CMPUT 274

Bits & Bytes, File Compression

Topics Covered:

- Data as bits
- Compression
- Huffman coding

Bit

- **bit** = binary digit
- A bit is the basic unit of memory in computer
- A single bit can have the value 0 or 1
- Consider multiple bits together to get more combinations, store more meaningful data

  → 8 bits = 1 byte

  (4 bits = 1 nibble)

Characters in Text Files

- Dealing with data as bits is cumbersome
- More natural to think of data as series of characters (e.g. in a text file)
  - e.g. 0-9, a-z, A-Z, $, @, %, &, *, (,), -, +, ?, etc
- (extended) ASCII table defines binary representation of 256 ($=2^8$) different characters

  → each character has an 8 bit representation: 1 byte

Always Need All Those Bits?

- If 1 character = 1 byte of data, a text file containing 1000 characters contains approximately 1kB of data
- But what if the text file only contains the characters 'a' and 'b'?
- We wouldn't need full byte to represent the characters in that particular file

  → just use a single bit: 'a' = 0, 'b' = 1
- Using this approach, the 1000 a's and b's would take up 1/8 the space of normal ASCII representation

Data Compression

Compression

- Basic Goal of Compression:

  Represent a file using fewer bits, even if we have to store file contents in an unconventional format

- Benefits:
  - Use less memory to store the file
  - Transmit files faster

Another Example

- What if text file only has characters 'a', 'b', and 'n'?

`banana`

- What is a good compression technique?
- Can use:
  - 'a' = 00
  - 'b' = 01
  - 'n' = 10
- So banana → 010010001000
- This gives a compression rate of ¼ that of a plain-text file

Better Compression Choice?

- Is there a better way to compress a text file with only 3 characters?

- Yes!
  - Associate the most frequent character with 0, and the remaining two with 10 and 11

- Compression Rate:
  - At least 1/3 of the characters go from 8 bits to 1 bit
  - Remaining characters go from 8 bits to 2 bits

Calculate Compression Rate

- n_a = number of times 'a' occurs
- n_b = number of times 'b' occurs
- n_n = number of times 'n' occurs

$\boxed{n = n_a + n_b + n_n}$

- Suppose 'a' is most frequent. Then $n_a \geq n/3$
- Number of bits used in the compressed string is:

$$\begin{aligned} \text{bits} &= 1*n_a + 2*n_b + 2*n_n \\ &= 2*n - n_a \\ &\leq 2*n - n/3 \\ &\leq 5/3n \end{aligned}$$

- Number of bytes is (5/3n)/8 = **0.2083n or less**
- Better than the **0.25n** we got before when all three characters were a 2 bit sequence

Decoding

- Can we decode a compressed file when some characters were encoded with 1 bit, others with 2 bits?

- Example:

  decode: 10010011

  when 'a' = 0, 'b' = 10, 'n' = 11

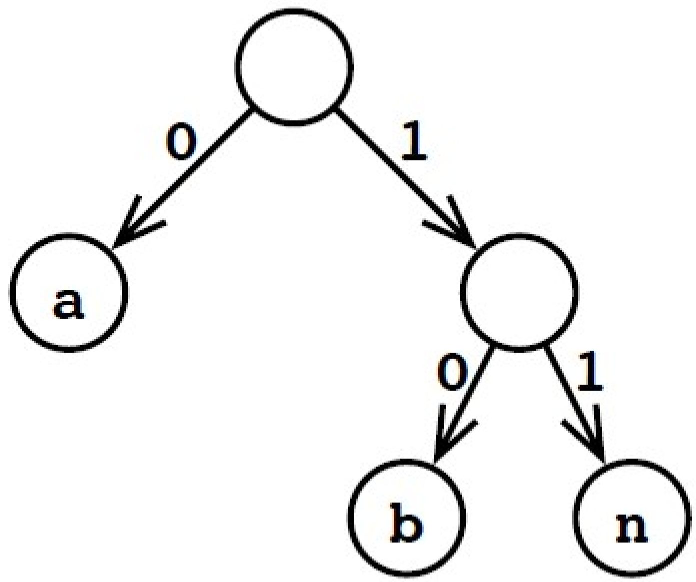
- Notice: no bit sequence is the beginning of another bit sequence. Therefore, can uniquely determine how to decode!

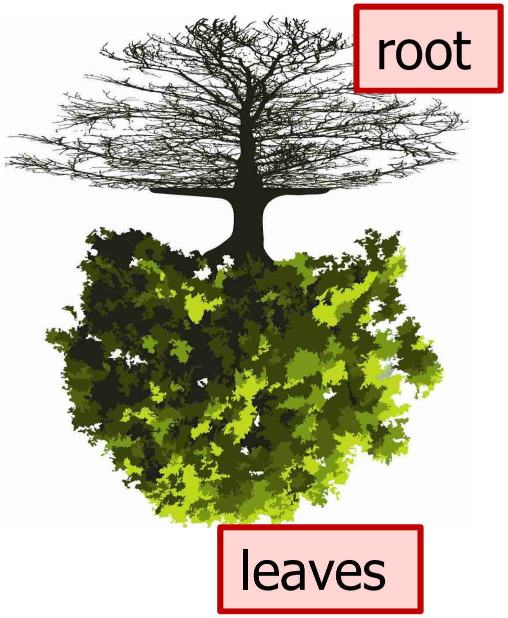
  10010011

  b a b a n

Decoding Tree

- If no bit sequence is the beginning of another in our encoding, we can build a decoding tree



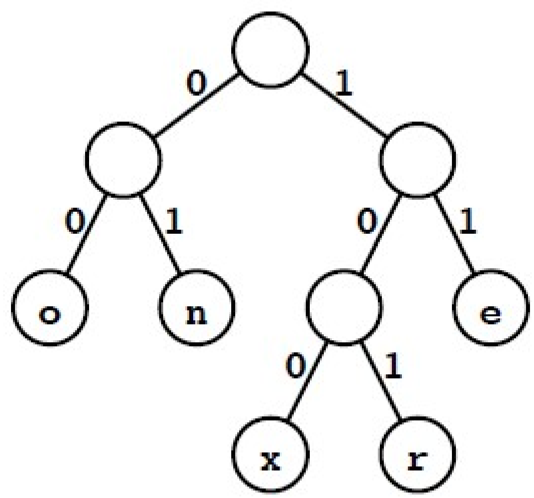


- Each character is a leaf node

- 0/1 labels on the edges of the root-to-leaf path = encoding of the character in a given leaf

How to Use Decoding Tree

- Decode a bit sequence by using the bits to traverse the given tree.
  - Start at the root, follow the 0/1 edge according to the next bit in the sequence
  - Output the character at a leaf whenever you reach one
  - Return to root and repeat for next part of sequence



- 001001101 → 00 (o) 100 (x) 11 (e) 01 (n) oxen

Build a Decoding Tree

- The key is picking the encoding for each character
- **Requirement:** no bit sequence for any character is the beginning (prefix) of another bit sequence.
  - → This type of encoding scheme is called a prefix code
- **Desire:** characters that occur more frequently should have shorter bit sequences
- **Optimization Problem:** construct a decoding tree to minimize total number of bits used to compress the file
- This can be achieved using Huffman Trees: trees constructed according to a simple <u>greedy procedure</u>

Greedy Algorithms

- A greedy algorithm works in steps
- At each step
  - take the best step one can get <u>right now</u>, without regard to the eventual optimization
  - hope that by choosing a local optimum at each step, one will end up at a global optimum
- e.g: count out $6.39, using the fewest bills and coins

→ **Greedy algorithm**: at each step, take the largest bill or coin that does not overshoot

- one $5 bill
- one $1 coin, to make $6
- one 25¢ coin, to make $6.25
- one 10¢ coin, to make $6.35
- four 1¢ coins, to make $6.39

Huffman Coding

Building a Huffman Tree

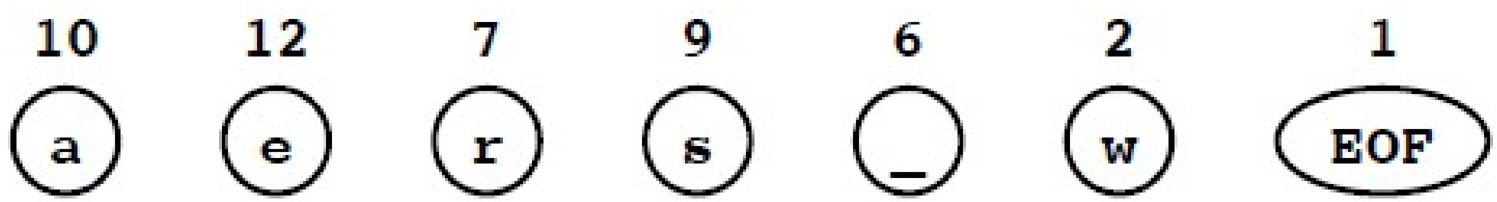
- First, do a frequency count of all characters that appear in the file
- Example

freq['a'] = 10	freq[' '] = 6
freq['e'] = 12	freq['w'] = 2
freq['r'] = 7	freq[EOF] = 1
freq['s'] = 9	

- For technical reasons, we include a special EOF (end of file) sentinel in our compression, even though it is not in the original file.
- **Note:** Ultimately keys will be bytes, not characters

Building a Huffman Tree

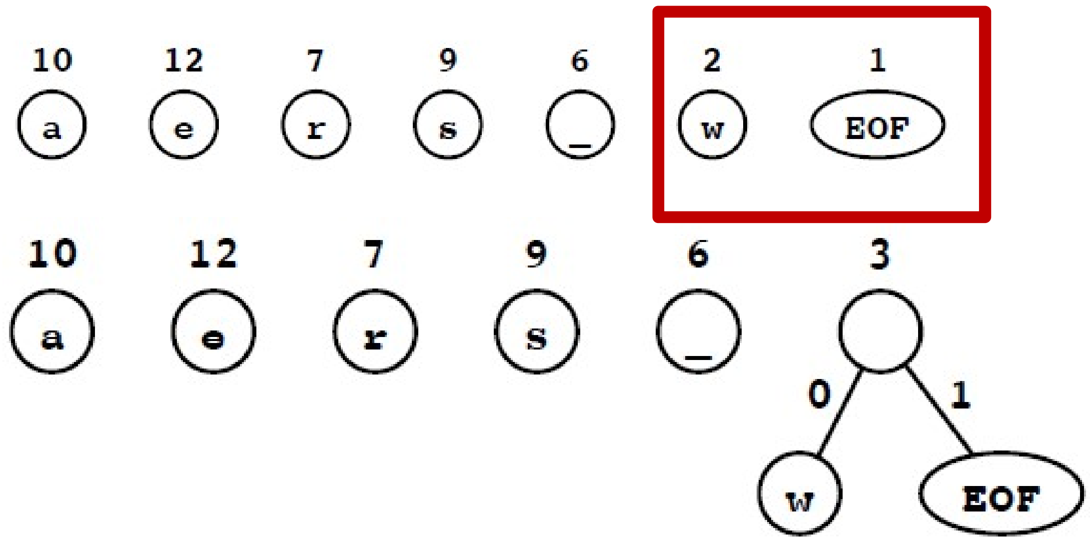
- Initially, each character is a (trivial) Huffman tree by itself



(the numbers above each tree represents frequency count)

Building a Huffman Tree

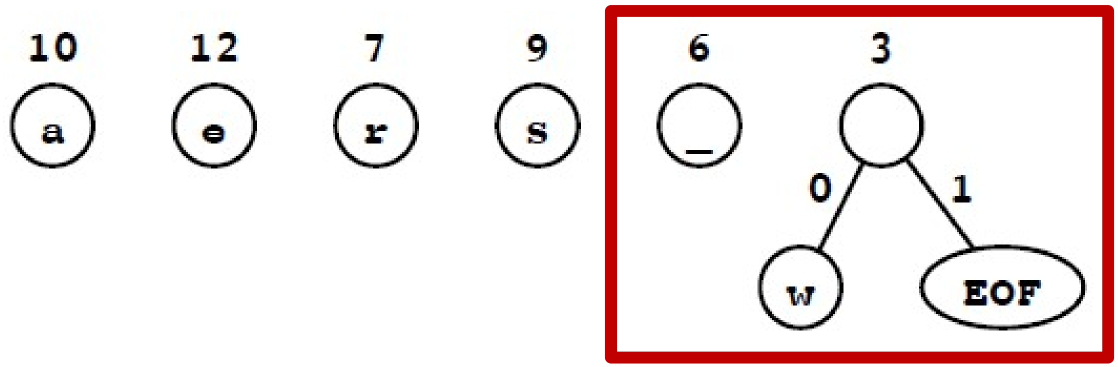
- Pick the 2 trees with lowest frequency counts, and merge their trees
  - → make each tree a child of a new root node
  - → it doesn't matter which tree is the left child and which is the right child

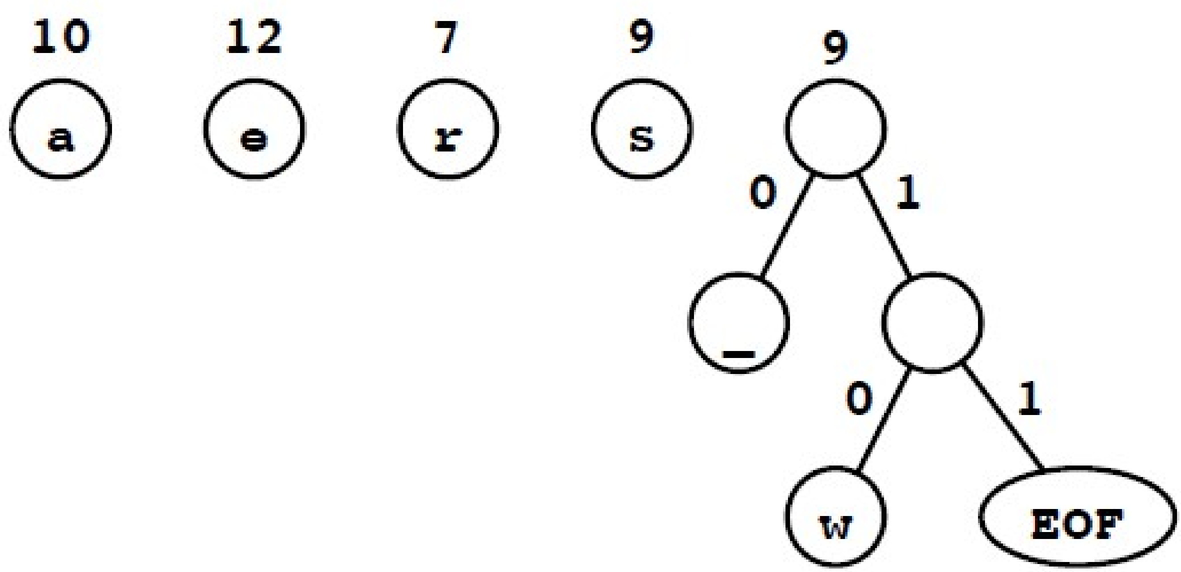


- The number on this new tree is the total frequency count of all leaves

Building a Huffman Tree

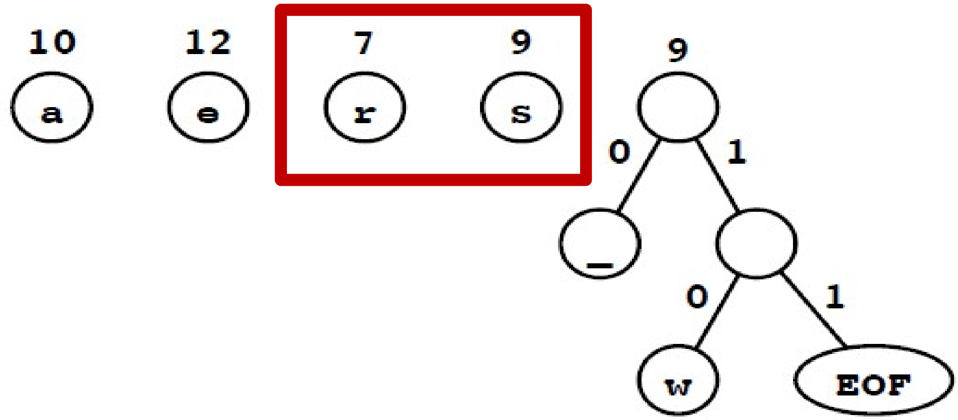
- Repeat: pick the 2 trees with lowest total frequency count, and merge

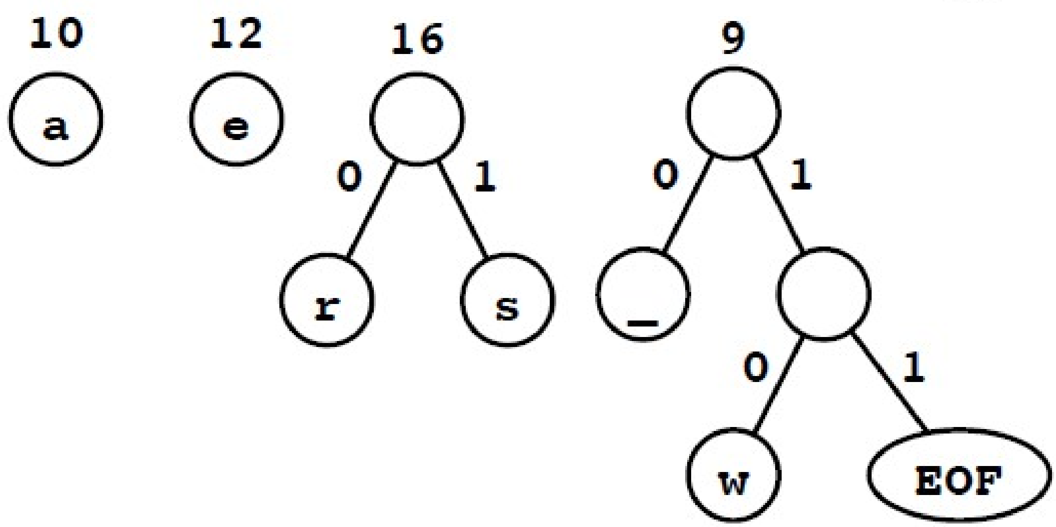




Building a Huffman Tree

- Repeat: pick the 2 trees with lowest total frequency count, and merge

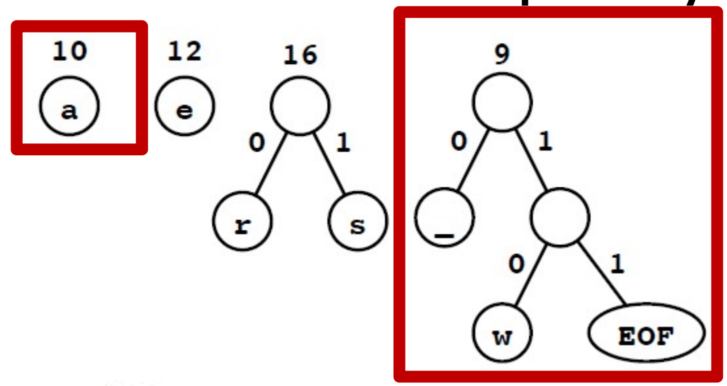


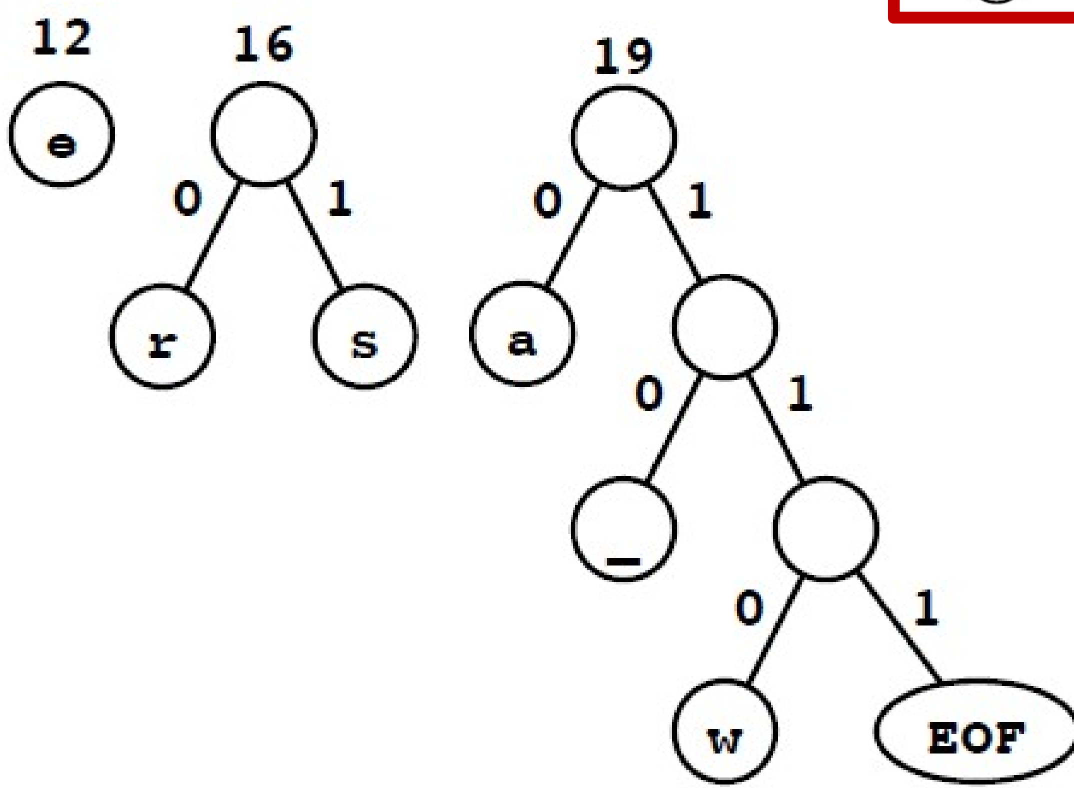


- In case of a tie, it doesn't matter which one you pick

Building a Huffman Tree

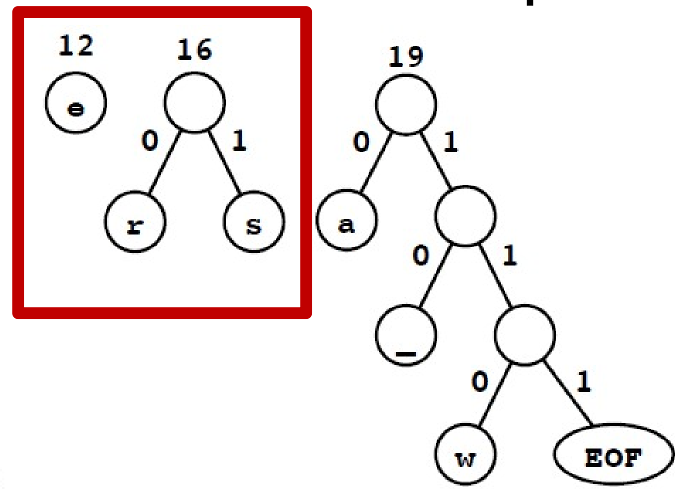
- Repeat: pick the 2 trees with lowest total frequency count, and merge

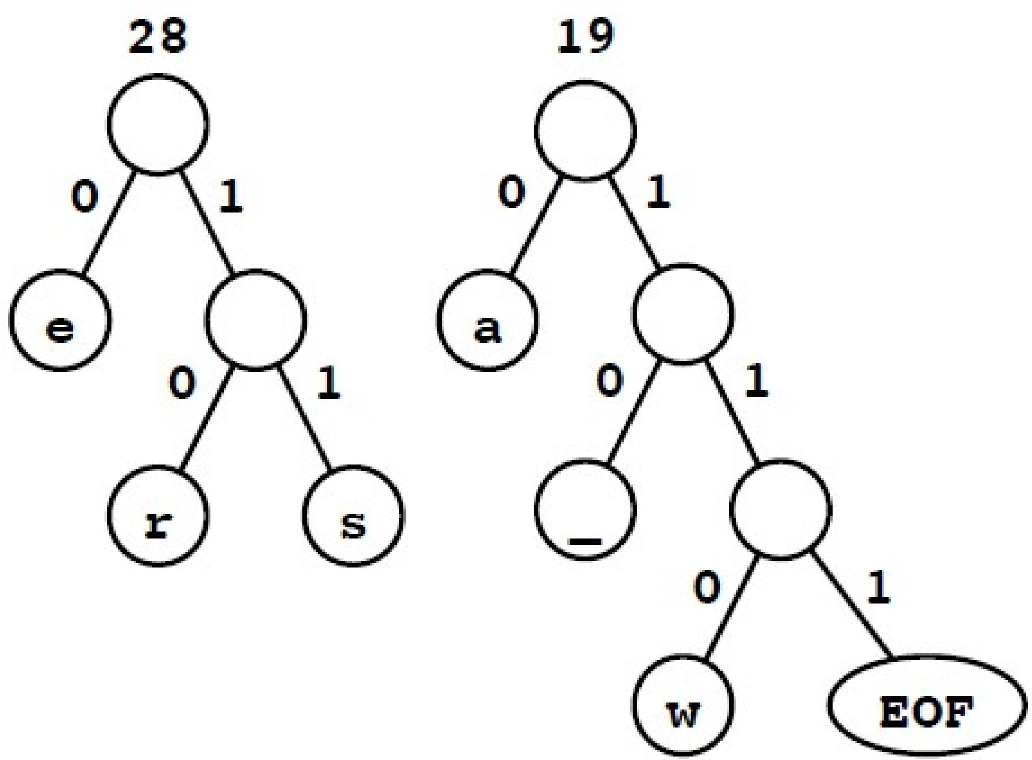




Building a Huffman Tree

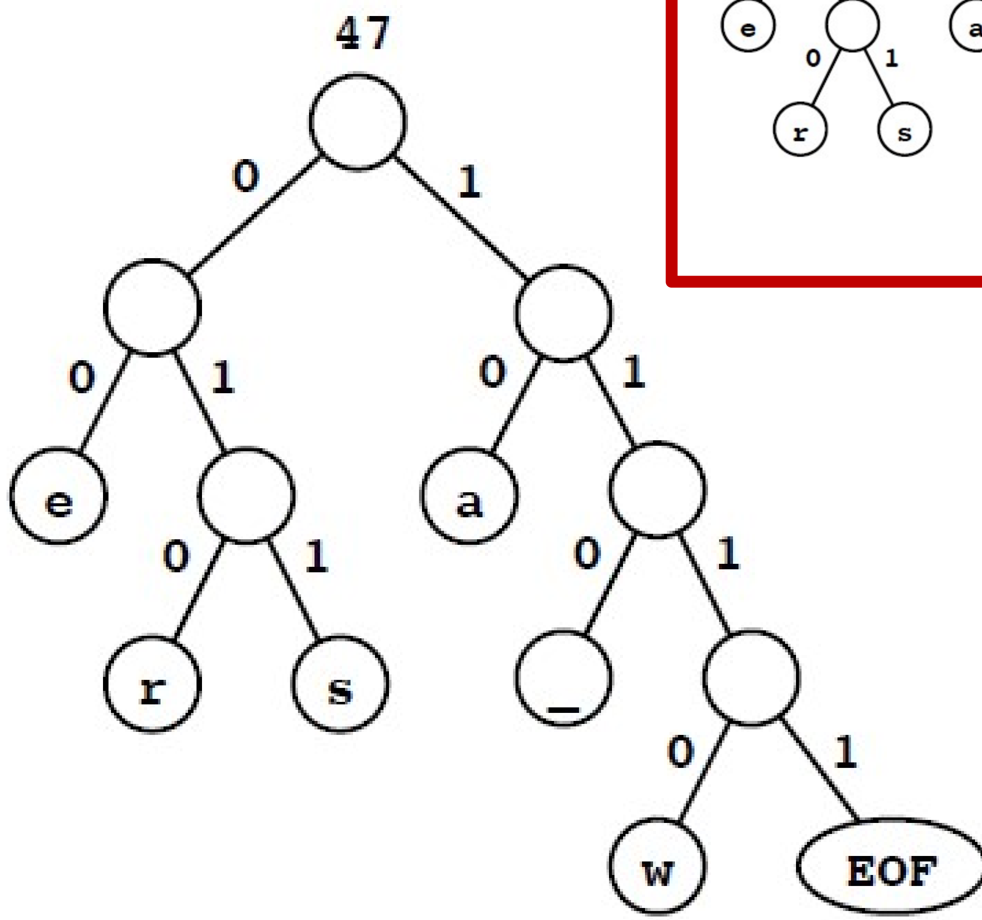
- Repeat: pick the 2 trees with lowest total frequency count, and merge

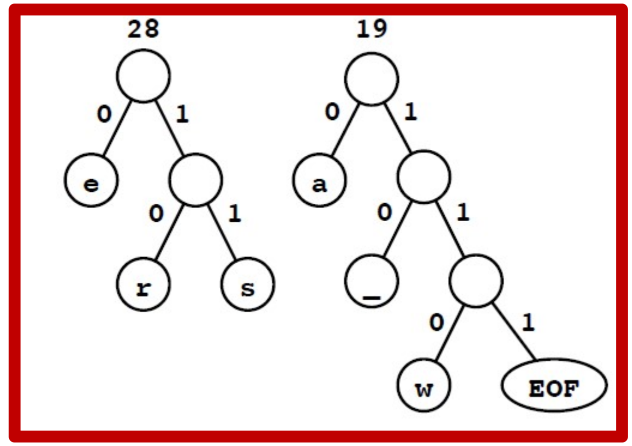




Building a Huffman Tree

- Repeat: Done, here is our Huffman Tree!





Summary: Build Huffman Tree

Algorithm:

- Do a frequency count of all characters in the file (include a count of 1 for the EOF sentinel)
- Initially, each character is single node in a trivial Huffman tree
- The total frequency count of a tree is the sum of frequencies of its leaves
- While there is more than one tree, merge the two with the smallest frequency counts
- Merging trees T_1 and T_2 means creating a new root node and setting T_1 and T_2 as its children

Summary: Compress the File

- For each character, determine its 0/1 compression encoding by looking at the root-to-leaf path
- Output the sequence of 0/1 bits obtained by replacing the character with its compressed bits
- Don’t forget the final sequence for the EOF sentinel

Summary: Decompress the File

- Starting from the root, traverse the Huffman tree. Each bit from the input sequence dictates when to go left or right

- When you reach a leaf, output the character, return to the root and continue traversing the tree according to the next bit(s) in the sequence

- Quit when you reach the EOF leaf

Why Include an EOF?

- Use an EOF sentinel because the last byte of the compressed file might not be “complete.”
  - → e.g. maybe we needed 35 bits in our compression sequence: that’s 4 bytes and 3 bits

- Therefore, decoding EOF tells us when to stop

Considerations

- In order to decompress a file using this approach, we need to know the structure of the Huffman tree used to compress the file in the first place

- When we send the compressed file, need to also send a representation of the Huffman tree used

Final Notes

- Huffman compression exploits frequencies of characters. Fairly simple compression scheme, but it does result in reduced file sizes if that file contains a subset of the 2^8 different bytes (characters)

- Huffman compression tends to work best on plain text files and bitmap images with a small range of colours

- Other compression schemes exploit other patterns, and often target specific file types (pictures, text, etc). Some compressions even allow for some data loss (e.g. with .jpeg files).

  → Fairly advanced topic

- No compression scheme can make <u>every</u> file smaller