arrays.

*Output:* The root of the binary tree constructed for Huffman codes.

1. Let  $Q$  be an empty priority queue.
2. For  $i = 1$  to  $n$ :
  - A. Construct a new node  $z$  containing  $char[i]$  and whose frequency is  $freq[i]$ .
  - B. Call INSERT( $Q, z$ ).
3. For  $i = 1$  to  $n - 1$ :
  - A. Call EXTRACT-MIN( $Q$ ), and set  $x$  to the node extracted.
  - B. Call EXTRACT-MIN( $Q$ ), and set  $y$  to the node extracted.
  - C. Construct a new node  $z$  whose frequency is the sum of  $x$ 's frequency and  $y$ 's frequency.
  - D. Set  $z$ 's left child to be  $x$  and  $z$ 's right child to be  $y$ .
  - E. Call INSERT( $Q, z$ ).
4. Call EXTRACT-MIN( $Q$ ), and return the node extracted.

# Huffman encoding

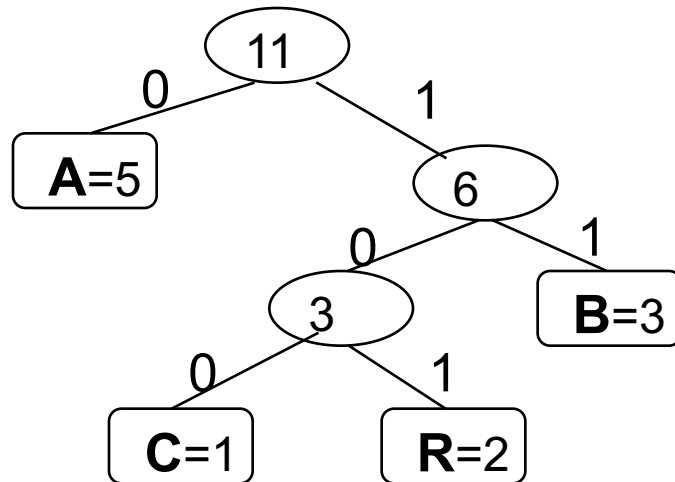
- Input: a text, using an alphabet of  $n$  characters
- Output: a Huffman codes table and the encoded text
- **Preprocessing:**
  - Computing frequencies of characters in text (requires one full pass over the input text)
  - Building Huffman codes
- **Encoding:**
  - Read input text character by character, replace every character by its code(=string of bits) and write output text

# Huffman decoding

- Input: a Huffman codes table and the encoded text
- Output: the original text
- Starting at the root of the Huffman tree, read one bit of the encoded text and travel down the tree on the left child(bit 0) or right child (bit 1) until arriving at a leaf. Write the decoded character (corresponding to the leaf) and resume procedure from the root.

# Huffman encoding - Example

- Input text: ABRACABABRA
- Compute char frequencies: A=5, B=3, R=2, C=1
- Build code tree:

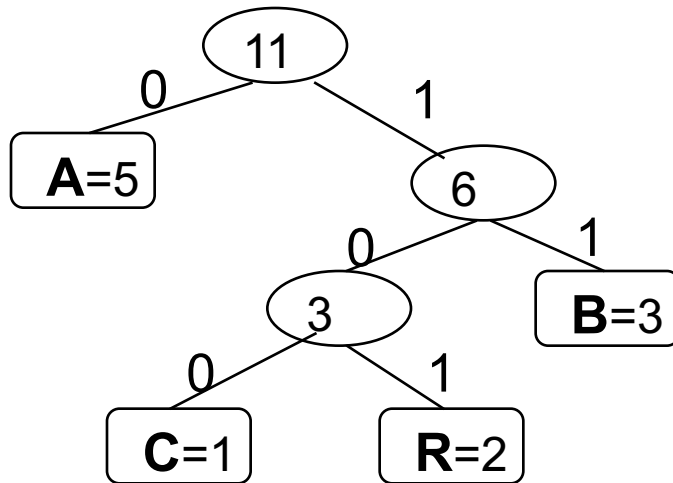


- Encoded text: 01110101000110111010    20 bits
- Coding of original text with fixed-length code:  $11 \times 2 = 22$  bits
- Attention ! **The output will contain the encoded text + coding information !** (in this small example, actual size of output will be bigger than input in this case)



# Huffman decoding - Example

- Input: coding information + encoded text
  - A=5, B=3, R=2, C=1
  - 01110101000110111010
- Build code tree:



- Decoded text:
- ABRACABABRA

# Huffman coding in practice

- Can be applied to compress as well binary files (characters = bytes, alphabet = 256 “characters”)
- Codes = strings of bits
- Implementing Encoding and Decoding involves *bitwise operations* !

# Disadvantages of Huffman codes

- Requires **two passes over the input** (one to compute frequencies, one for coding), thus encoding is slow
- Requires **storing the Huffman codes** (or at least character frequencies) **in the encoded file**, thus reducing the compression benefit obtained by encoding
- => these disadvantages can be improved by **Adaptive Huffman Codes** (also called *Dynamic Huffman Codes*)

# Principles of Adaptive Huffman

- Encoding and Decoding work adaptively, *updating character frequencies and the binary tree* as they compress or decompress in just one pass

# Adaptive Huffman encoding

The compression program starts with an empty binary tree.

```
While (input text not finished)
  Read character c from input
  If (c is already in binary tree) then
    Writes code of c
    Increases frequency of c
    If necessary updates binary tree
  Else
    Writes c unencoded ( + escape sequence)
    Adds c to the binary tree
```

# Adaptive Huffman decoding

The decompression program starts with an empty binary tree.

While (coded input text not finished)

    Read bits from input until reaching a code or the escape sequence

    If (bits represent code of a character c) then

        Write c

        Increases frequency of c

        If necessary updates binary tree

    Else

        Read bits of new character c

        Write c

        Adds c to the binary tree

# Adaptive Huffman

- Main issue: to *correctly* and *efficiently* update the code tree when adding new character or increasing the frequency of a character
  - one cannot just run the Huffman algo for *building* the tree every time one frequency gets modified
- Both the coder and the decoder must use *exactly the same* algo for updating code trees (otherwise decoding will not work !)
- Known solutions to this problem:
  - FGK algorithm (Faller, Gallagher, Knuth)
  - Vitter algorithm

# Outline

- The Data compression problem
- Techniques for **lossless compression**:
  - Based on codewords
    - Huffman codes
  - Based on **dictionaries**
    - Lempel-Ziv, Lempel-Ziv-Welch



# Dictionary-based encoding

- **Dictionary-based algorithms** do not encode single symbols as variable-length bit strings; they **encode variable-length strings of symbols as single tokens**
  - The tokens form an index into a phrase dictionary
  - If the tokens are smaller than the phrases they replace, compression occurs.

# Dictionary-based encoding example

- **Dictionary:**

1. ASK
2. NOT
3. WHAT
4. YOUR
5. COUNTRY
6. CAN
7. DO
8. FOR
9. YOU

- **Original text:**

- ASK NOT WHAT YOUR COUNTRY CAN  
DO FOR YOU ASK WHAT YOU CAN  
DO FOR YOUR COUNTRY

- **Encoded based on dictionary :**

- 1 2 3 4 5 6 7 8 9 1 3 9 6 7 8  
4 5

# Dictionary-based encoding in practice

- Problems in practice:
  - Where is the dictionary ? (external/internal) ?
  - Dictionary is known in advance (static) or not ?
  - Size of dictionary is large -> size of dictionary index word may be comparable or bigger than some words
    - If index word is on 4 bytes => dictionary may hold  $2^{32}$  words

# LZ-77

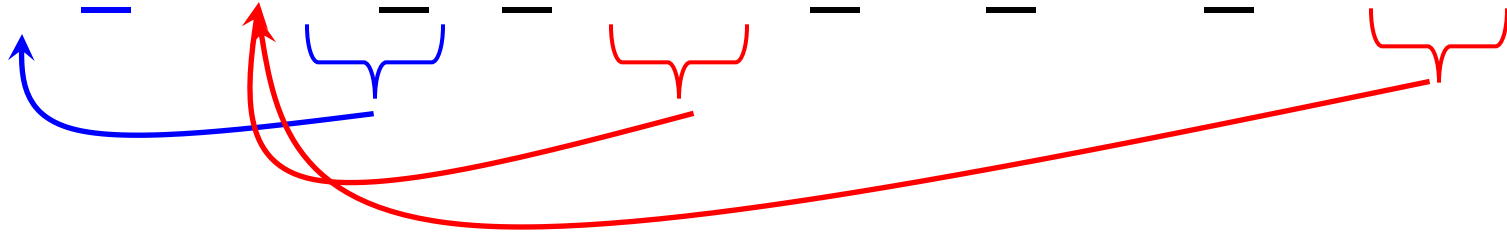
- Abraham Lempel & Jacob Ziv: 1977: proposed a dictionary-based approach for compression
  - Idea:
    - *dictionary is actually the text itself*
    - First occurrence of a “word” in input => “word” is written in output
    - Next occurrences of a “word” in input => **instead of writing “word” in output, write only a “reference” to its first occurrence**
  - “word”: any sequence of characters
  - “reference”: A match is encoded by a *length-distance pair*, meaning: **“the next length characters are equal to the characters exactly distance characters behind it in the input”**.

# LZ-77 Principle Example

- Input text:
- IN\_SPAIN\_IT\_RAINS\_ON\_THE\_PLAIN

- Coding:

• IN\_SPAIN\_IT\_RAINS\_ON\_THE\_PLAIN



- Coded output:

• IN\_SPA{3,6}IT\_R{3,8}S\_ON\_THE\_PL{3,22}

# LZ-78 and LZW

- Lempel-Ziv 1978
  - Builds an explicit Dictionary structure of all character sequences that it has seen and uses indices into this dictionary to represent character sequences
- Welch 1984 -> LZW
  - The dictionary is not empty at start, but initialized with 256 single-character sequences (the  $i$ th entry is ASCII code  $i$ )

# LZW compressing principle

- The **compressor** builds up strings, inserting them into the dictionary and producing as output indices into the dictionary.
- The compressor builds up strings in the dictionary **one character at a time, so that whenever it inserts a string into the dictionary, that string is the same as some string already in the dictionary but extended by one character.** The compressor manages a string  $s$  of consecutive characters from the input, maintaining **the invariant** that the **dictionary always contains  $s$  in some entry** (even if  $s$  is a single character)

*Procedure LZW-COMPRESSOR* (*text*)

*Input: text:* A sequence of characters in the ASCII character set.

*Output:* A sequence of indices into a dictionary.

1. For each character *c* in the ASCII character set:
  - A. Insert *c* into the dictionary at the index equal to *c*'s numeric code in ASCII.
2. Set *s* to the first character from *text*.
3. While *text* is not exhausted, do the following:
  - A. Take the next character from *text*, and assign it to *c*.
  - B. If *s c* is in the dictionary, then set *s* to *s c*.
  - C. Otherwise (*s c* is not yet in the dictionary), do the following:
    - i. Output the index of *s* in the dictionary.
    - ii. Insert *s c* into the next available entry in the dictionary.
    - iii. Set *s* to the single-character string *c*.
4. Output the index of *s* in the dictionary.



# LZW Compressor Example

- Input text: TATAGATCTTAATATA
- Step 1: initialize dictionary with entries indices 0-255, corresponding to all ASCII characters
- Step 2: s=T
- Step 3:

# LZW Compressor Example (cont)

Input text: TATAGATCTTAATATA

Iteration	<i>s</i>	<i>c</i>	Output	New dictionary string
1	T	A	84 (T)	256: TA
2	A	T	65 (A)	257: AT
3	T	A		
4	TA	G	256 (TA)	258: TAG
5	G	A	71 (G)	259: GA
6	A	T		
7	AT	C	257 (AT)	260: ATC
8	C	T	67 (C)	261: CT
9	T	T	84 (T)	262: TT
10	T	A		
11	TA	A	256 (TA)	263: TAA
12	A	T		
13	AT	A	257 (AT)	264: ATA
14	A	T		
15	AT	A		
step 4	ATA		264 (ATA)	

# LZW Decompressing principle

- Input: a sequence of indices only.
- The dictionary does not have to be stored with the compressed information, **LZW decompression rebuilds the dictionary directly from the compressed information !**
- Like the compressor, the decompressor seeds the dictionary with the 256 single-character sequences corresponding to the ASCII character set. It reads a sequence of indices into the dictionary as its input, and **it mirrors what the compressor did to build the dictionary.** Whenever it produces output, it's from a string that it has added to the dictionary.

### Procedure LZW-DECOMPRESSOR (*indices*)

*Input: indices*: a sequence of indices into a dictionary, created by LZW-COMPRESSOR.

*Output*: The text that LZW-COMPRESSOR took as input.

1. For each character *c* in the ASCII character set:
  - A. Insert *c* into the dictionary at the index equal to *c*'s numeric code in ASCII.
2. Set *current* to the first index in *indices*.
3. Output the string in the dictionary at index *current*.
4. While *indices* is not exhausted, do the following:
  - A. Set *previous* to *current*.
  - B. Take the next number from *indices* and assign it to *current*.
  - C. If the dictionary contains an entry indexed by *current*, then do the following:
    - i. Set *s* to be the string in the dictionary entry indexed by *current*.
    - ii. Output the string *s*.
    - iii. Insert, into the next available entry in the dictionary, the string at the dictionary entry indexed by *previous*, concatenated with the first character of *s*.
  - D. Otherwise (the dictionary does not yet contain an entry indexed by *current*), do the following:
    - i. Set *s* to be the string at the dictionary entry indexed by *previous*, concatenated with the first character of this dictionary entry.
    - ii. Output the string *s*.
    - iii. Insert, into the next available entry in the dictionary, the string *s*.

# LZW Decompressor Example

Input: indices: 84, 65, 256, 71, 257, 67, 84, 256, 257, 264

Iteration	<i>previous</i>	<i>current</i>	Output ( <i>s</i> )	New dictionary string
Steps 2, 3		84	T	
1	84	65	A	256: TA
2	65	256	TA	257: AT
3	256	71	G	258: TAG
4	71	257	AT	259: GA
5	257	67	C	260: ATC
6	67	84	T	261: CT
7	84	256	TA	262: TT
8	256	257	AT	263: TAA
9	257	264	ATA	264: ATA

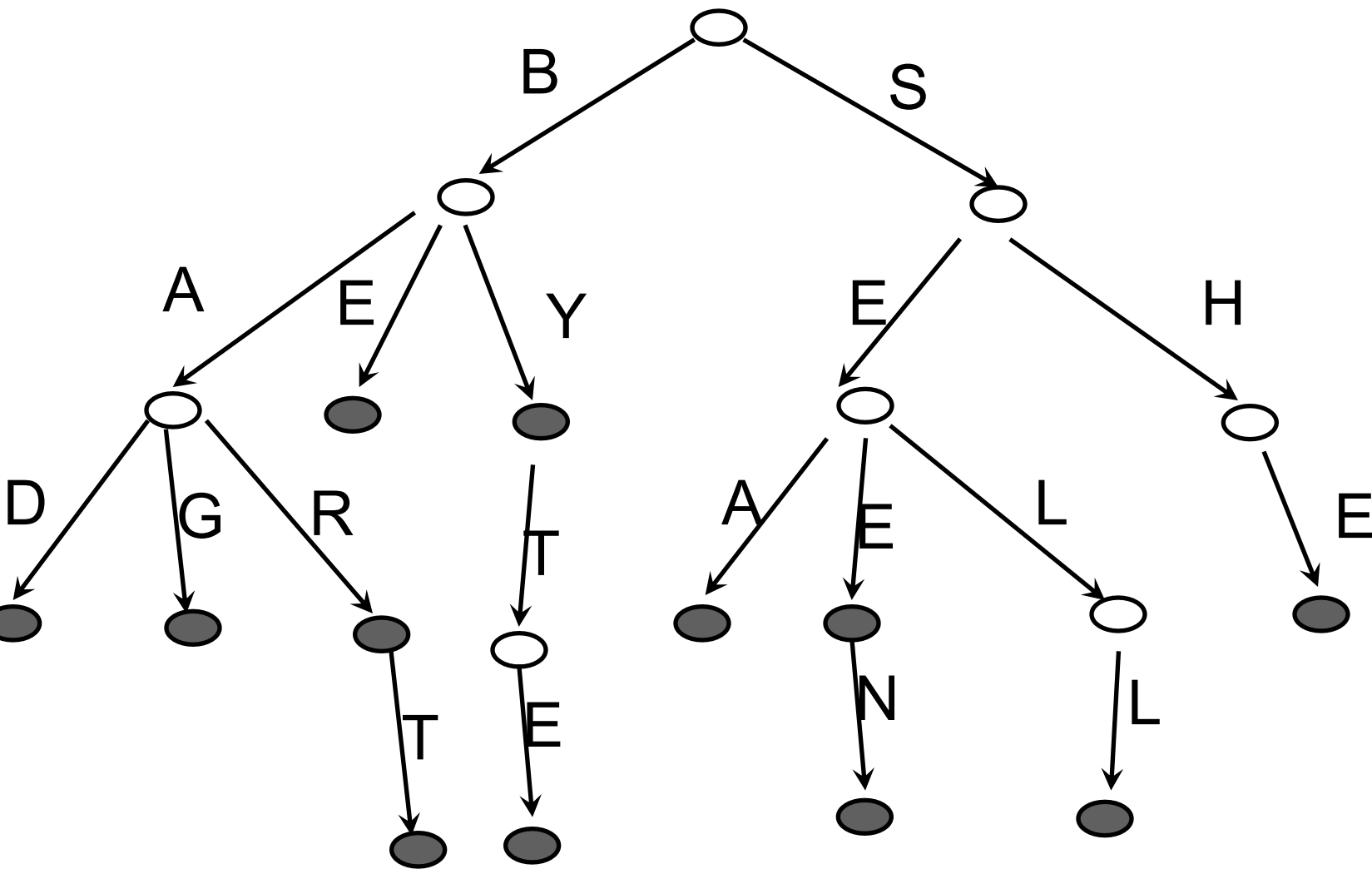
# LZW Implementation

- Dictionary has to be implemented in an efficient way
  - Tries
  - Hashtables

# Problem:

- A dictionary structure, used to store a *dynamic set* of *words* that may contain *common prefixes*.
  - *Word*=string of elements from an alphabet;  
*alphabet*=the set of all possible elements in the words
- Solution:
  - We can exploit the common prefixes of the words, and associate the words(the elements of the set) with *paths in a tree* instead of nodes of a tree
  - Solution: *Tries*
  - *Multipath trees*
  - *If the alphabet has  $N$  symbols, a node may have  $N$  children*

# Example Trie



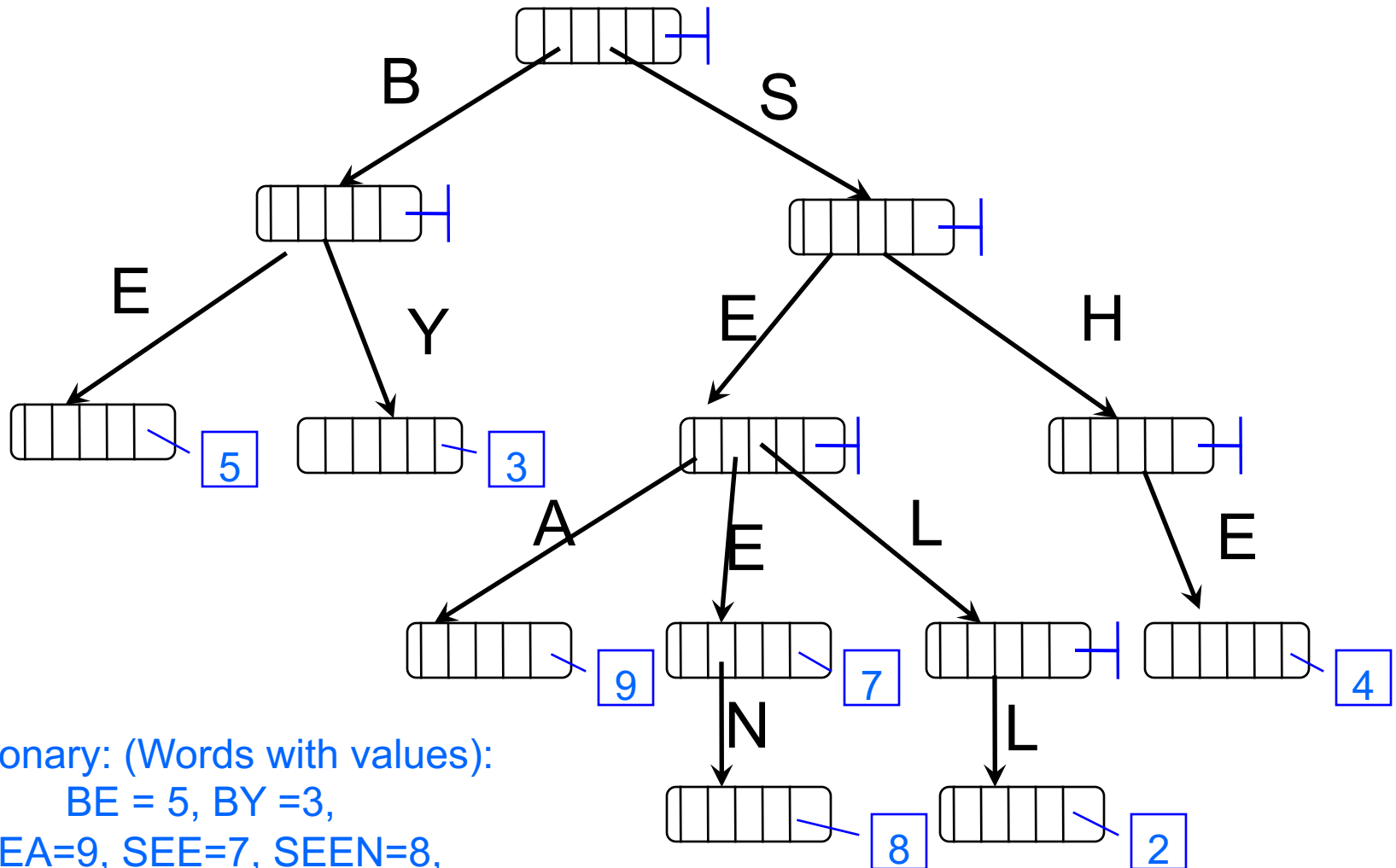
- BAD
- BAG
- BAR
- BART
- BE
- BY
- BYTE
- SEA
- SEE
- SEEN
- SELL
- SHE



# Tries

- **Trie tree:** tree structure whose edges are labeled with the elements of the alphabet.
- Each node can have up to  $N$  children, where  $N$  is the size of the alphabet.
- Nodes correspond to strings of elements of the alphabet: **each node corresponds to the sequence of elements traversed on the path from the root to that node.**
- The dictionary maps a string  $s$  to a value  $v$  by storing the value  $v$  in the node of  $s$ . If a string that is a prefix of another string in the dictionary is not used, it has nil as its value.
- **Possible implementation:** a trie tree node structure contains an array of  $N$  links to child nodes (a link to a child node can be also nil) and a link to the current strings value (it can be also nil).

# Trie Tree Example



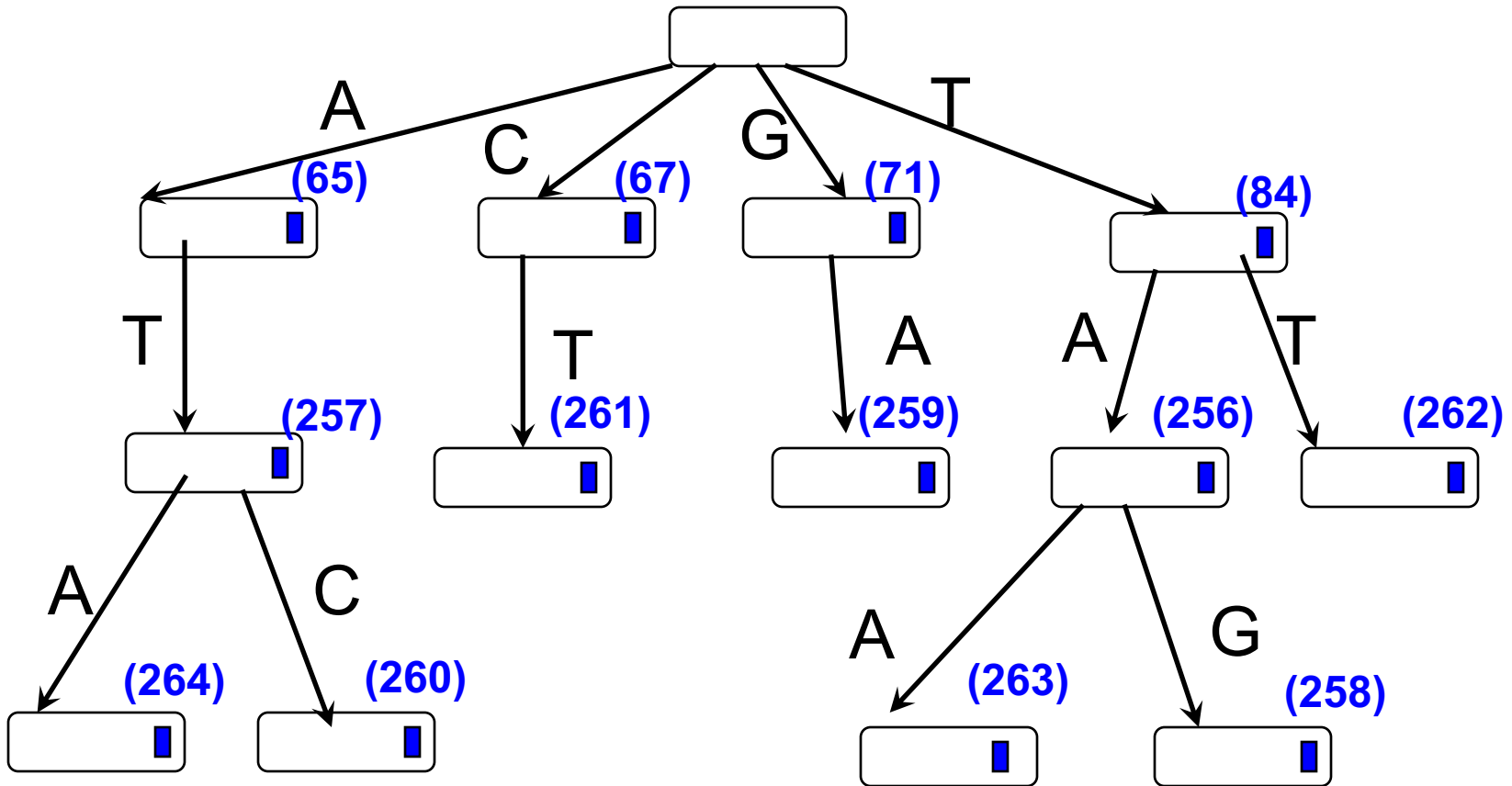
Dictionary: (Words with values):

BE = 5, BY = 3,

SEA=9, SEE=7, SEEN=8,

SELL=2, SHE=4

# Dictionary with Trie tree - Example



Words in dictionary: A, C, G, T, AT, CT, GA, TA, TT, ATA, ATC, TAA, TAG

# LZW Efficiency

- Biggest problem: **size of dictionary is large** => indices need several bytes to be represented => **compression rate is low**
- Possible measures:
  - Run Huffman encoding on LZW output (will work well because many indices in the LZW sequence are from the lower part)
  - Limit size of dictionary
    - once the dictionary reaches a maximum size, no other entries are ever inserted.
    - In another approach, once the dictionary reaches a maximum size, it is cleared out (except for the first 256 entries), and the process of filling the dictionary restarts from the point in the text

# Data compression in practice

- Known file compression utilities:
  - Gzip, PKZIP, ZIP: the DEFLATE approach( 2 phases compression, applying **LZ77** and **Huffman**)
  - Compress(UNIX distribution compressing tool ): **LZW**
- Microsoft NTFS : a modified **LZ77**
- Image formats:
  - GIF: **LZW**
- Fax machines: a modified Huffman encoding
- LZ77: free to use => in open-source sw
- LZ78, LZW: was protected by many patents