Huffman Code

CSCI 232

Data Structures & Algorithms

Huffman Codes

- An algorithm to *compress* data
- Purpose:
  - Apply a compression algorithm to take a large file and store it as a smaller set of data
  - Apply a decompression algorithm to take the smaller compressed data, and get the original back

- Only need to store the smaller compressed version, as long as you have a program to compress/decompress
  - Compression Examples: WinZip, tar

Quick Lesson In Binary

- Generally for an *n*-digit number in binary:
  - $b_{n-1} \ldots b_2 b_1 b_0 = b_{n-1}2^{n-1} + \ldots + b_2 2^2 + b_1 2^1 + b_0 2^0$

- Assume unsigned bytes, convert these:
  - 01011010
  - 10000101
  - 00101001
  - 10011100
  - 11111111

- Everything internally is stored in binary

Characters

- Take one byte of memory (8 bits)
- All files are just sequences of characters
- Internal storage (ASCII Codes):

CHAR	DEC	BINARY
A	65	01000001
B	66	01000010
C	67	01000011
…		
Y	89	01011001
Z	90	01011010

Example File

- ILOVETREES
  - (ASCII: I=73, L=76, O=79, V=86, E=69, T=84, R=82, S=83)

- Internal Storage
  - 01001001 01001100 01001111 01010110 01000101
    01010100 01010010 01000101 01000101 01010011

- All characters take one byte (8 bits) of storage, and those 8 bits correspond to the ASCII values

Underlying Motivation

- Why use the same number of bits to store all characters?
  - E is used much more often than Z
  - So what if we only used two bits to store E
  - And still used the eight to store Z
  - We should save space
- With ILOVETREES, if we change E to 01:
  - We save 6*3 = 18 bits
- For large files, we could save a significant amount of storage space

Variable-Length Code

- When choosing shorter codes, we cannot use any code that is the *prefix* of another code. For example, we could not have:
  - E: 01
  - X: 01011000

- If we come across this in the file:
  - …….010110000
  - We do not know if it is an E followed by something else, or just an X.

Frequency Table

- Computing Huffman Codes first requires computing the frequency of each character, for example "SUSIE SAYS IT IS EASY"

CHAR	COUNT
A	2
E	2
I	3
S	6
T	1
U	1
Y	2
Space	4
Linefeed	1

Computing Huffman Codes

- Huffman Codes are varying bit lengths depending on frequency

CHAR	CODE
A	010
E	1111
I	110
S	10
T	0110
U	01111
Y	1110
Space	00
Linefeed	01110

Coding SUSIE SAYS IT IS EASY

CHAR	CODE
A	010
E	1111
I	110
S	10
T	0110
U	01111
Y	1110
Space	00
Linefeed	01110

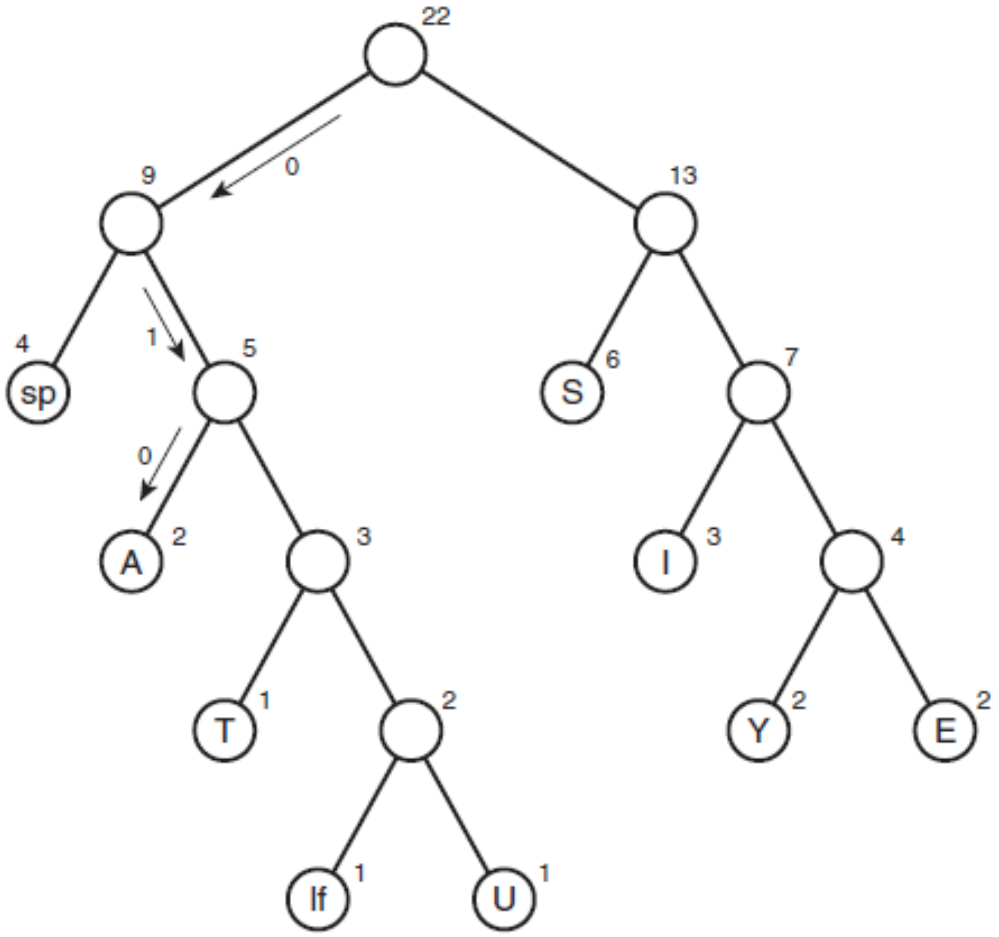
- 10 01111 10 110 1111 00 10 010 1110 10 00 110 0110 00 110 10 00 1111 010 10 1110 01110 (65 bits)
- Before, it would have been 22*8=176 bits

A Huffman Tree

- Each character appears as a leaf in the tree
- The higher the frequency of a character, the higher level in the tree it is
- Number/Key of a leaf is its frequency
- Number/Key of a non-leaf is the sum of all child frequencies

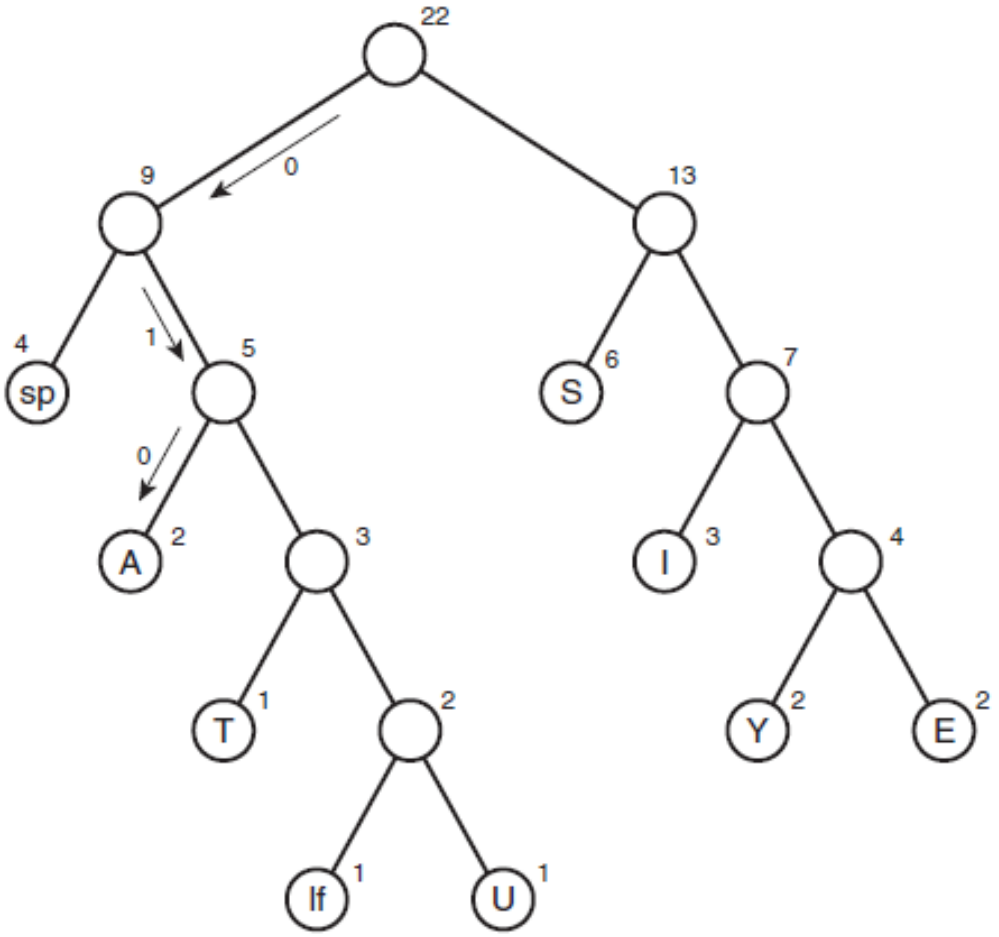
A Huffman Tree

CHAR	COUNT
A	2
E	2
I	3
S	6
T	1
U	1
Y	2
Space	4
Linefeed	1



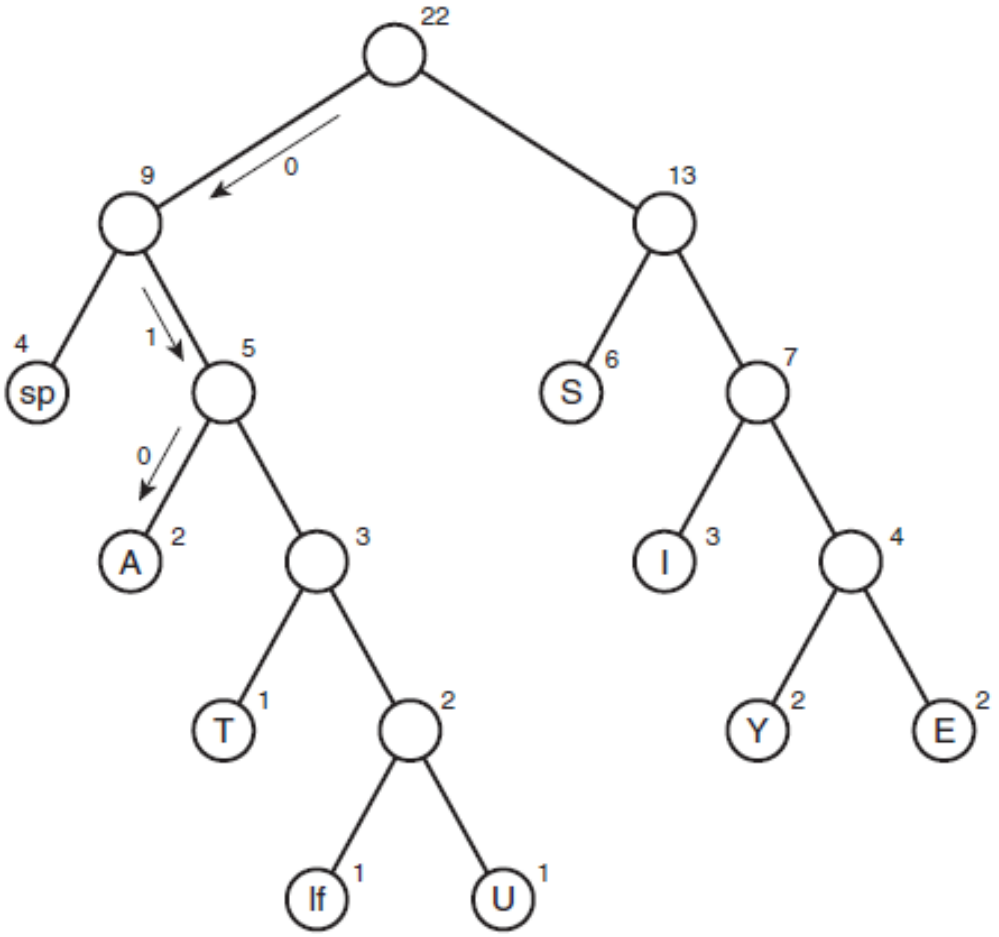
A Huffman Tree

- Decoding
  - For each bit, go right (1) or left (0)
  - Once you hit a character, print it, go back to the root and repeat
- Example: 0100110
  - Go L(0), R(1), L(0), get A
  - Go L(0), R(1), R(1), L(0), get T



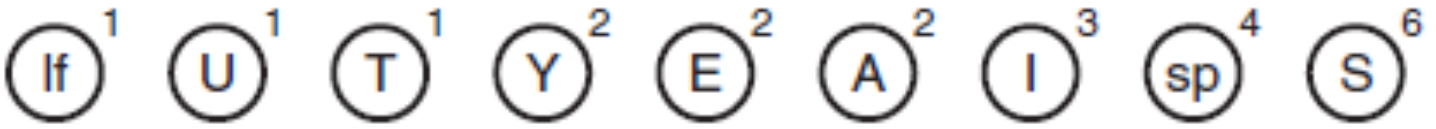
Encoding

- We must produce the tree
- The idea will be to start with just characters and frequencies (the leaves), and then grow the tree



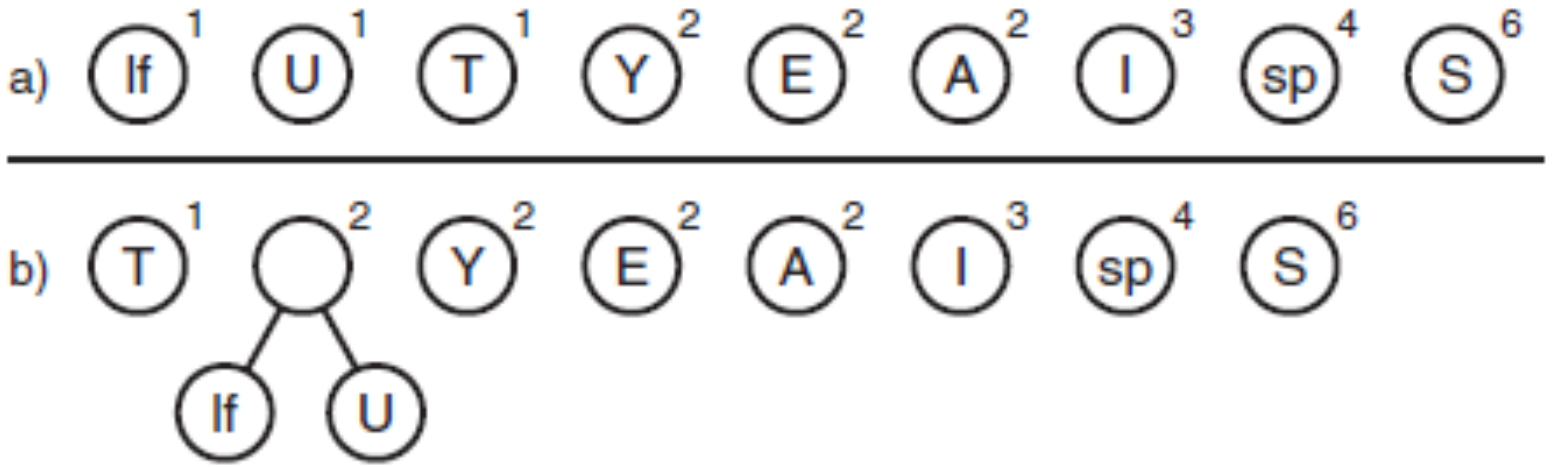
First step

- Start from the leaves, which contain single characters and their associated frequencies
- Store these nodes in a *priority queue*, ordered by frequency



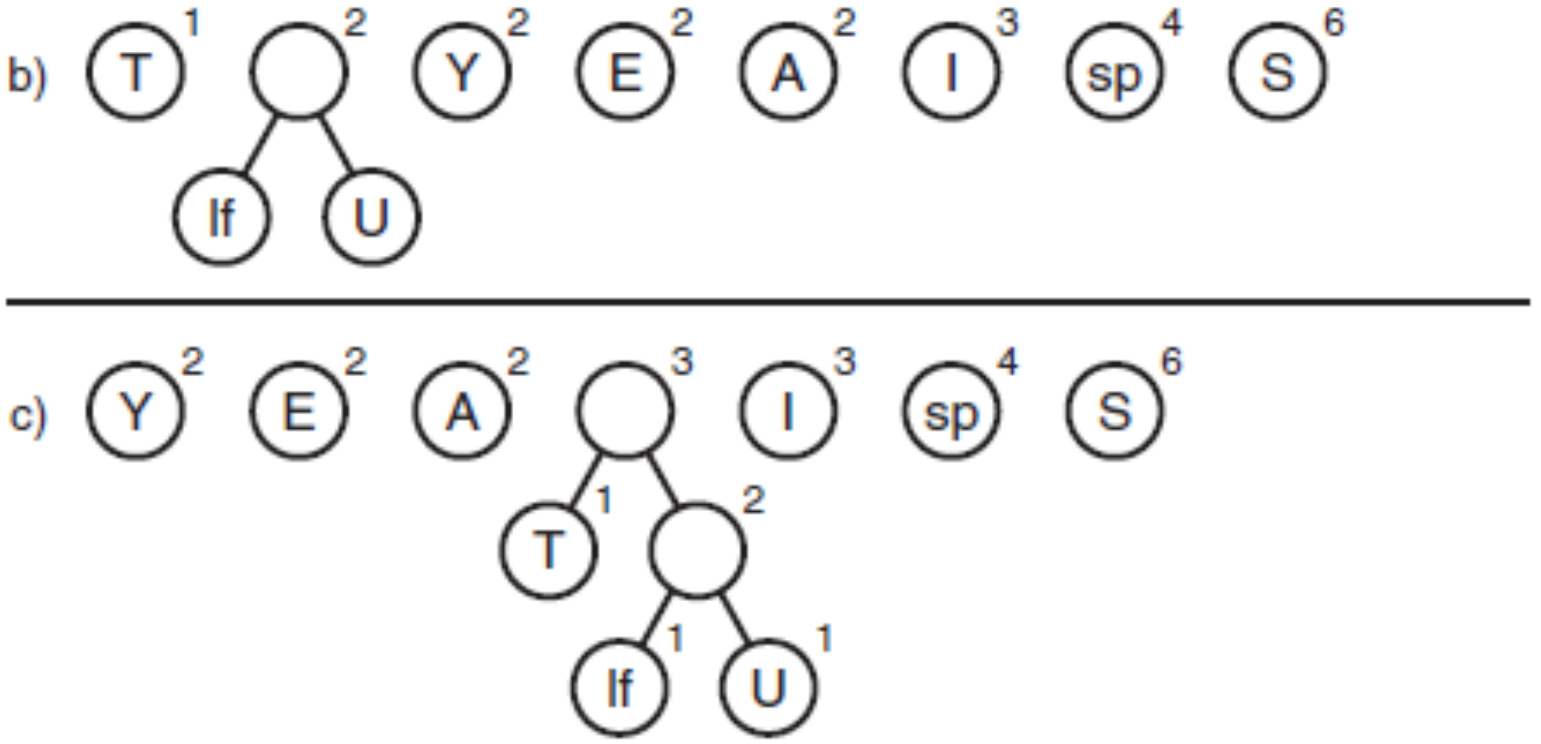
Next

- Take the left two elements, and form a subtree
- The two leaves are the two characters
- The parent is empty, with a frequency as the sum of its two children
- Put this back in the priority queue



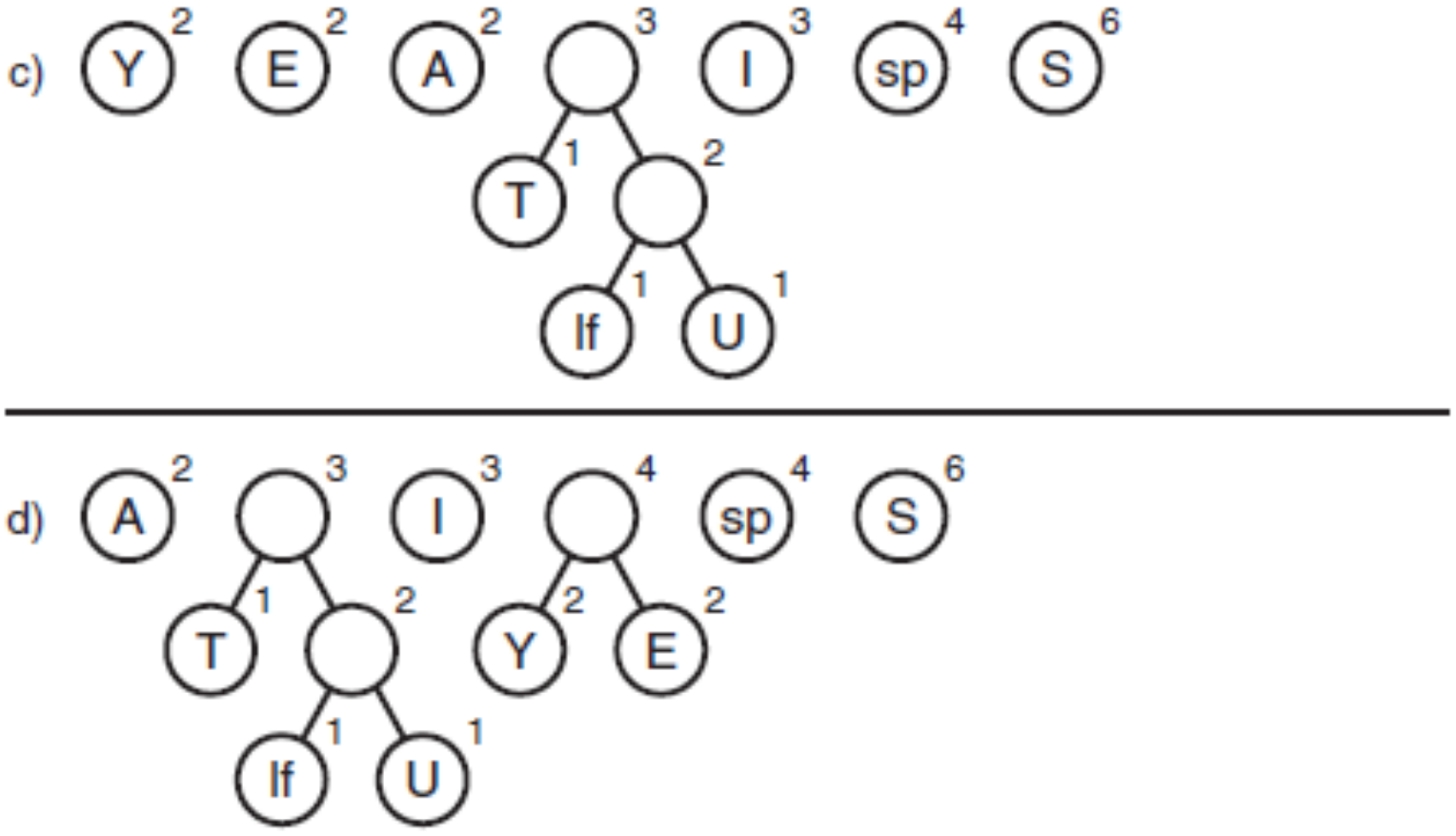
Continue this process...

- Again, adjoin the leftmost two elements (now we actually adjoin a leaf and a *subtree*):

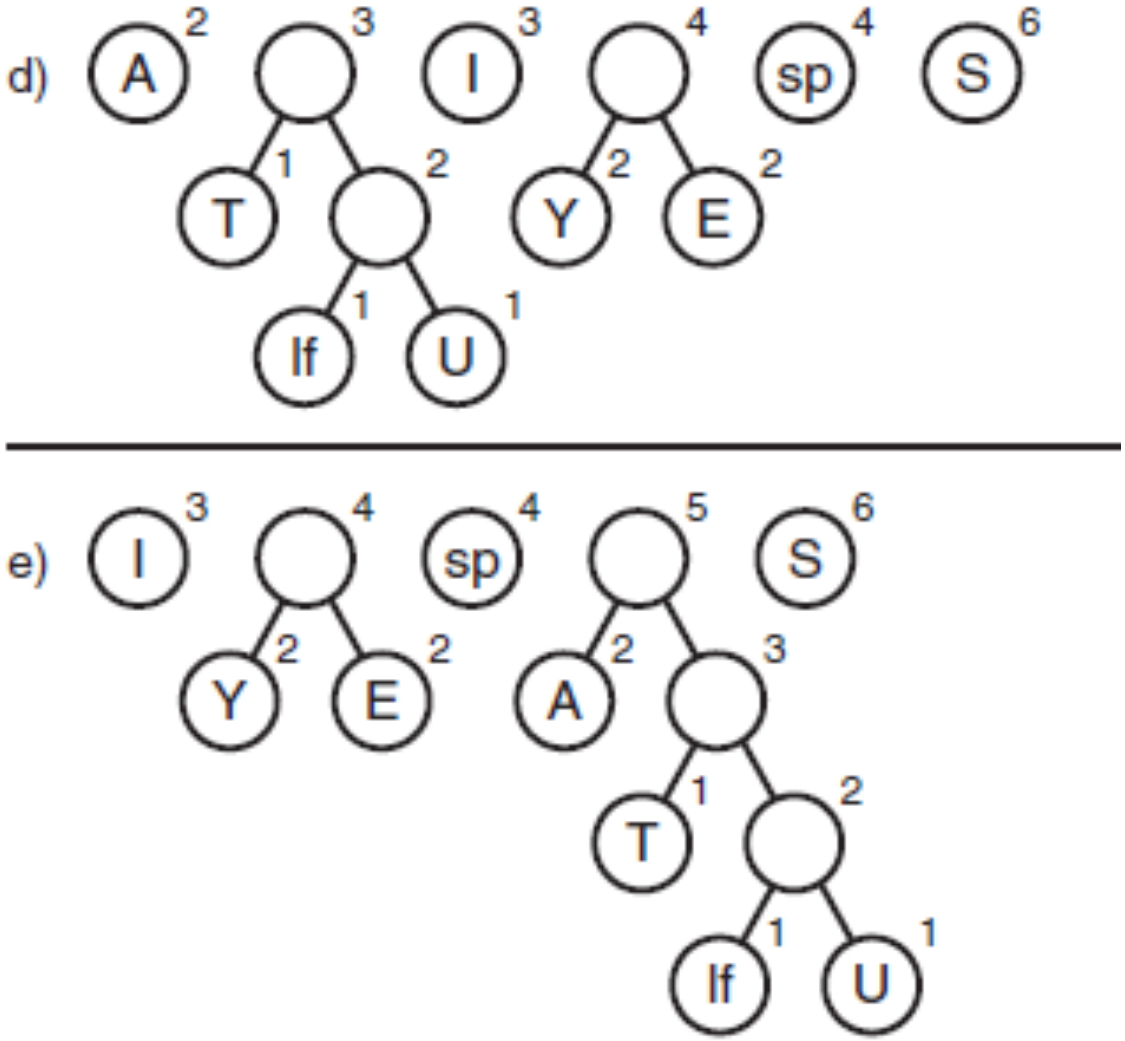


Keep going…

- Adjoin leaves Y (2) and E (2), this forms a subtree with root frequency of 4

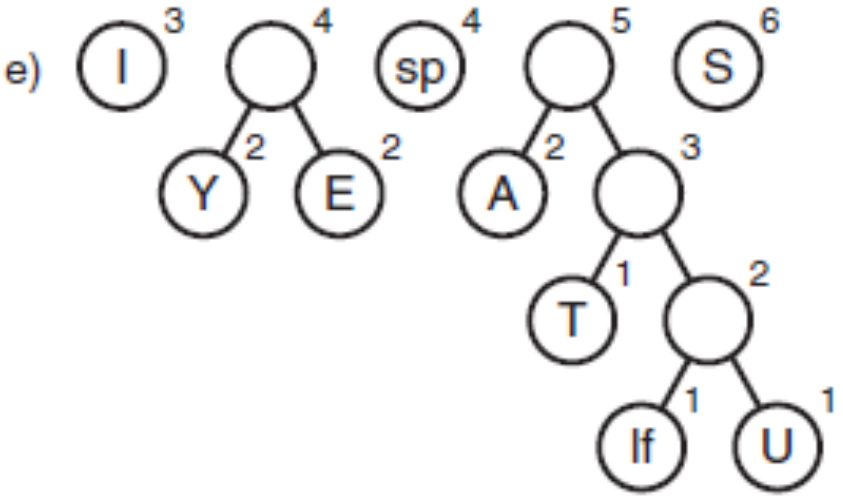


Continue until we have one tree...

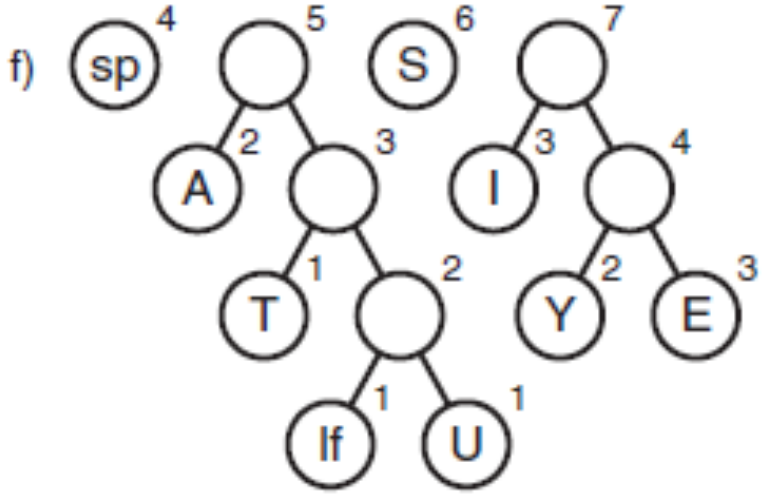


Continue ...

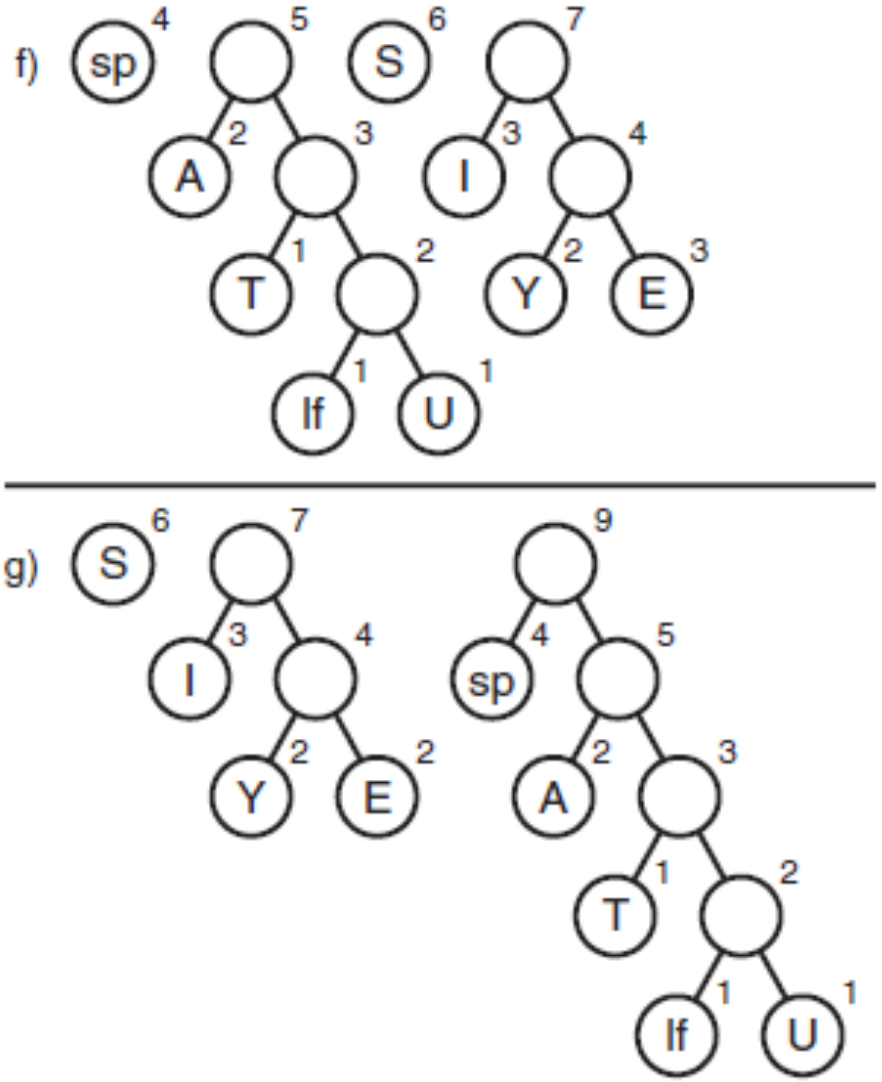
e)


I
3
4
sp
4
5
S
6
2
Y
2
E
2
A
2
3
1
T
2
1
If
1
U
1

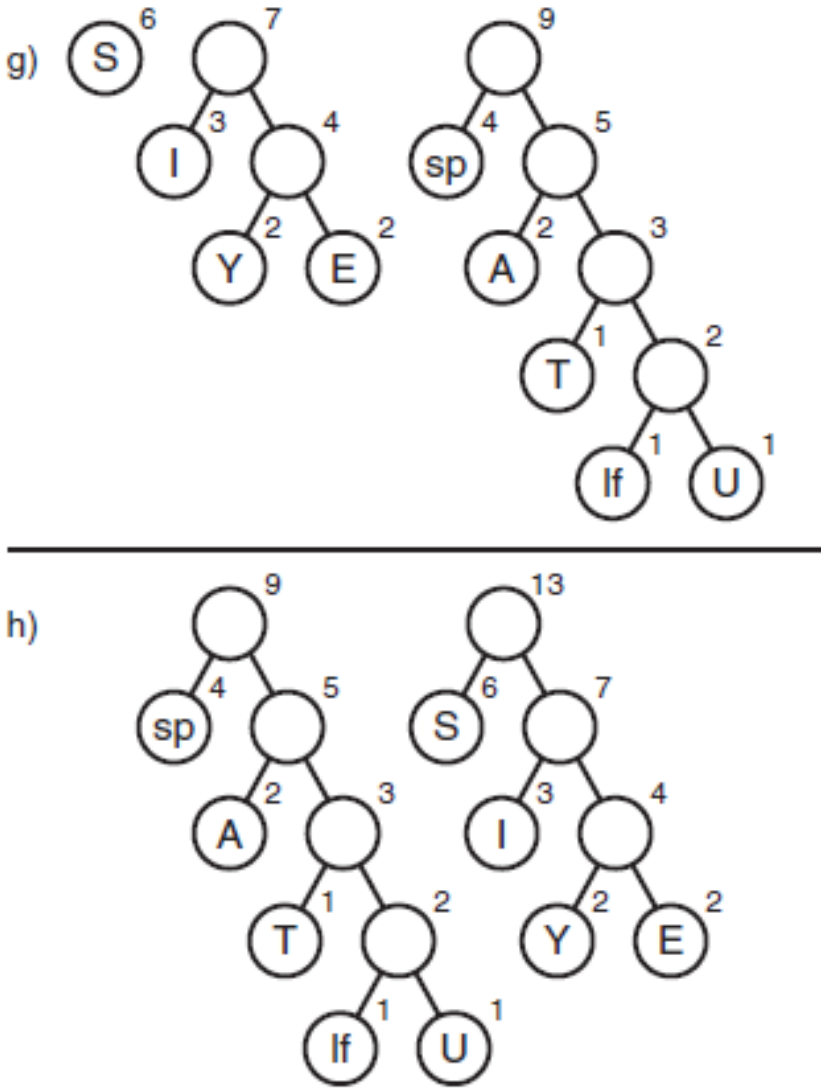
f)


sp
4
5
S
6
7
2
A
3
3
I
4
1
T
2
2
Y
3
E
1
If
1
U
1

Continue ...

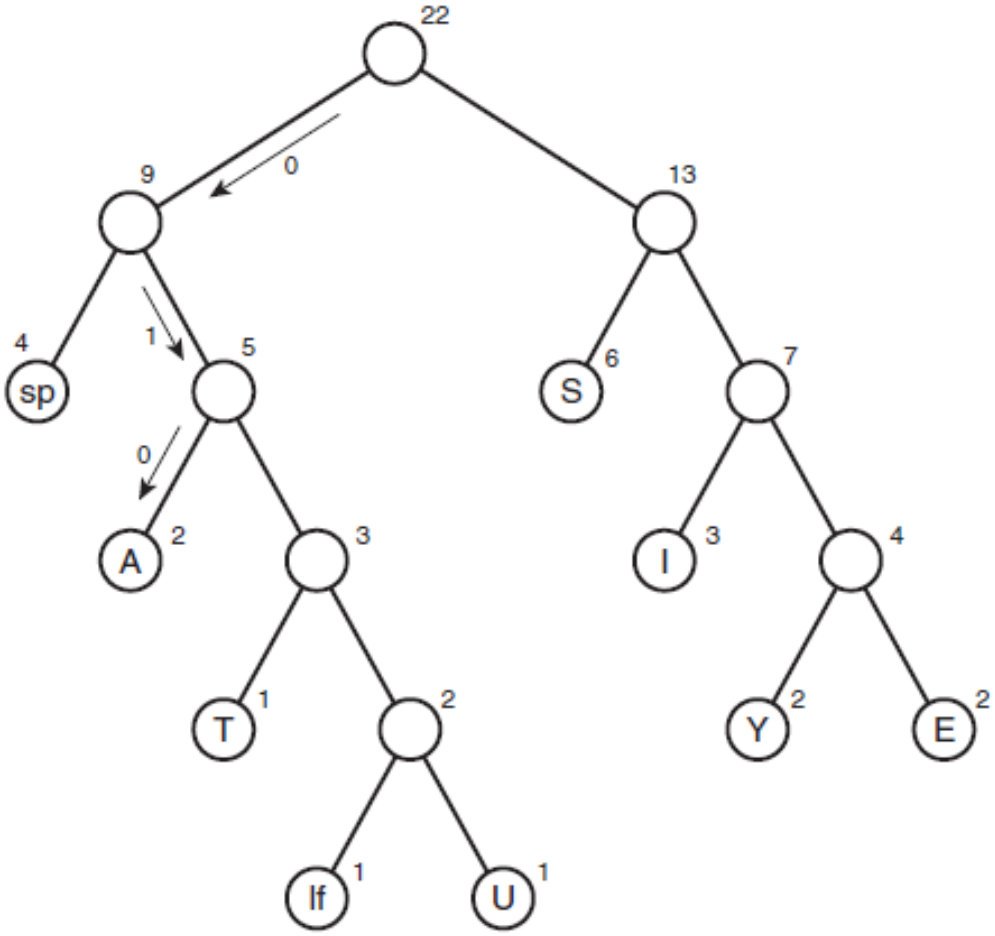


Continue ...



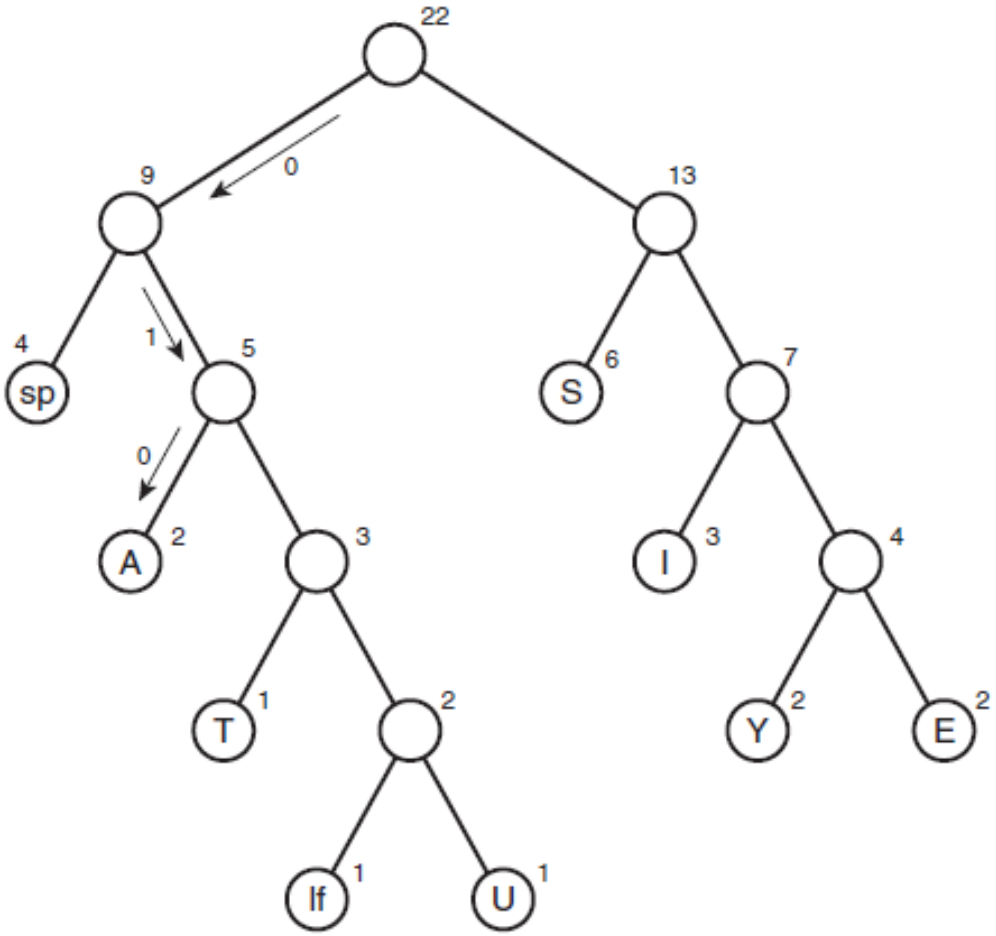
Our final tree

- Construct this from the frequency table



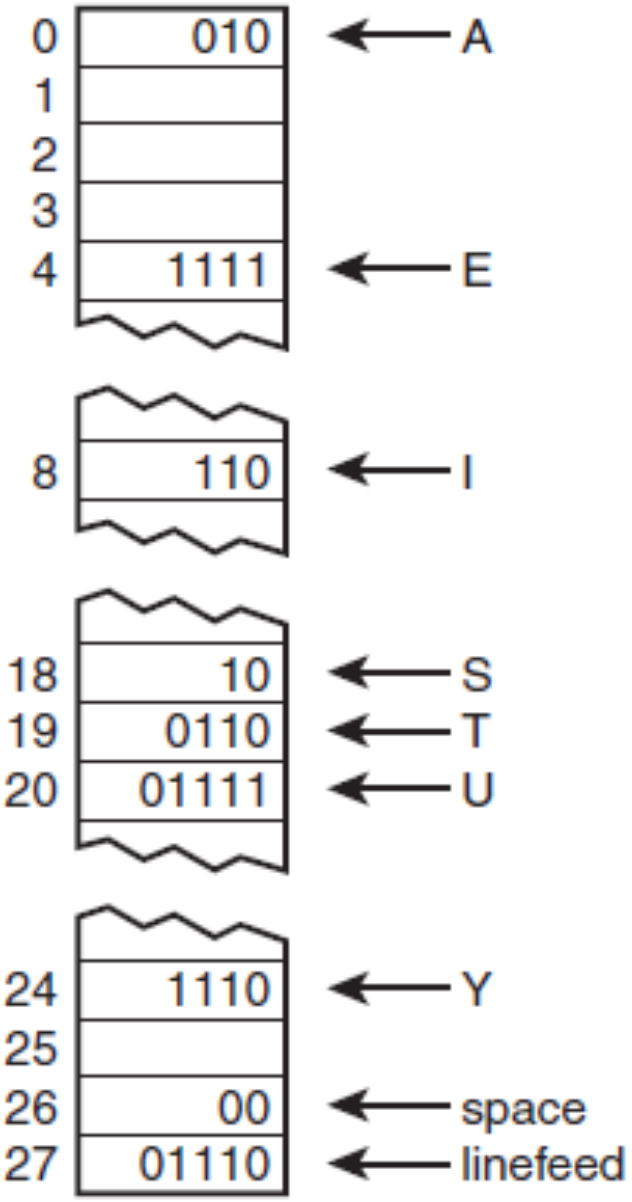
Huffman Code from Tree

- No way around this: we have to start from the root and traverse all possible paths to leaf nodes
  - As we go along, keep track of if we go left (0) or right (1)
  - So A went left (0), then right (1), then left (0) → A= 010



Code Table

- We insert Huffman Code into a Code Table
- Now encoding becomes easy
- For each character, lookup in the code table and store those binary digits
- For decoding, just use the tree



Official Steps

- Encoding a File
  - Obtain a frequency table
  - Make the Huffman Tree
  - Make the Code Table
  - Store each character as its Huffman Code from the Code Table

- Decoding a File
  - Read the compressed file bit-by-bit
  - Use the Huffman Tree to get each character

Self-Testing

- Consider the phrase “dogs do not spot hot pots or cats”
- Write down the frequency array of the letters
- Draw the corresponding Huffman tree
- Each letter should have its own code including the spaces