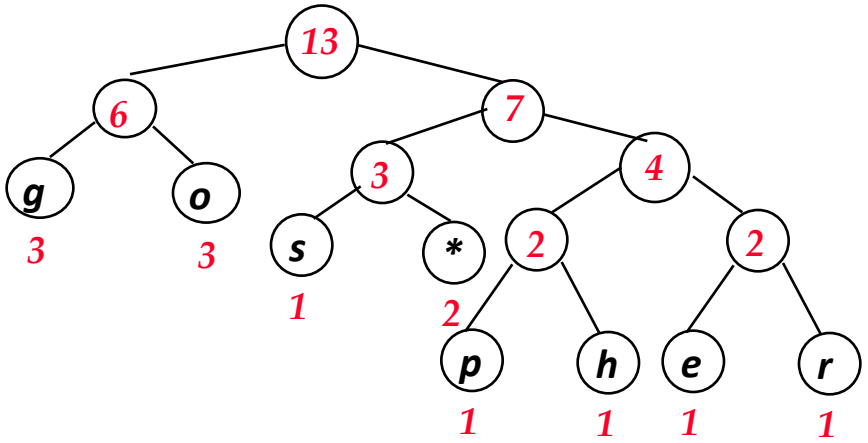
# CompSci 100e
# Program Design and Analysis II




April 7, 2011

Prof. Rodger

# Announcements

- Boggle due today
- APT Wordladder – due Tuesday
  - Lab this week – work on Wordladder
- Test 2 is April 14
  - Will review next time
  - Try writing answers for Test 2 CompSci 100 Fall 2010

# Compression and Coding

- What gets compressed?
  - Save on storage, why is this a good idea?
  - Save on data transmission, how and why?

- What is information, how is it compressible?
  - Exploit redundancy, without that, hard to compress
- Represent information: code (Morse cf. Huffman)
  - Dots and dashes or 0's and 1's
  - How to construct code?

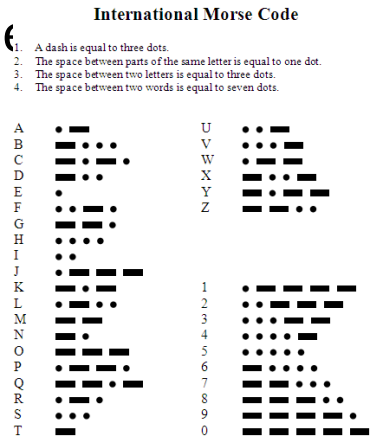
# Huffman Coding

- D.A Huffman in early 1950's: story of invention
  - Analyze and process data before compression
  - Not developed to compress data "on-the-fly"
- Represent data using variable length codes
  - Each letter/chunk assigned a codeword/bitstring
  - Codeword for letter/chunk is produced by traversing the Huffman tree
  - Property: *No codeword produced is the prefix of another*
  - Frequent letters/chunk have short encoding, while those that appear rarely have longer ones
- Huffman coding is optimal *per-character* coding method

# Coding/Compression/Concepts

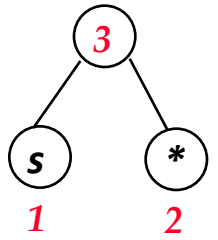
- For ASCII we use 8 bits, for Unicode 16 bits
  - Minimum number of bits to represent N values?
  - Representation of genomic data (a, c ,g, t)?
  - What about noisy genomic data?

- We can use a variable-length encoding, e.g., Huffman
  - How do we decide on lengths? How do we decode
  - Values for Morse code encodings, why?
  - — … - - - …

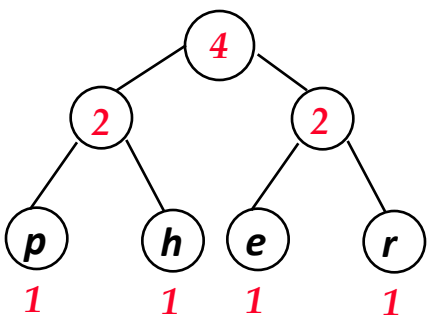


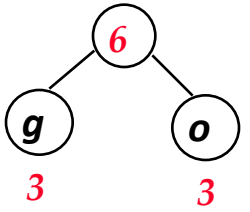
# Huffman coding: *go go gophers*

| | ASCII | | 3 bits |
|---|---|---|---|
| g | 103 | *1100111* | 000 ?? |
| o | 111 | *1101111* | 001 ?? |
| p | 112 | *1110000* | 010 |
| h | 104 | *1101000* | 011 |
| e | 101 | *1100101* | 100 |
| r | 114 | *1110010* | 101 |
| s | 115 | *1110011* | 110 |
| sp. | 32 | *1000000* | 111 |



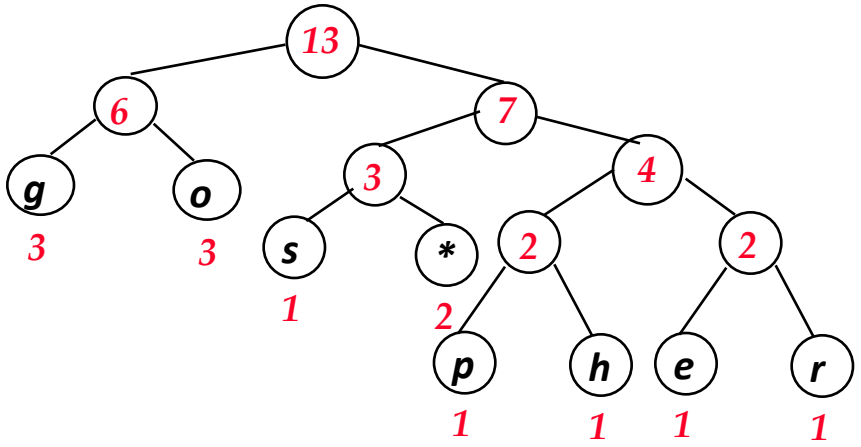

- choose two smallest weights
  - combine nodes + weights
  - Repeat
  - Priority queue?
- Encoding uses tree:
  - 0 left/1 right
  - How many bits?



# Huffman coding: *go go gophers*

| | ASCII | | 3 bits | |
|---|---|---|---|---|
| g | 103 | *1100111* | 000 | 00 |
| o | 111 | *1101111* | 001 | 01 |
| p | 112 | *1110000* | 010 | 1100 |
| h | 104 | *1101000* | 011 | 1101 |
| e | 101 | *1100101* | 100 | 1110 |
| r | 114 | *1110010* | 101 | 1111 |
| s | 115 | *1110011* | 110 | 100 |
| sp. | 32 | *1000000* | 111 | 101 |

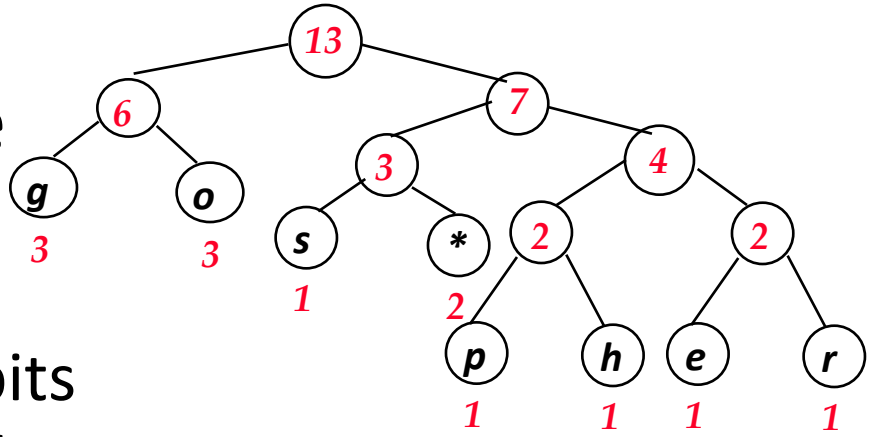



- Encoding uses tree/trie:
  - 0 left/1 right
  - "g" code is
    - left left
    - 00
  - "p" code is
    - right right left left
    - 1100

# Compress to bits – “go go gophers”

- 13 characters total
- 3 bits/char is 39 bits
- 8 bits/char is 104 bits
- Huff: characters that appear more often have shorter codes

| | ASCII | | 3 bits |
|---|---|---|---|
| g | 103 *1100111* | 000 | 00 |
| o | 111 *1101111* | 001 | 01 |
| p | 112 *1110000* | 010 | 1100 |
| h | 104 *1101000* | 011 | 1101 |
| e | 101 *1100101* | 100 | 1110 |
| r | 114 *1110010* | 101 | 1111 |
| s | 115 *1110011* | 110 | 100 |
| sp. | 32 *1000000* | 111 | 101 |




- Huffman coding is 37 bits
- Variable length of bits/char
- “go go gophers”
- 00011010001101000111001101111101111100

# Building a Huffman tree

- Begin with a forest of single-node trees/tries (leaves)
  - Each node/tree/leaf is weighted with character count
  - Node stores two values: character and count

- Repeat until there is only one node left: root of tree
  - Remove two minimally weighted trees from forest
  - Create new tree/internal node with minimal trees as children,
    - Weight is sum of children's weight (no char)
- How does process terminate? Finding minimum?
  - Remove minimal trees, hummm……

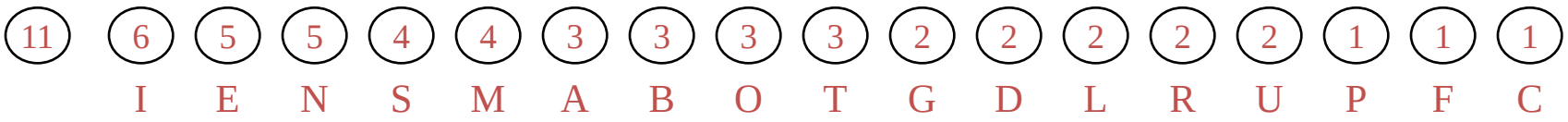
# How do we create Huffman Tree/Trie?

- Insert weighted values into priority queue
  - What are initial weights? Why?
  -
- Remove minimal nodes, weight by sums, re-insert
  - Total number of nodes?

```
PriorityQueue<TreeNode> pq = new PriorityQueue<TreeNode>();
for(int k=0; k < freq.length; k++){
      pq.add(new TreeNode(k,freq[k],null,null));
}
while (pq.size() > 1){
   TreeNode left = pq.remove();
   TreeNode right = pq.remove();
   pq.add(new TreeNode(0,left.weight+right.weight,
                       left,right));
}
TreeNode root = pq.remove();
```

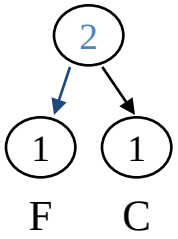
# Building a tree

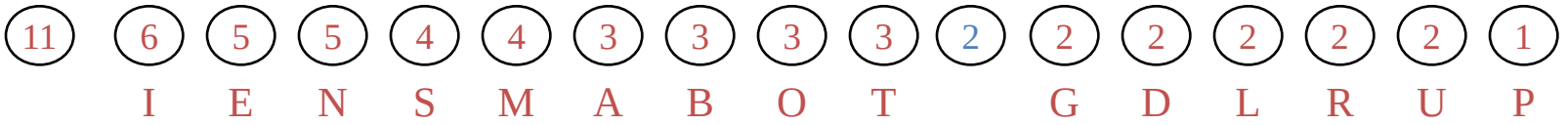
“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”



# Building a tree

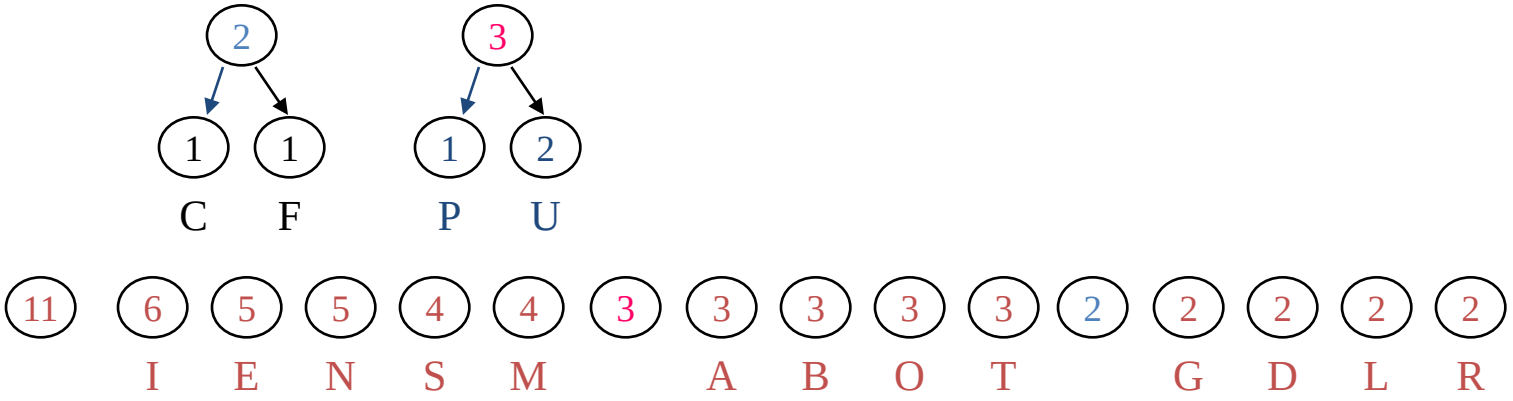
“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”



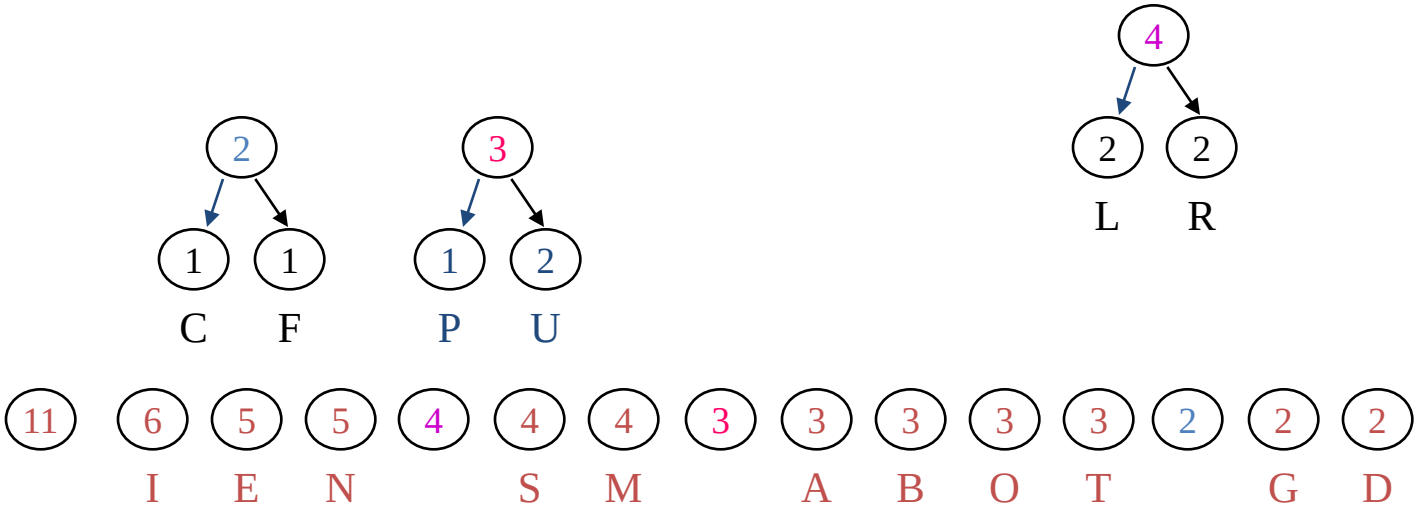
# Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”



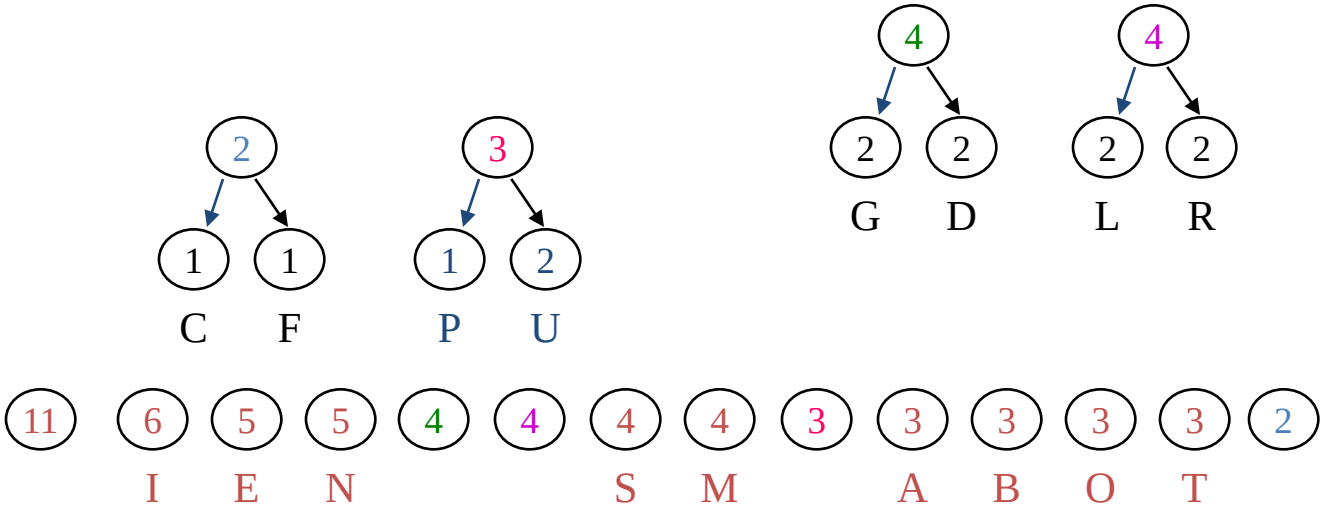
# Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”



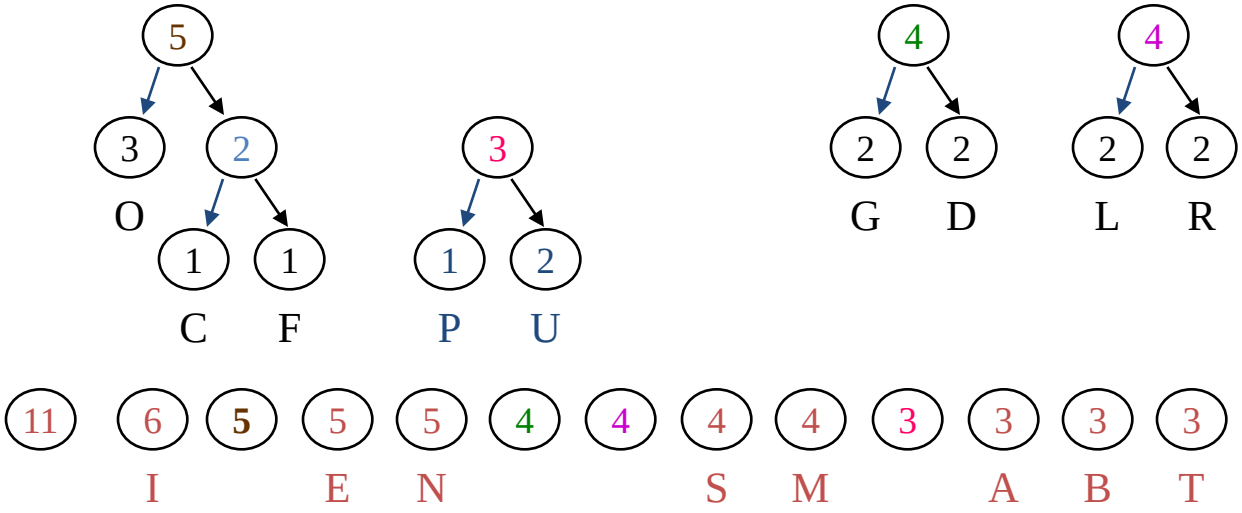
# Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”



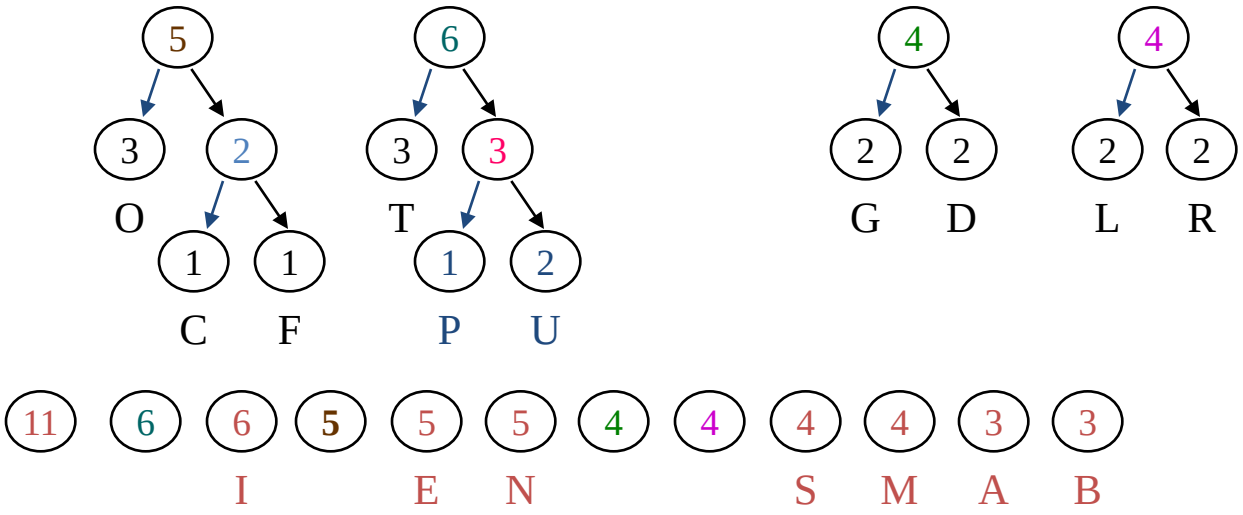
# Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”



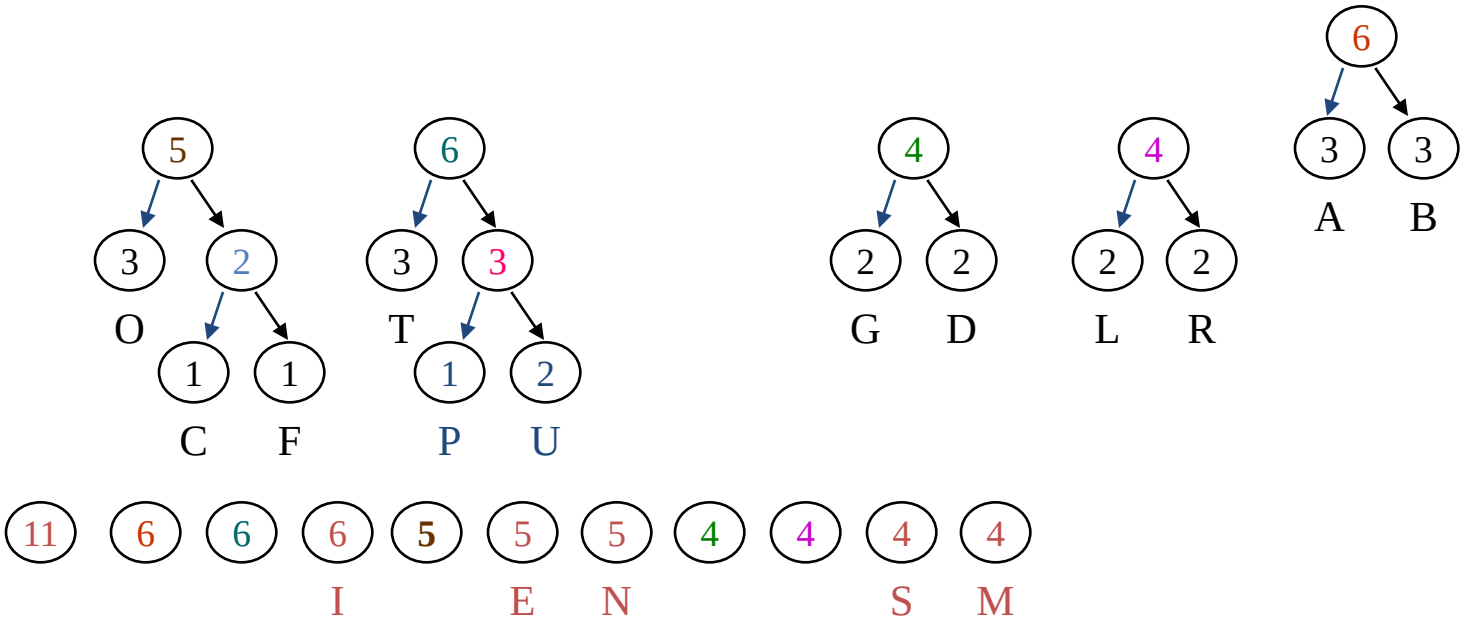
# Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”



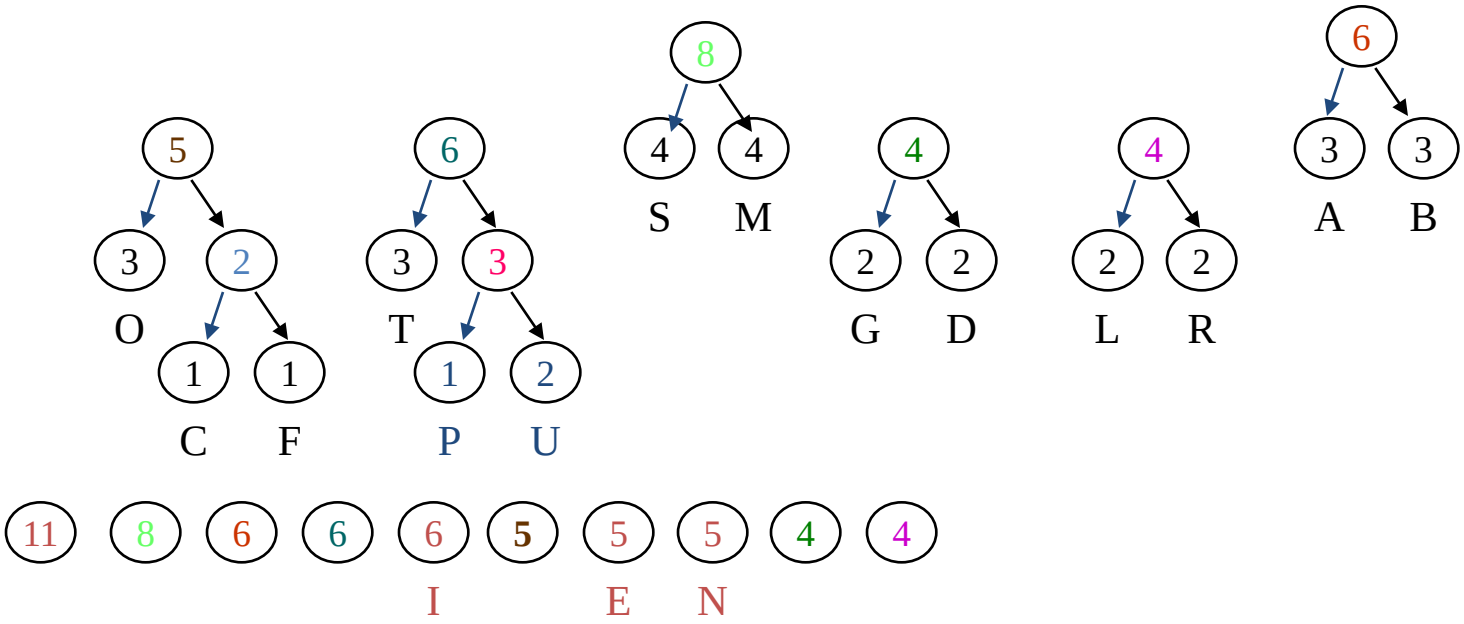
# Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”



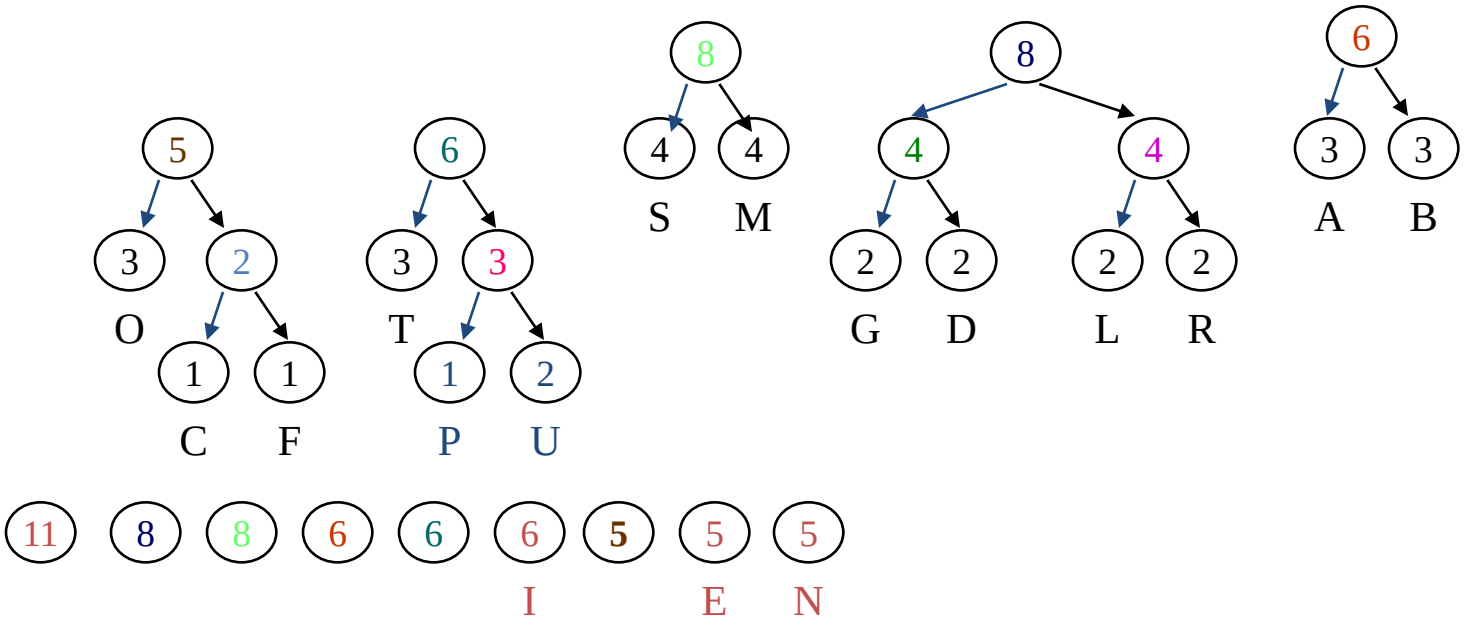
# Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”



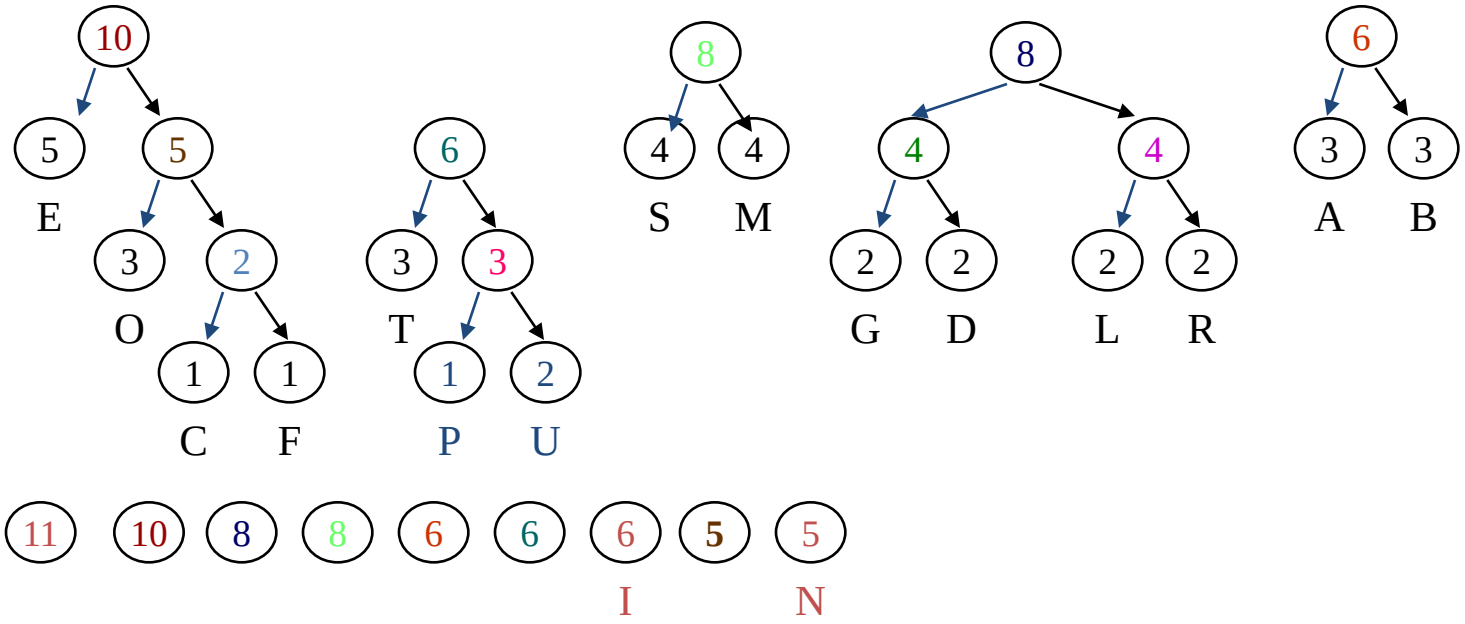
# Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”



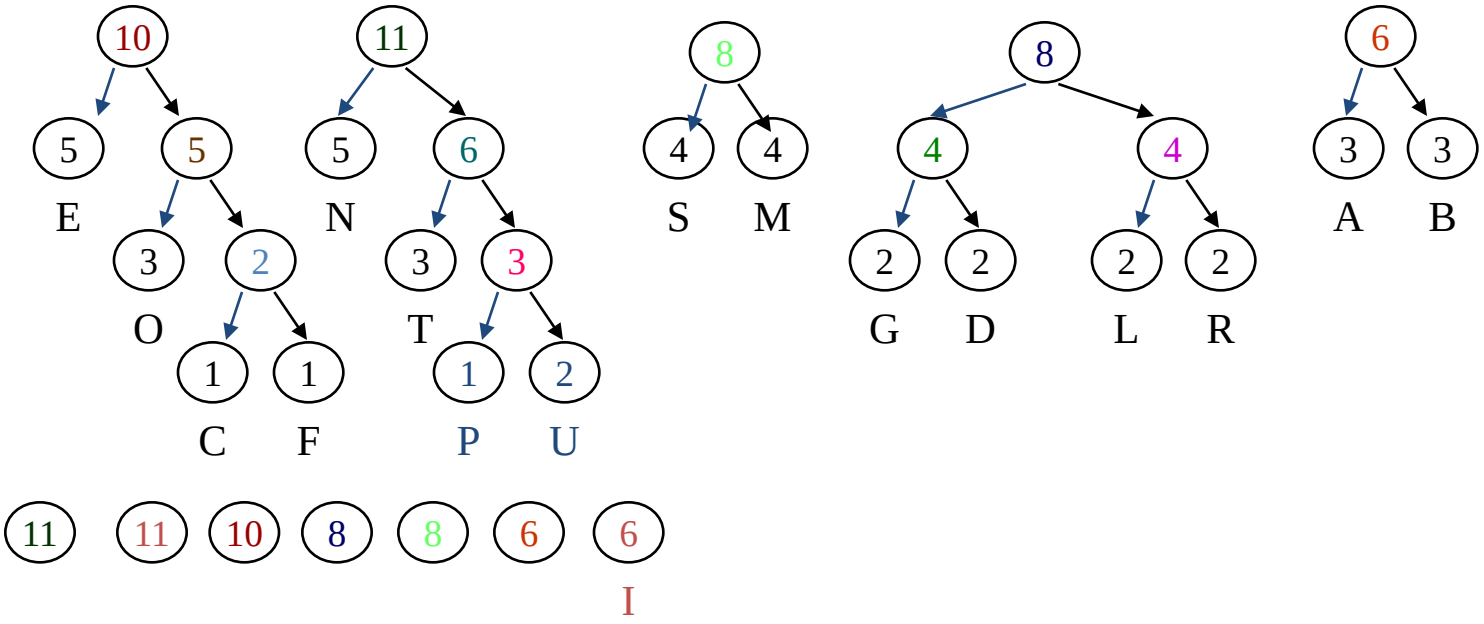
# Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”



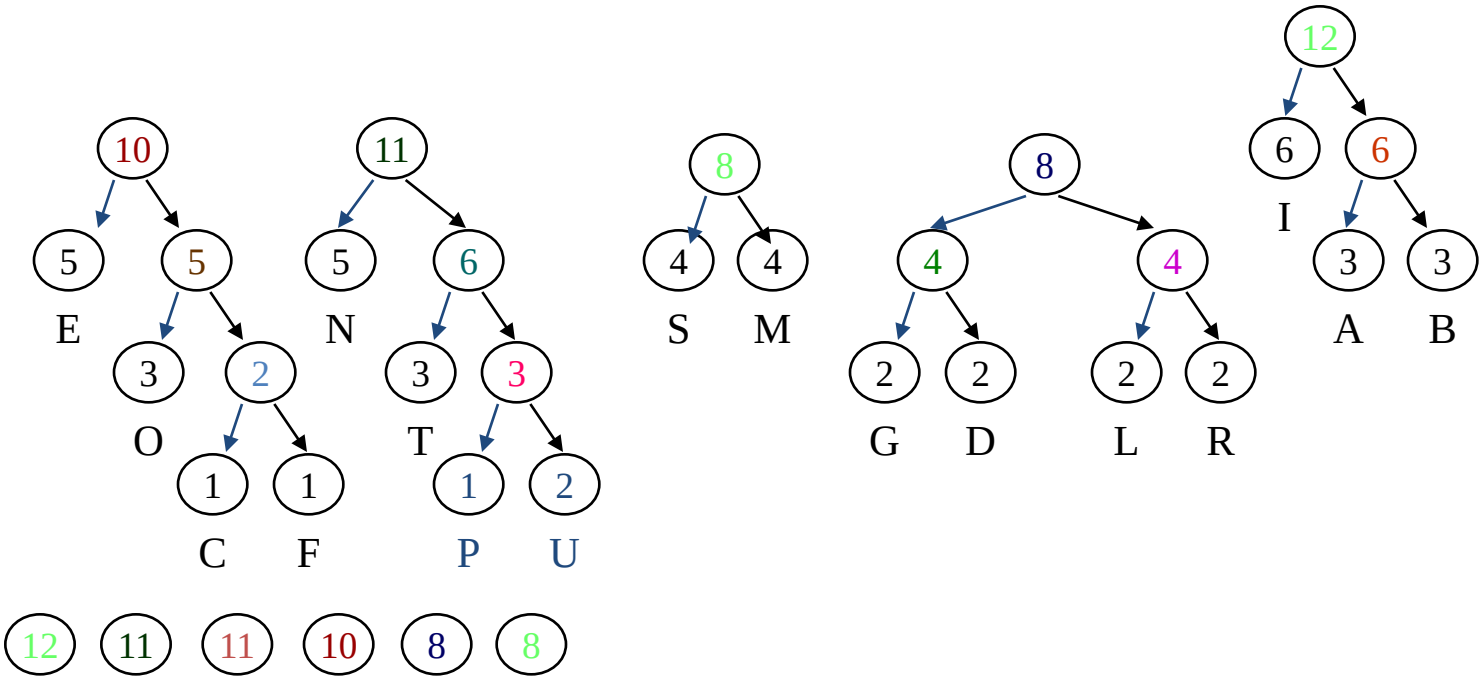
# Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”



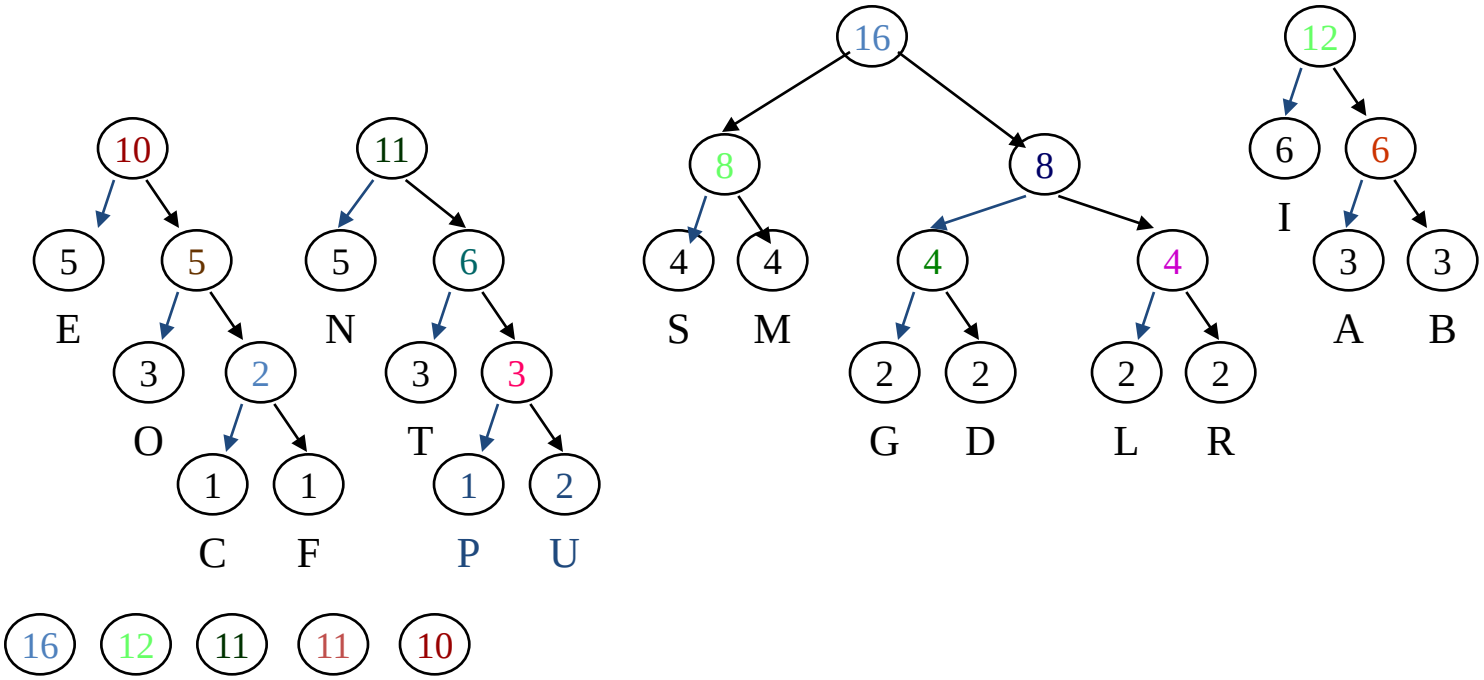
# Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”



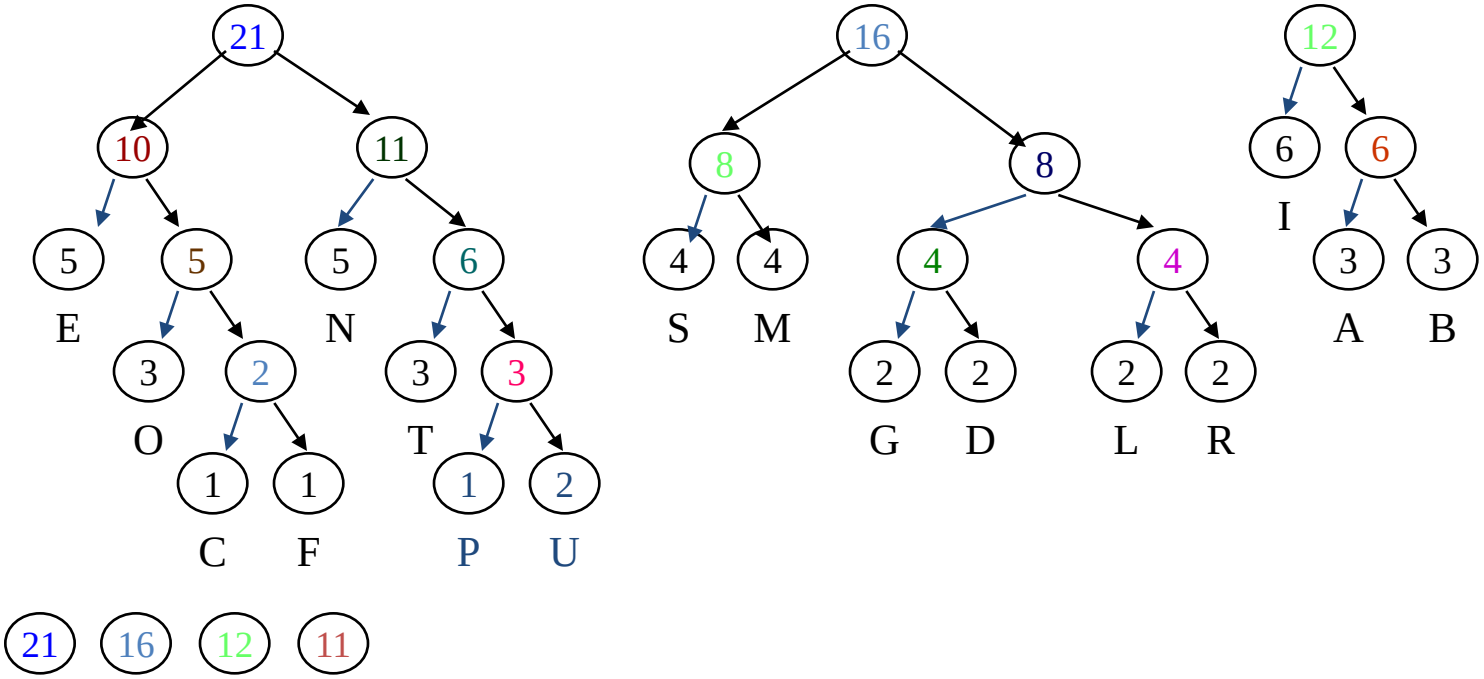
# Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”



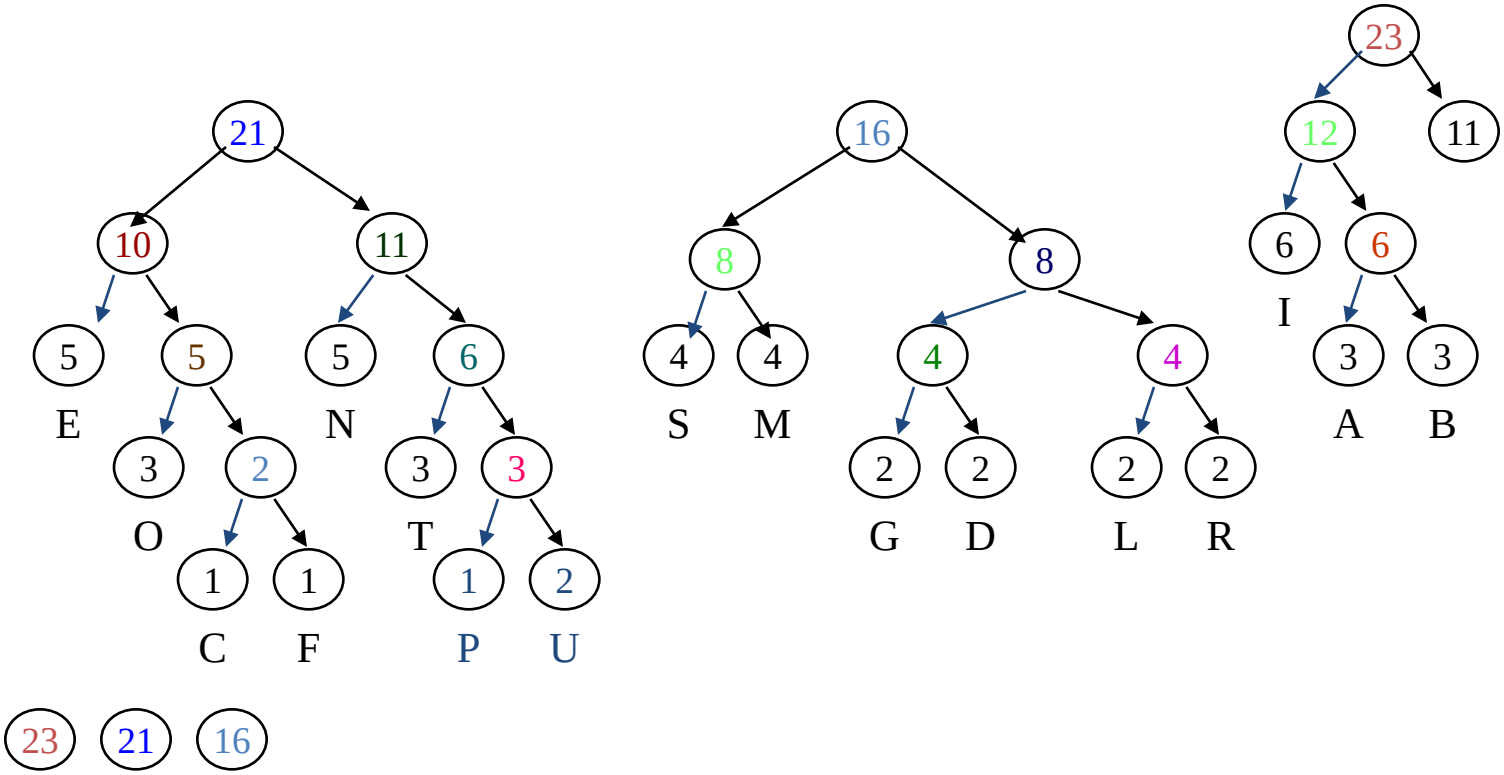
# Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”



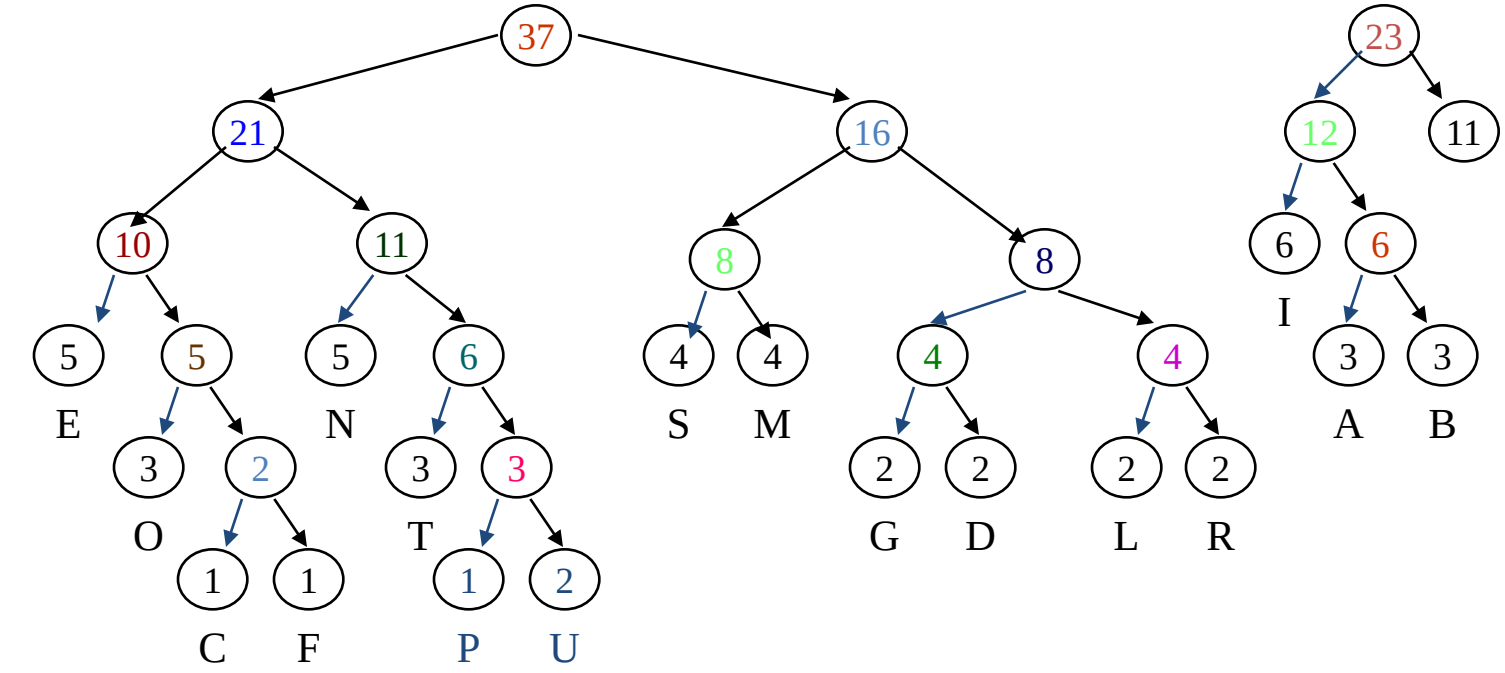
# Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”



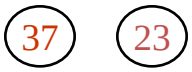
# Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”



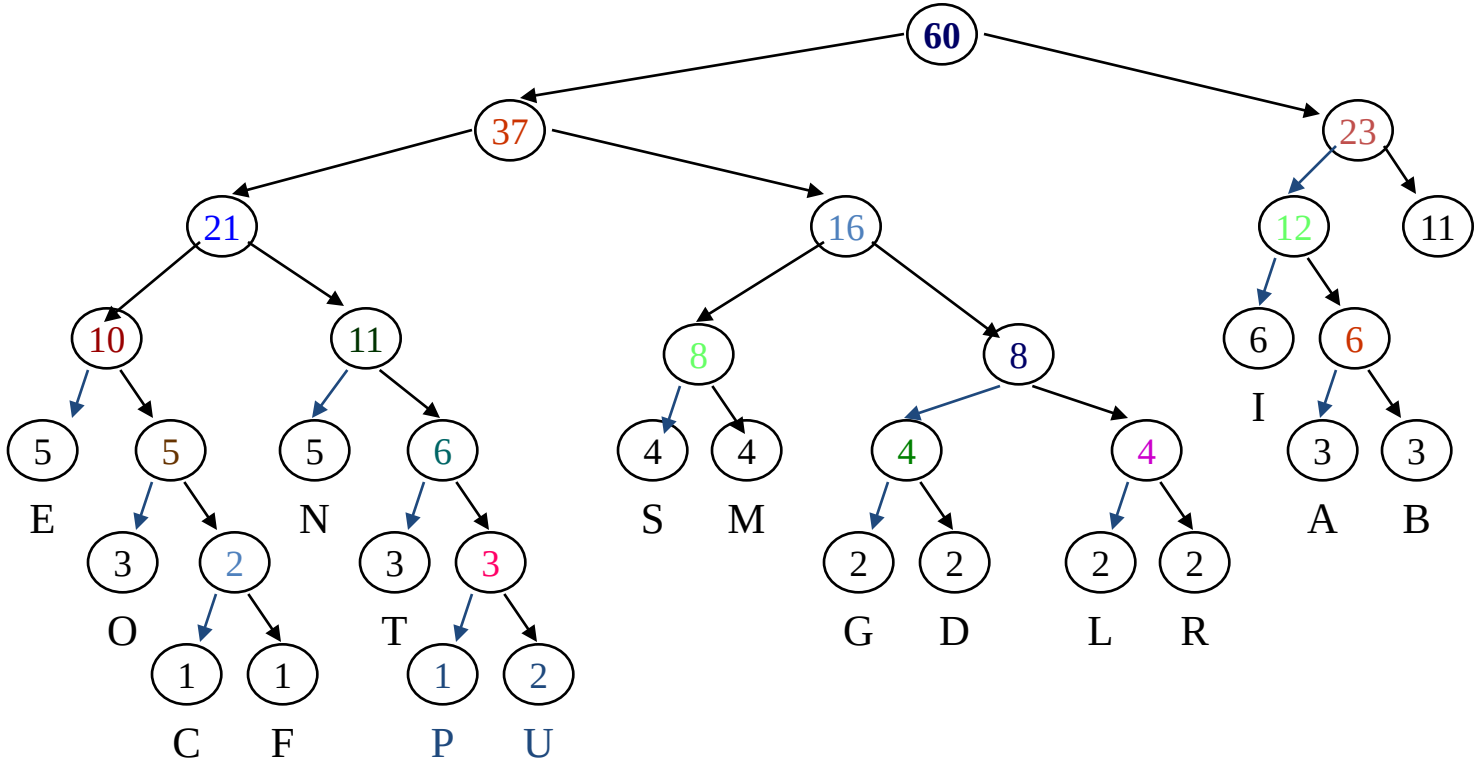






# Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”




60

# Creating compressed file

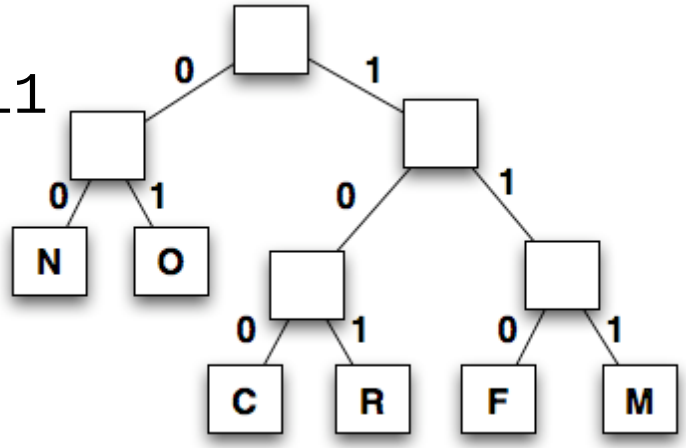
- Once we have new encodings, read every character
  - Write encoding, not the character, to compressed file
  - Why does this save bits?
  - What other information needed in compressed file?

- How do we uncompress?
  - How do we know foo.hf represents compressed file?
  - Is suffix sufficient? Alternatives?

- Why is Huffman coding a two-pass method?
  - Alternatives?

# Uncompression with Huffman

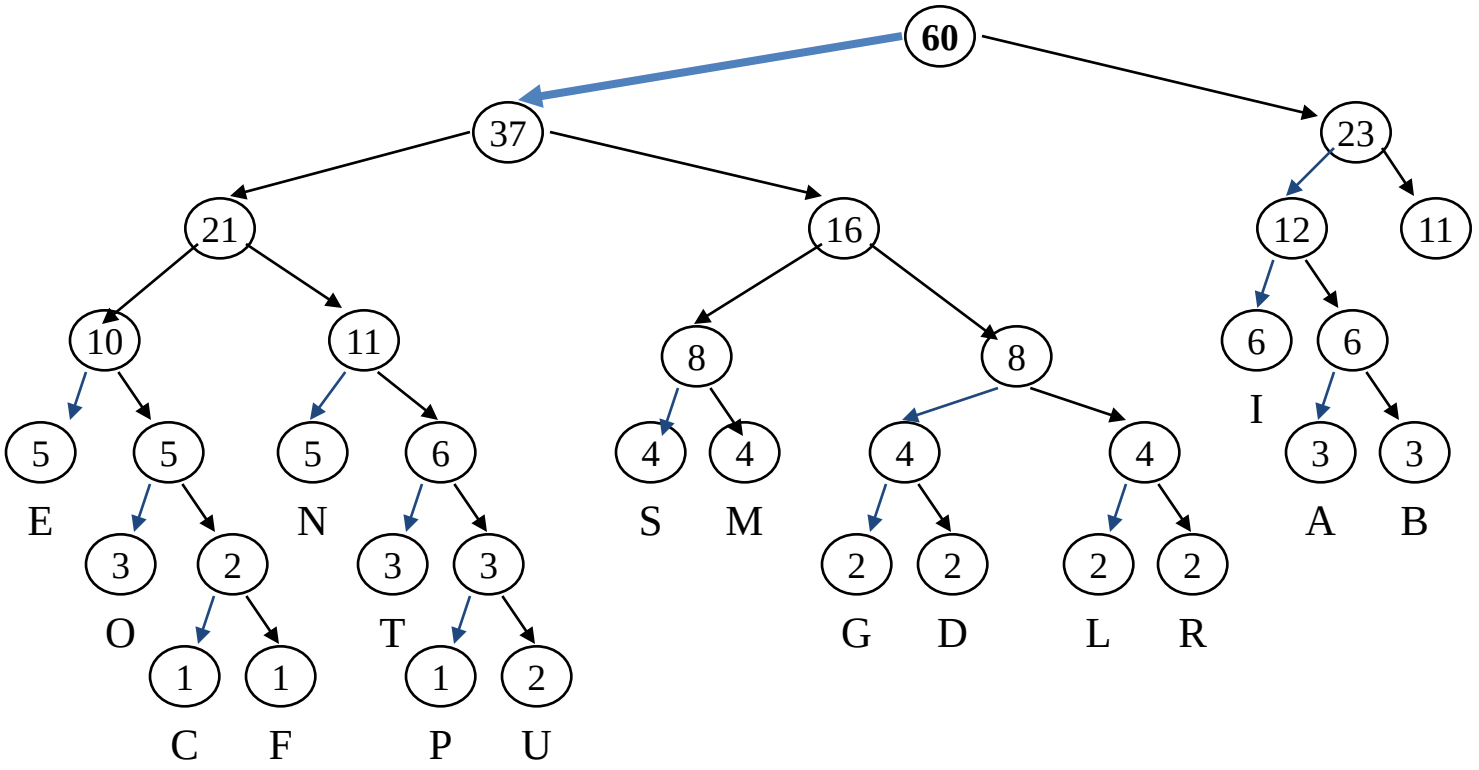
- We need the trie to uncompress
  - `000100100010011001101111`
  - `What is this?`
- As we read a bit, what do we do?
  - Go left on 0, go right on 1
  - When do we stop? What to do?




- How do we get the trie?
  - How did we get it originally? Store 256 int/counts
    - How do we read counts?
  - How do we store a trie? 20 Questions relevance?
    - Reading a trie? Leaf indicator? Node values?

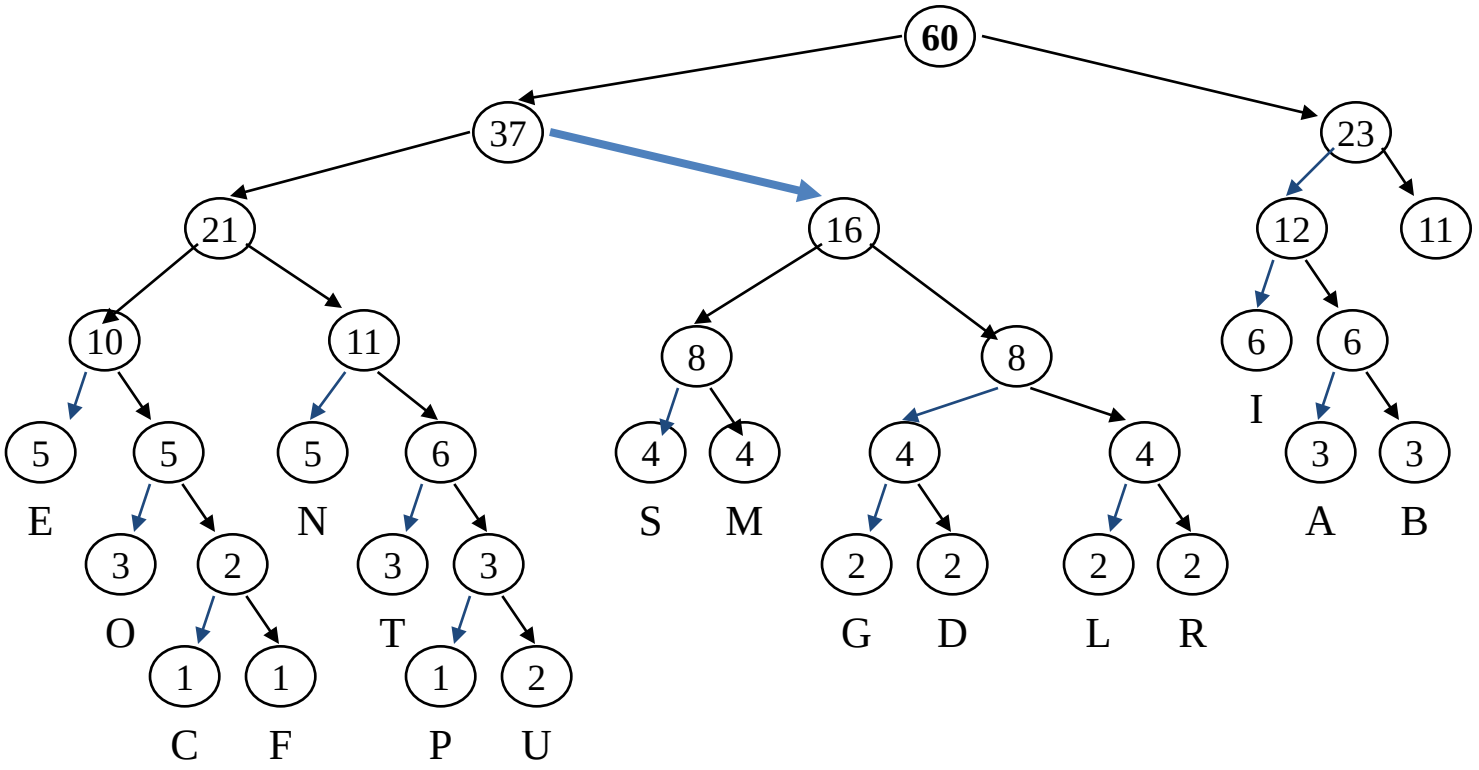
# Decoding a message

**0**11000001000001001101



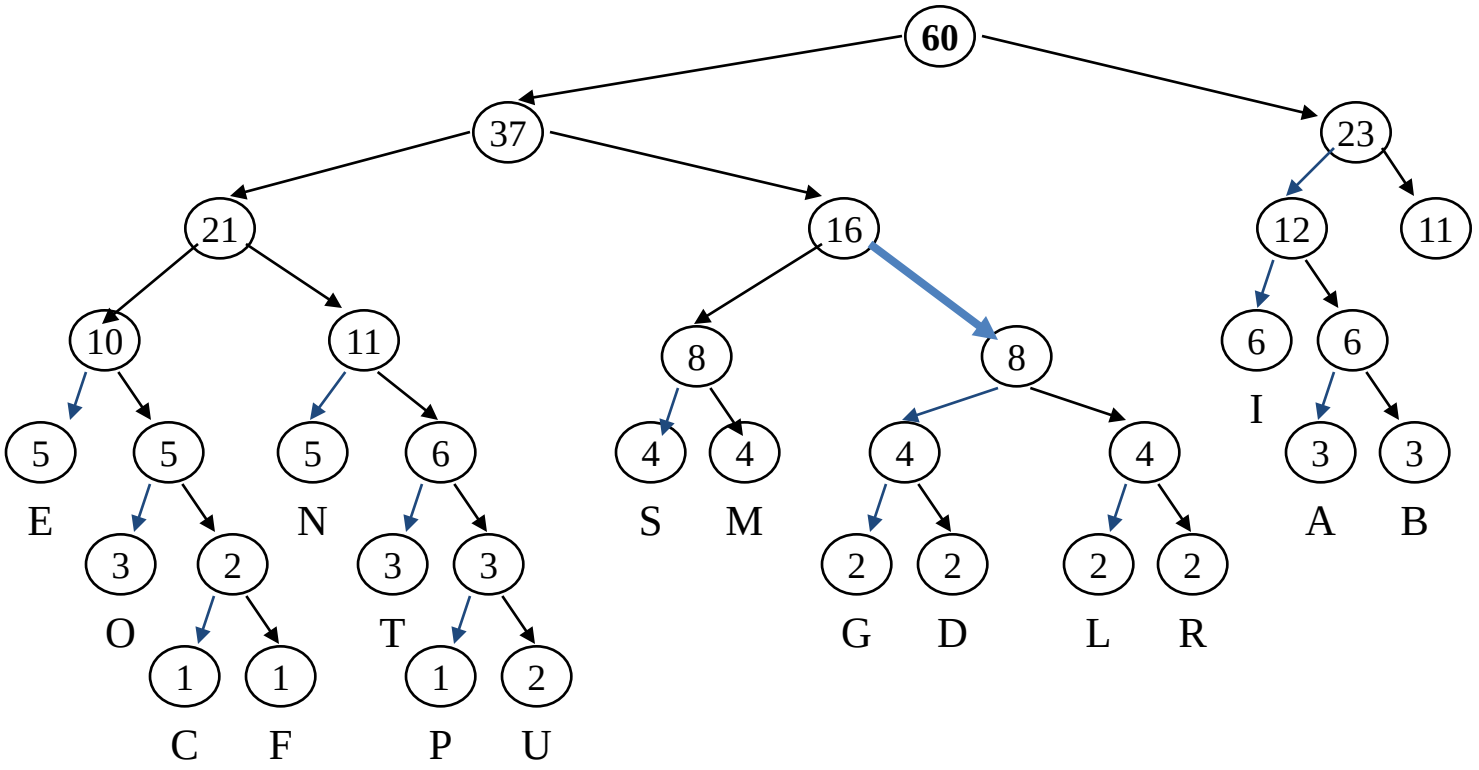
# Decoding a message

1100000100001001101



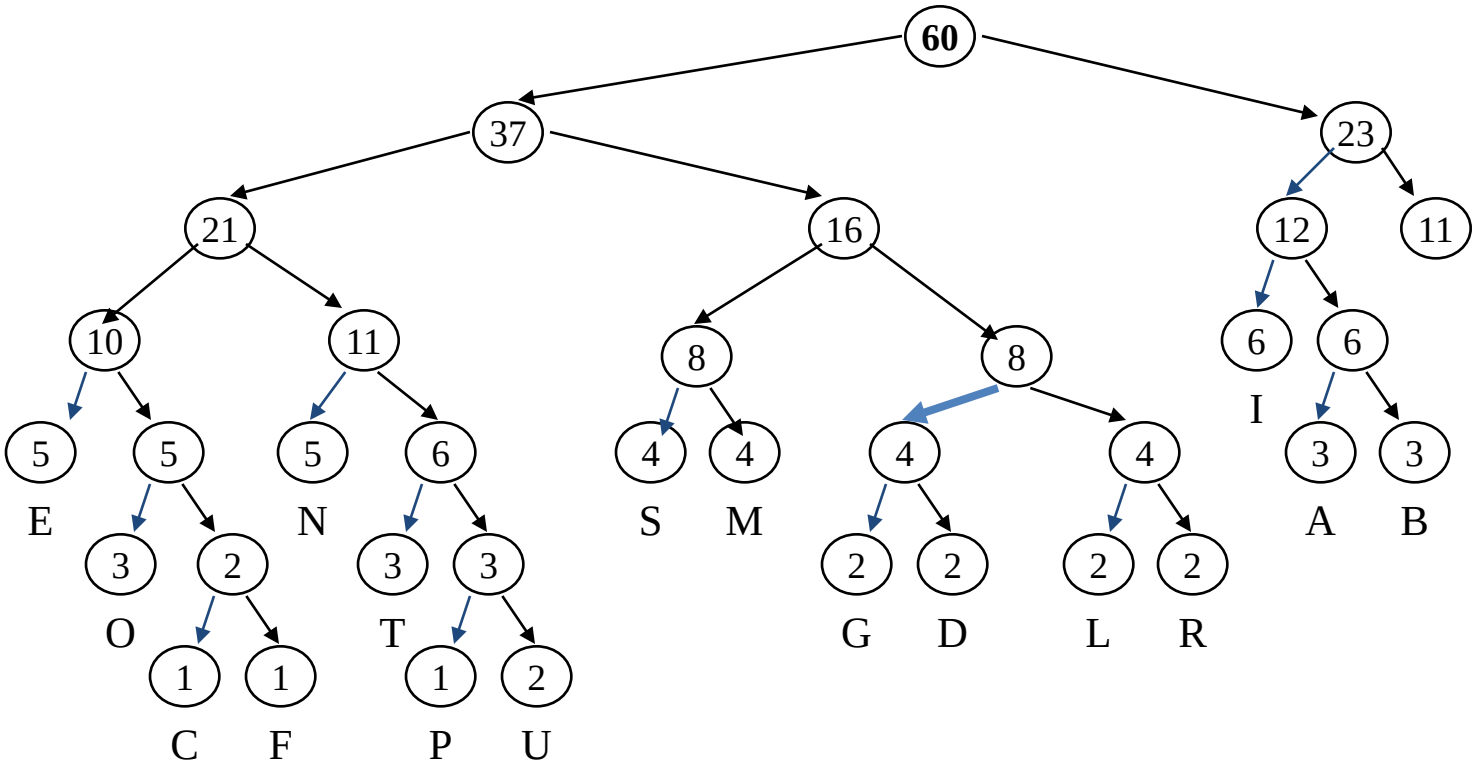
# Decoding a message

10000010000101101



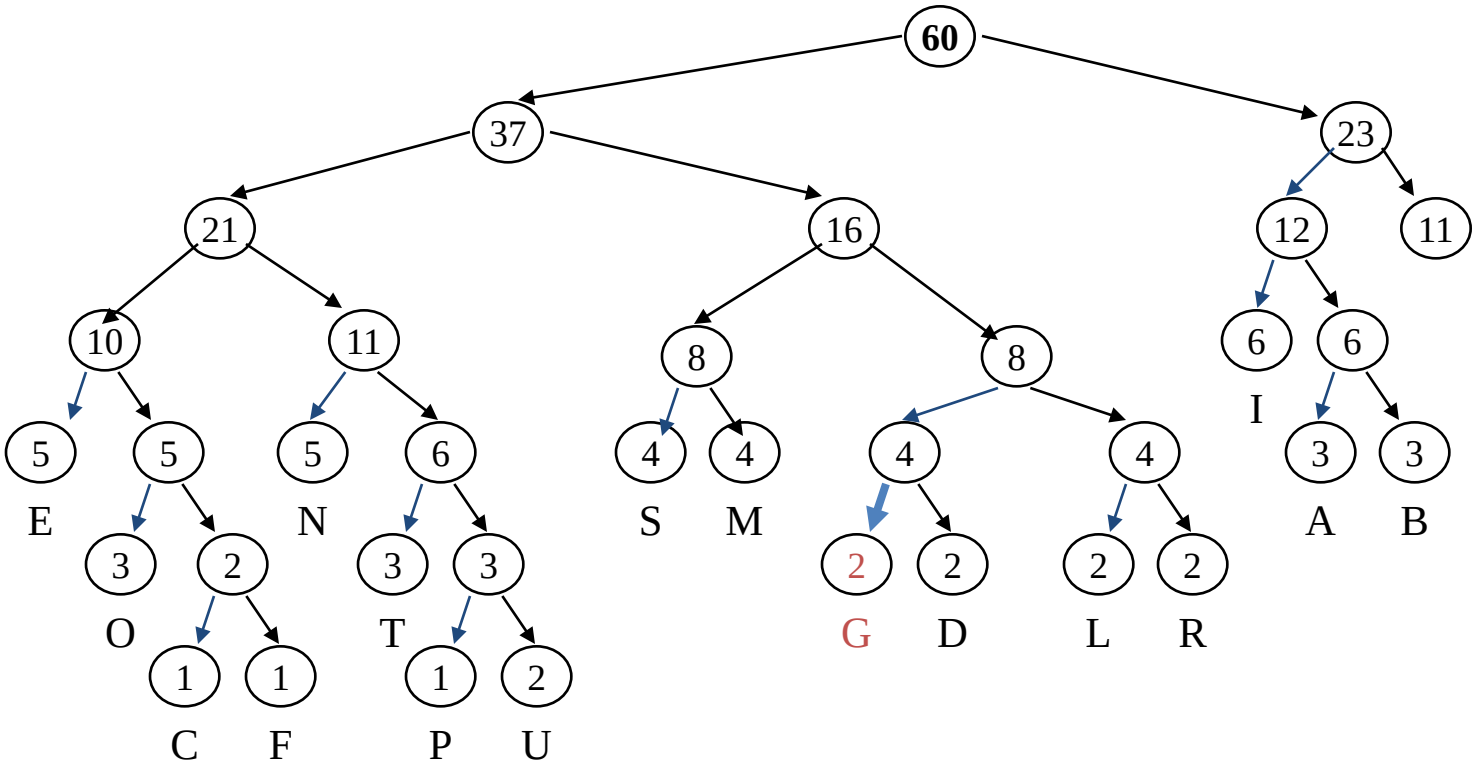
# Decoding a message

0000010000101101



# Decoding a message

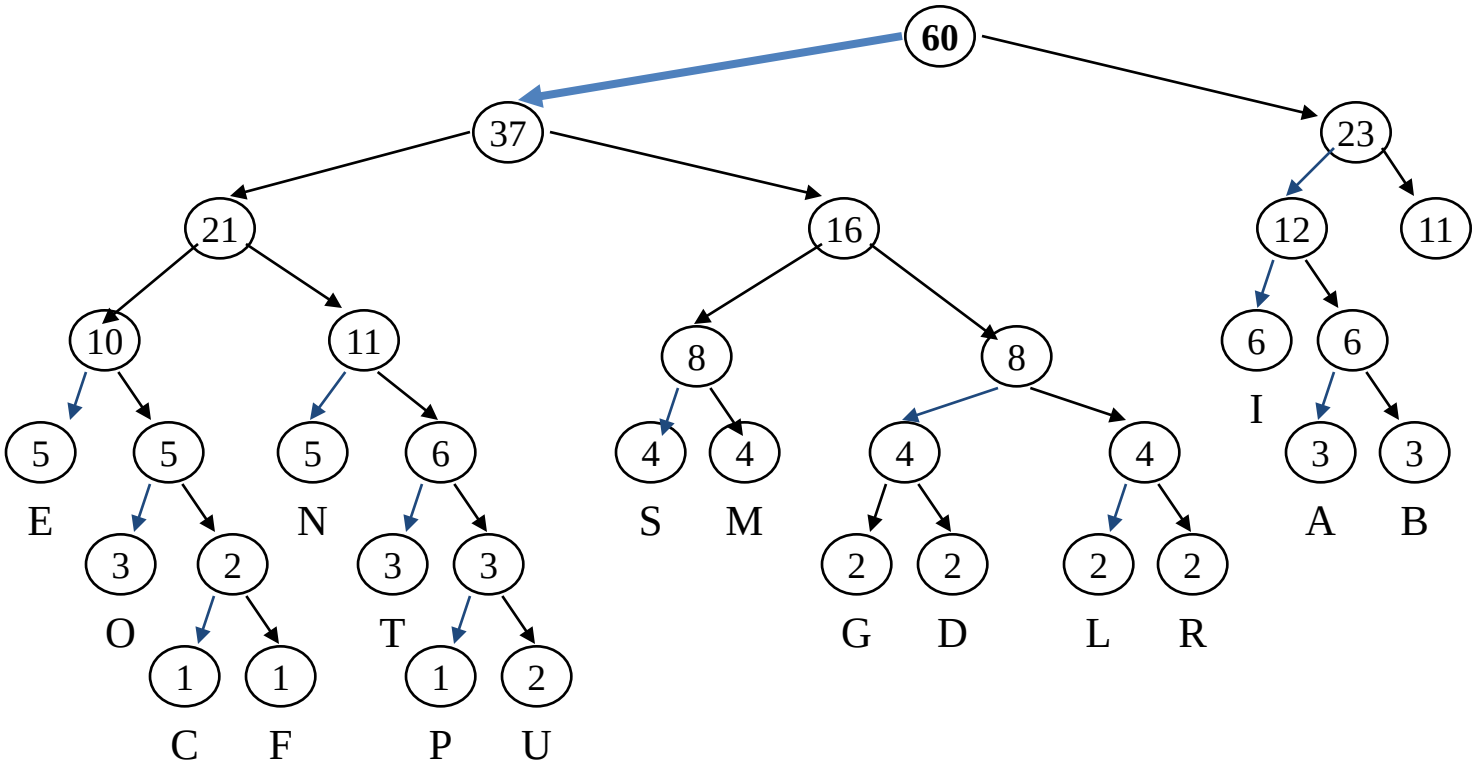
0000100001001101




G

# Decoding a message

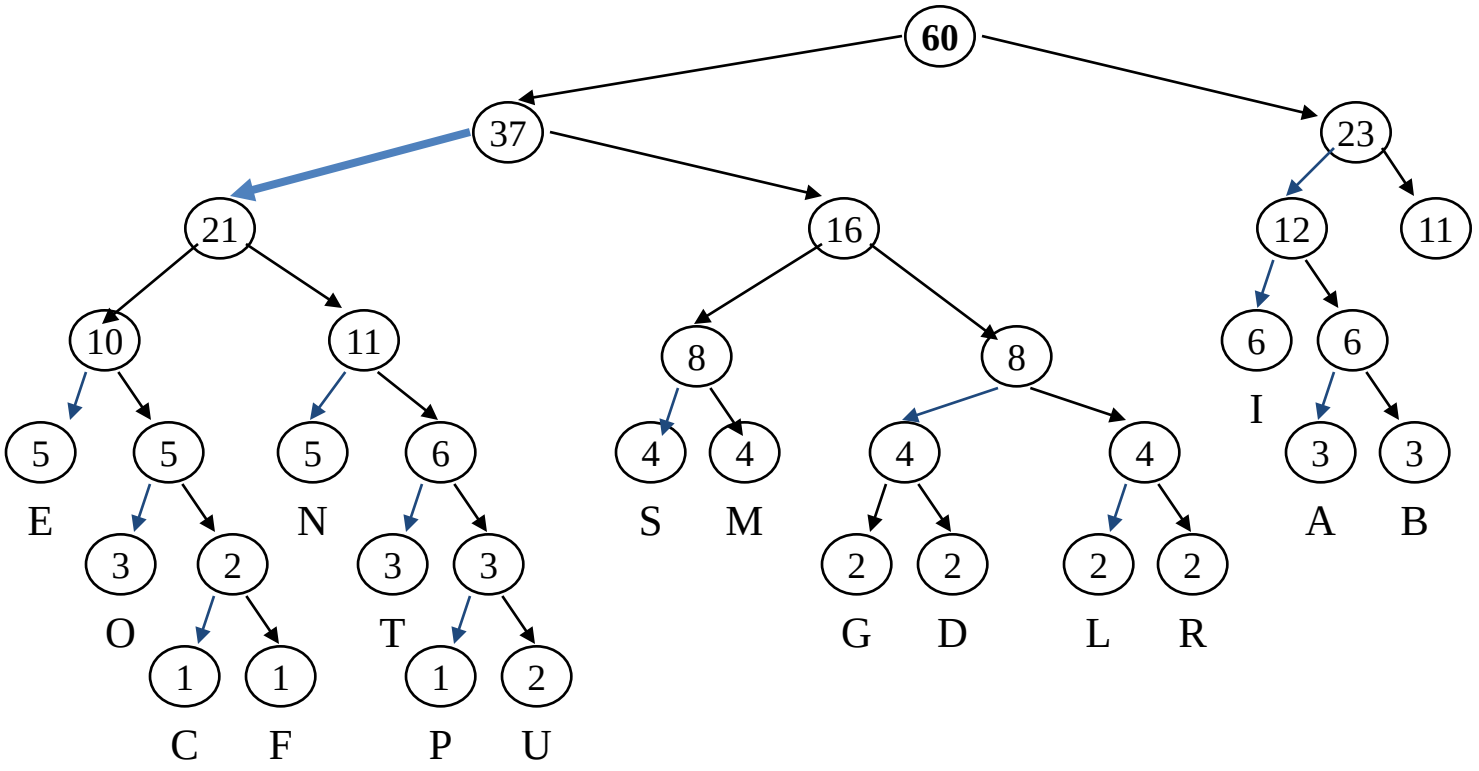
0001000010011101




G

# Decoding a message

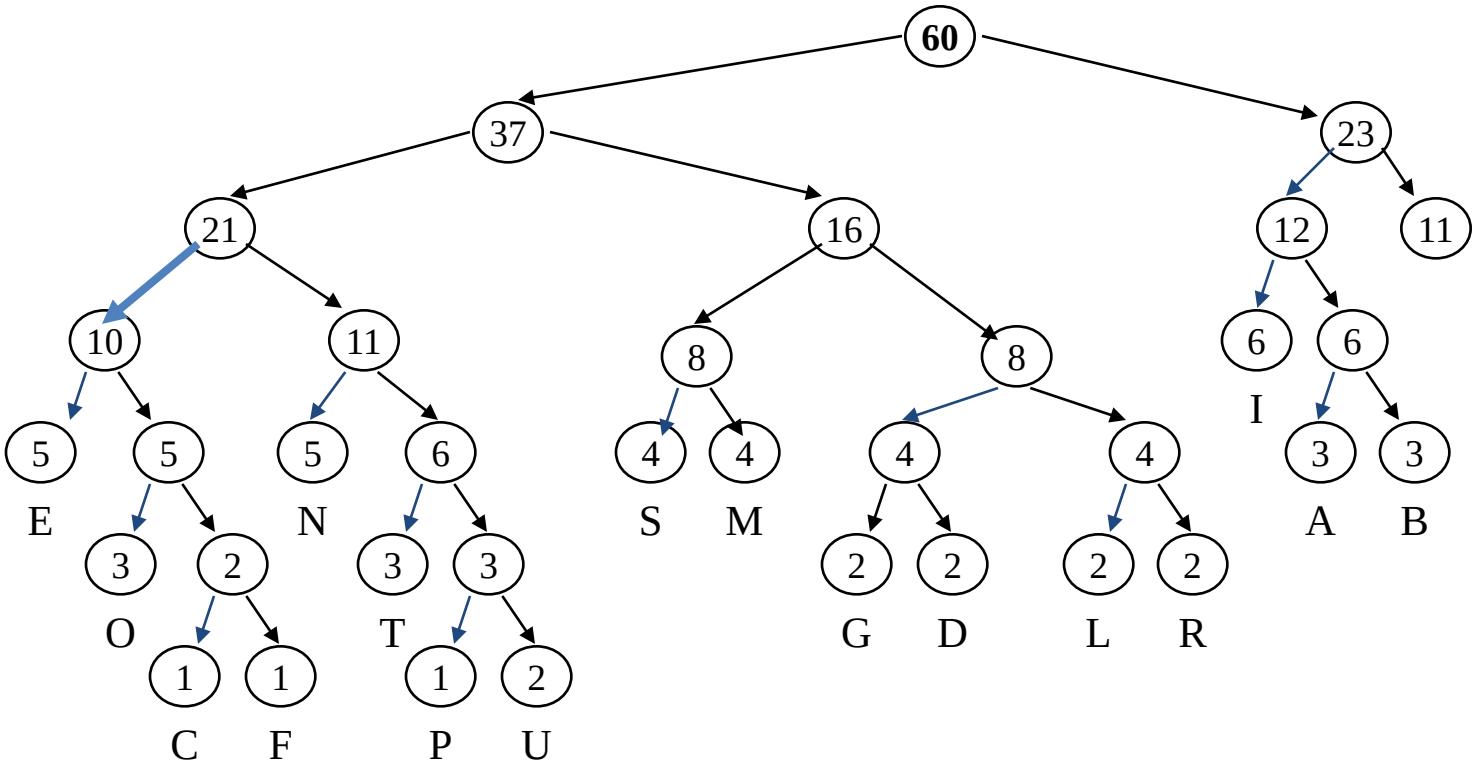
0010000101101




G

# Decoding a message

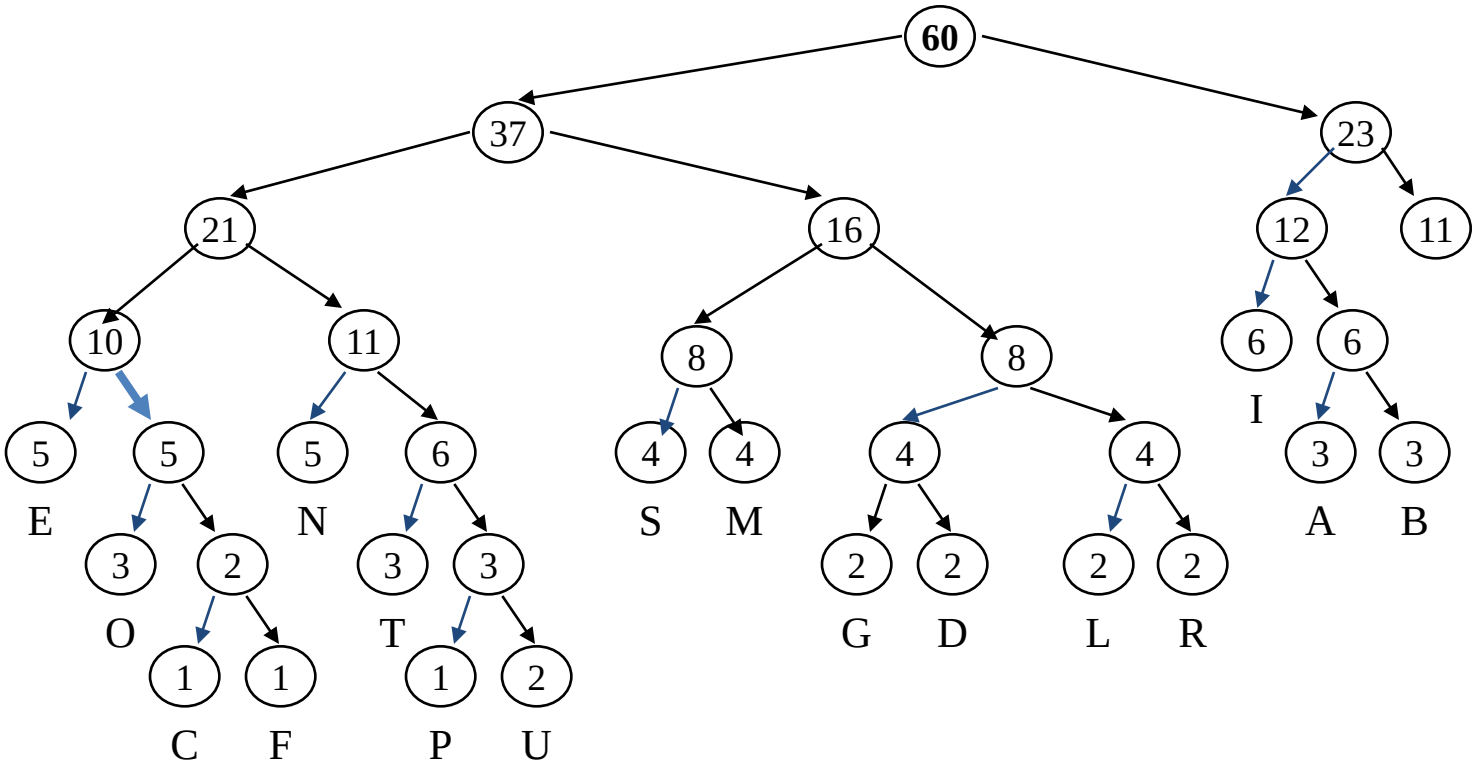
0100001001101




G

# Decoding a message

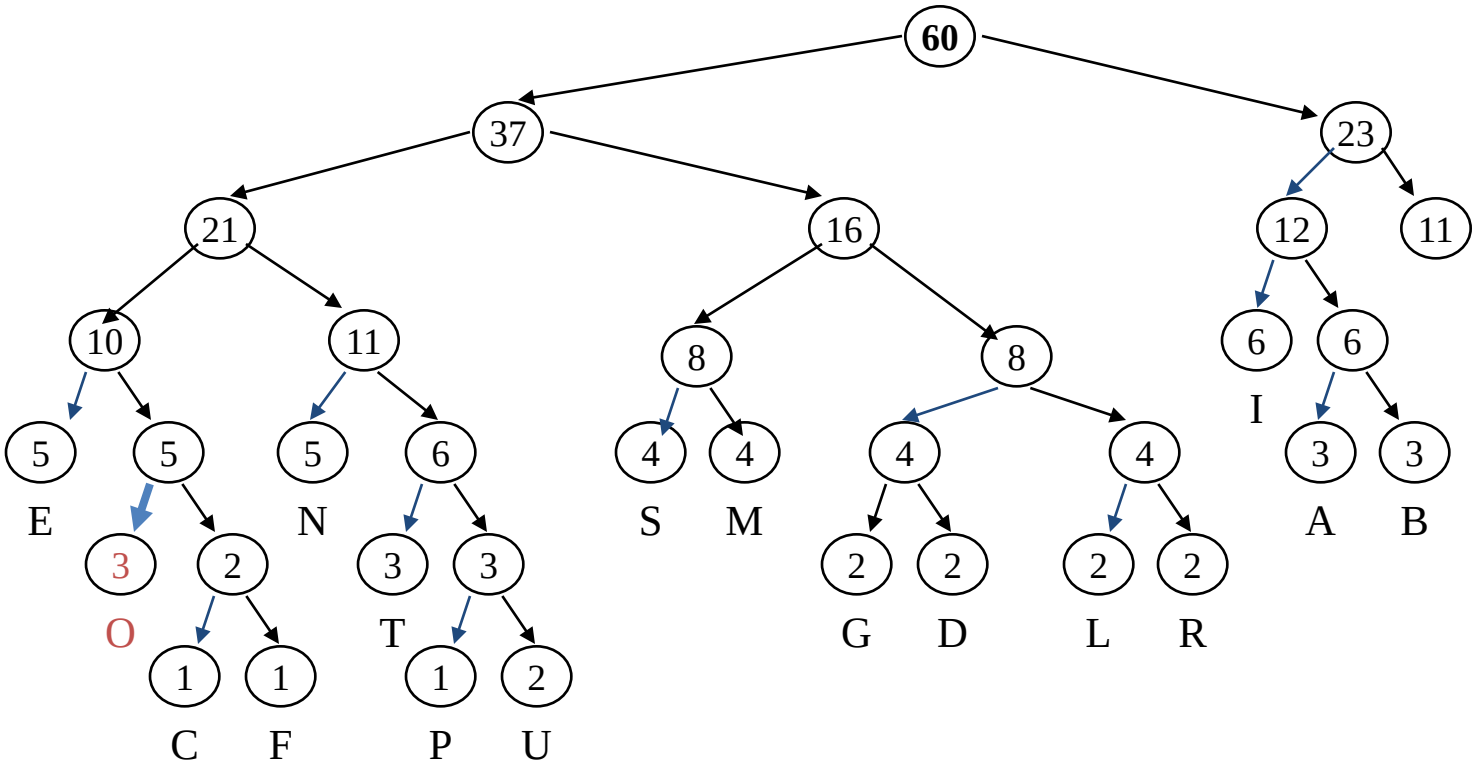
100001001101




G

# Decoding a message

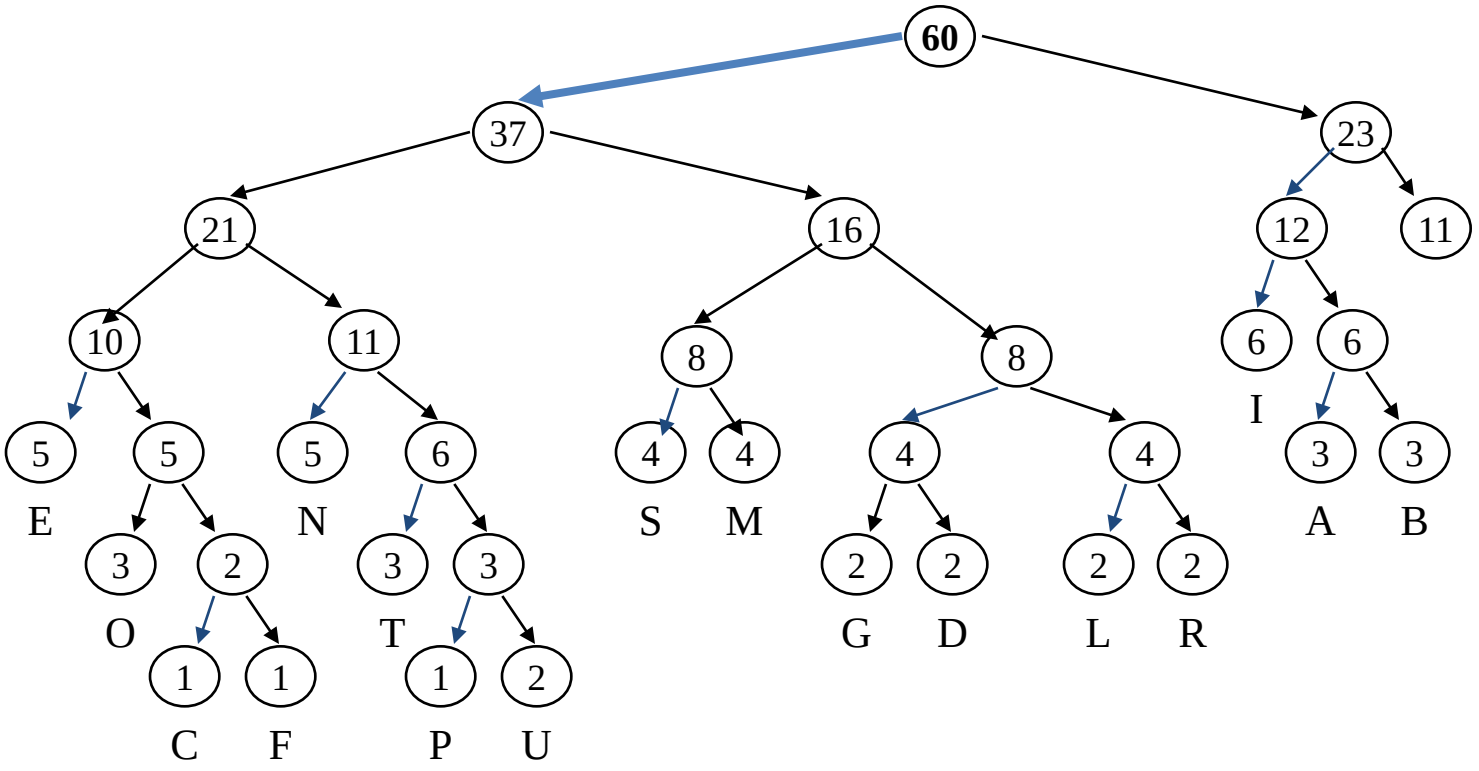
00001001101




GO

# Decoding a message

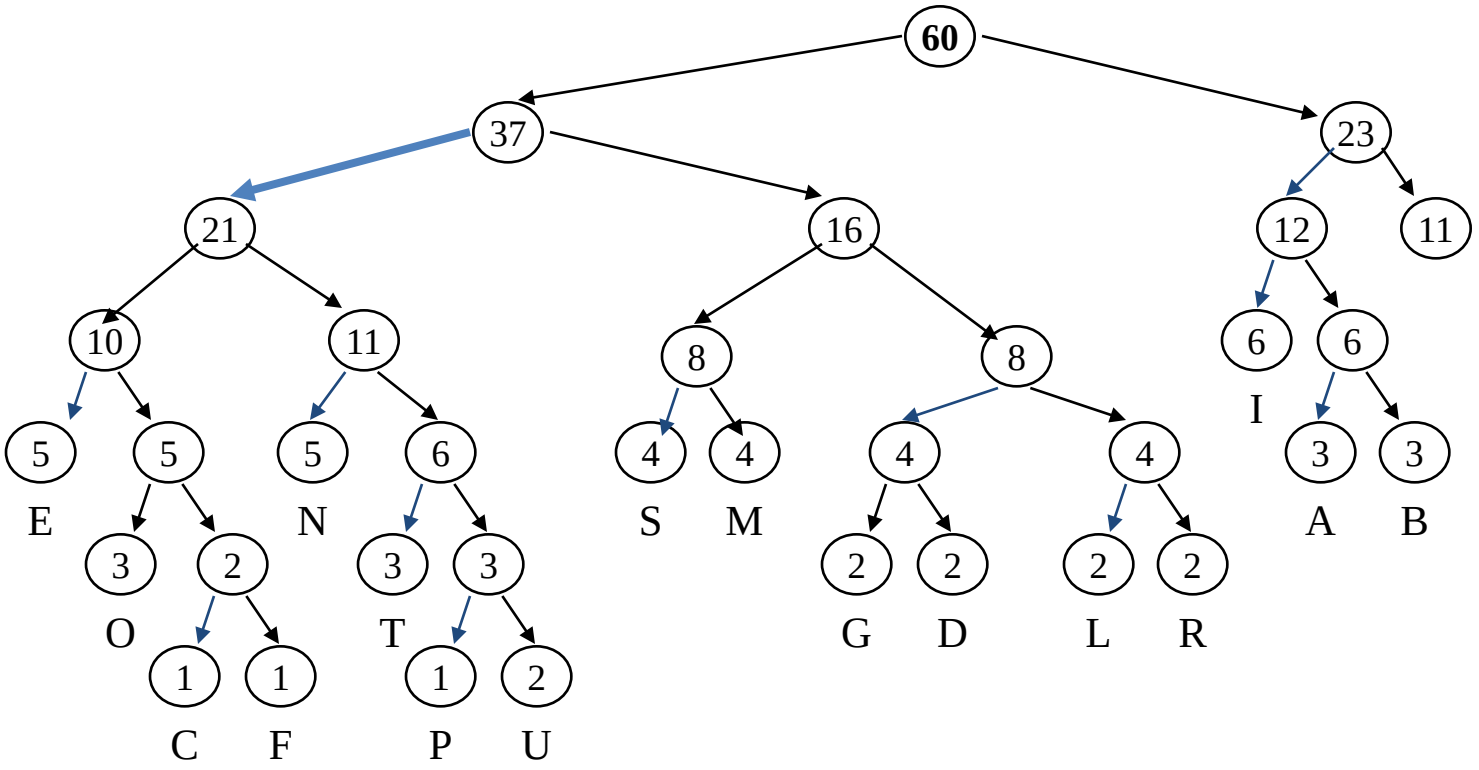
0001001101




GO

# Decoding a message

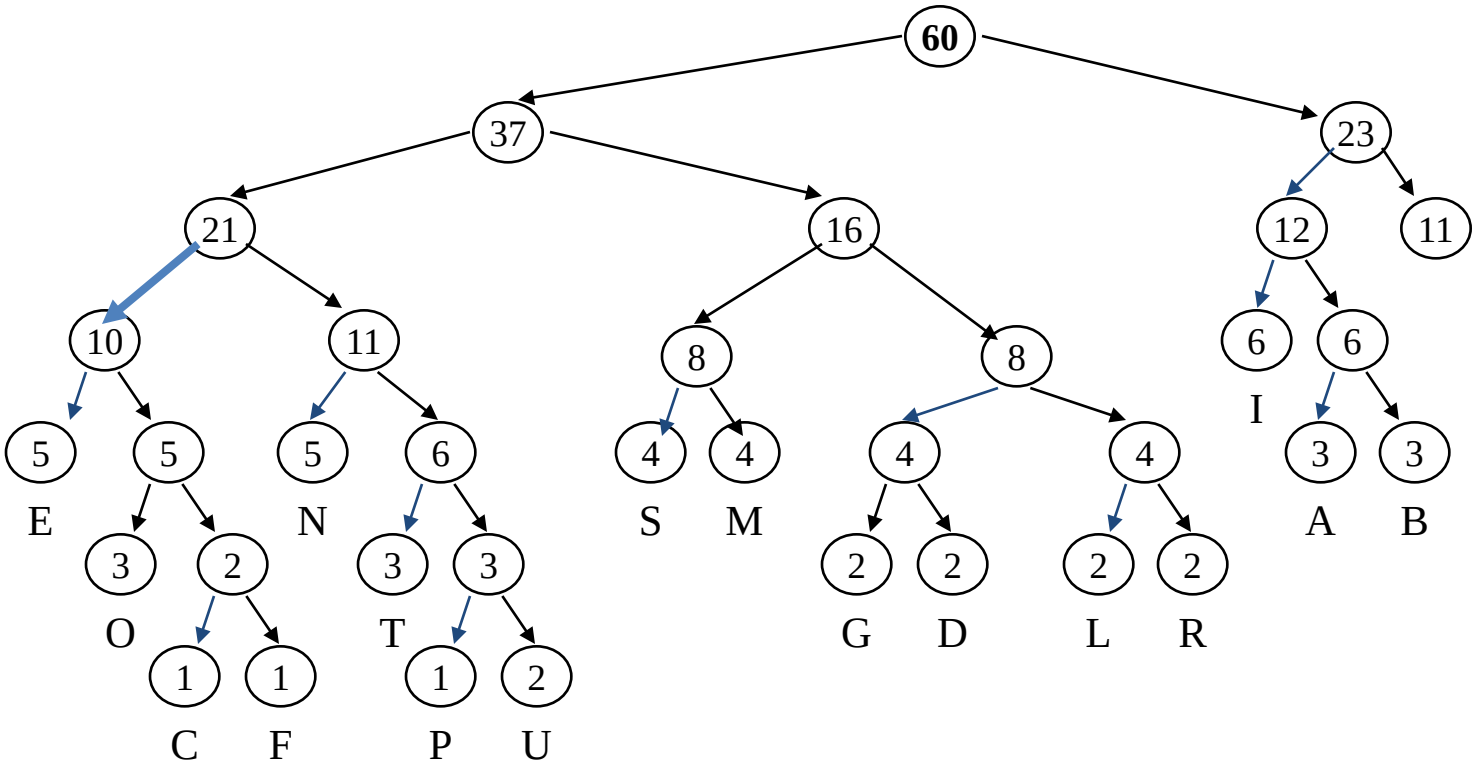
0**0**1001101




GO

# Decoding a message

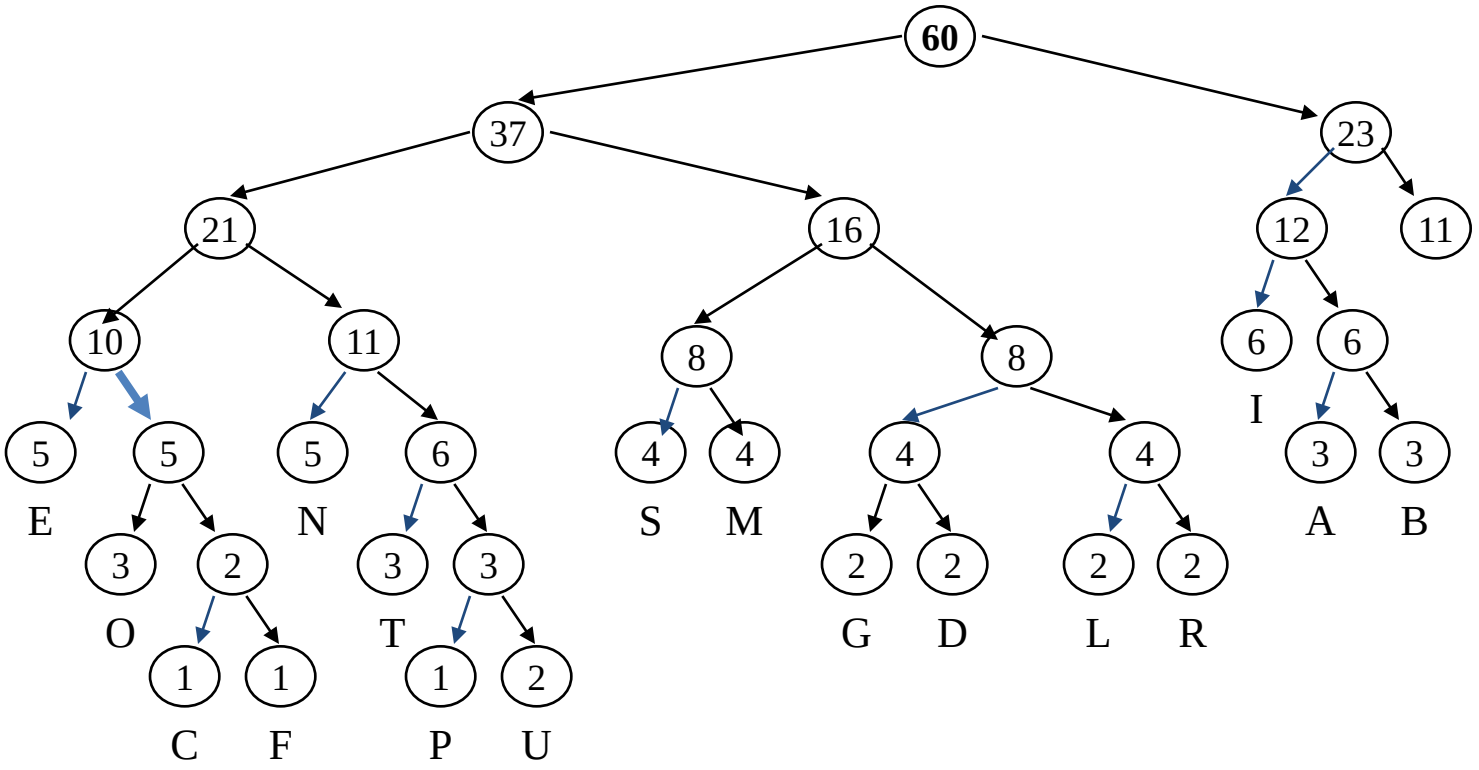
01001101




GO

# Decoding a message

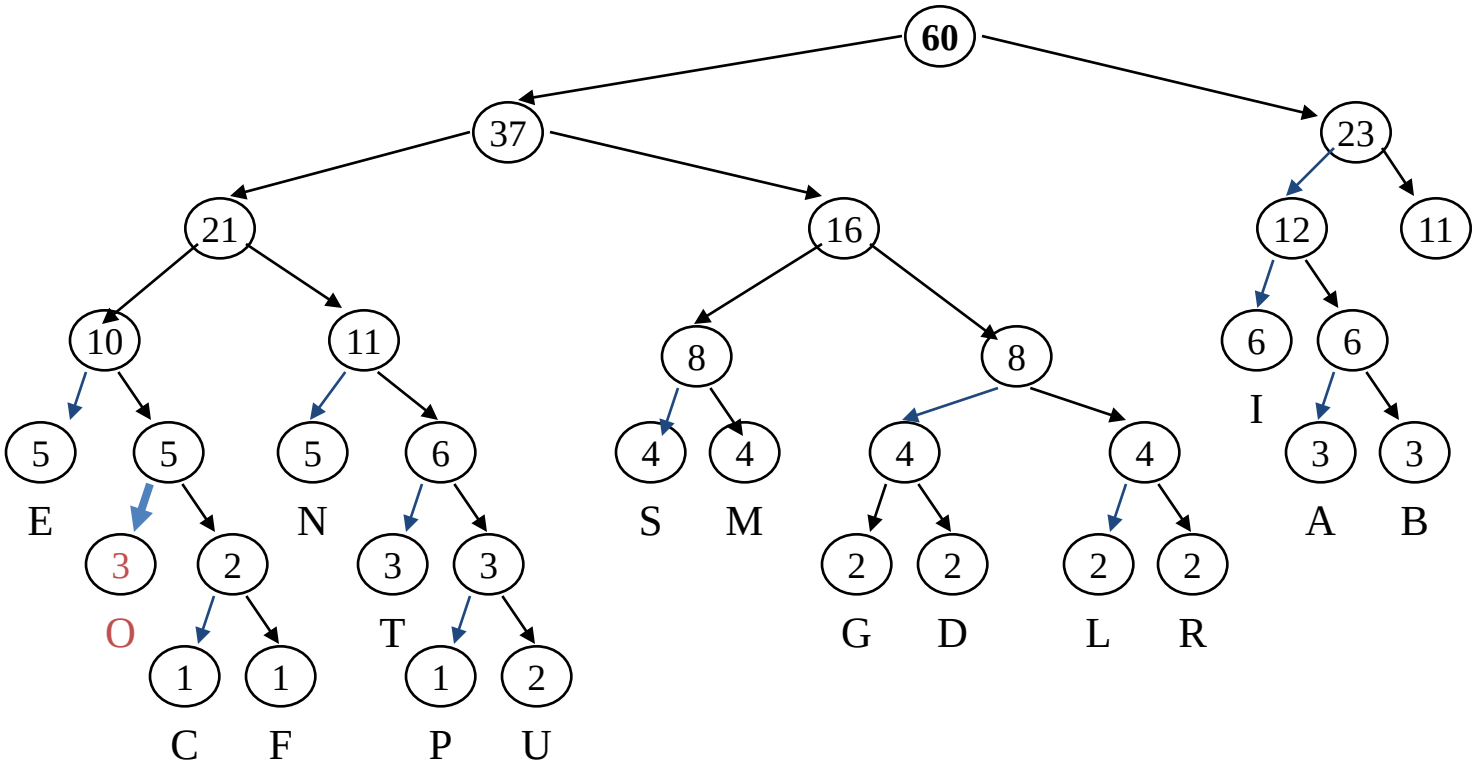
1001101




GO

# Decoding a message

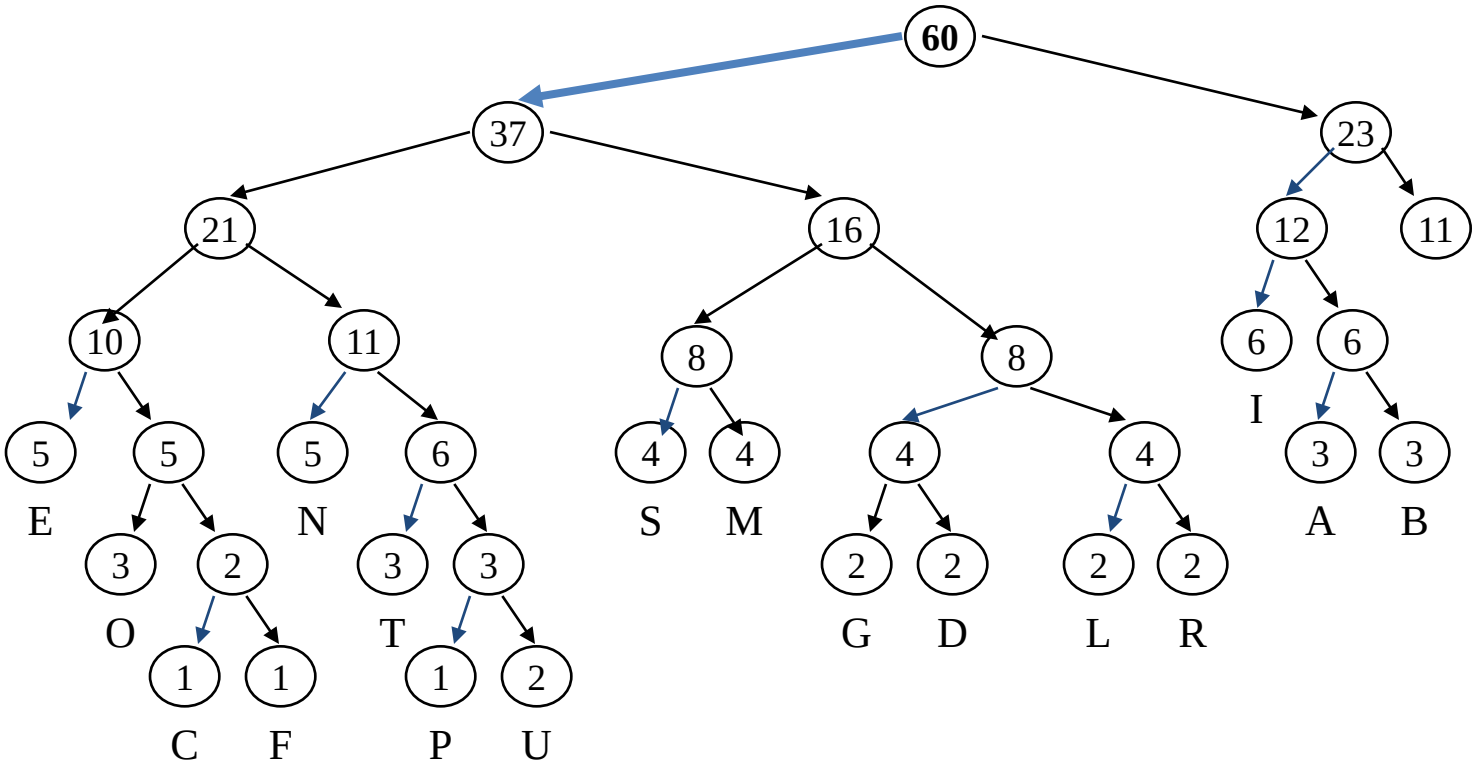
001101




GOO

# Decoding a message

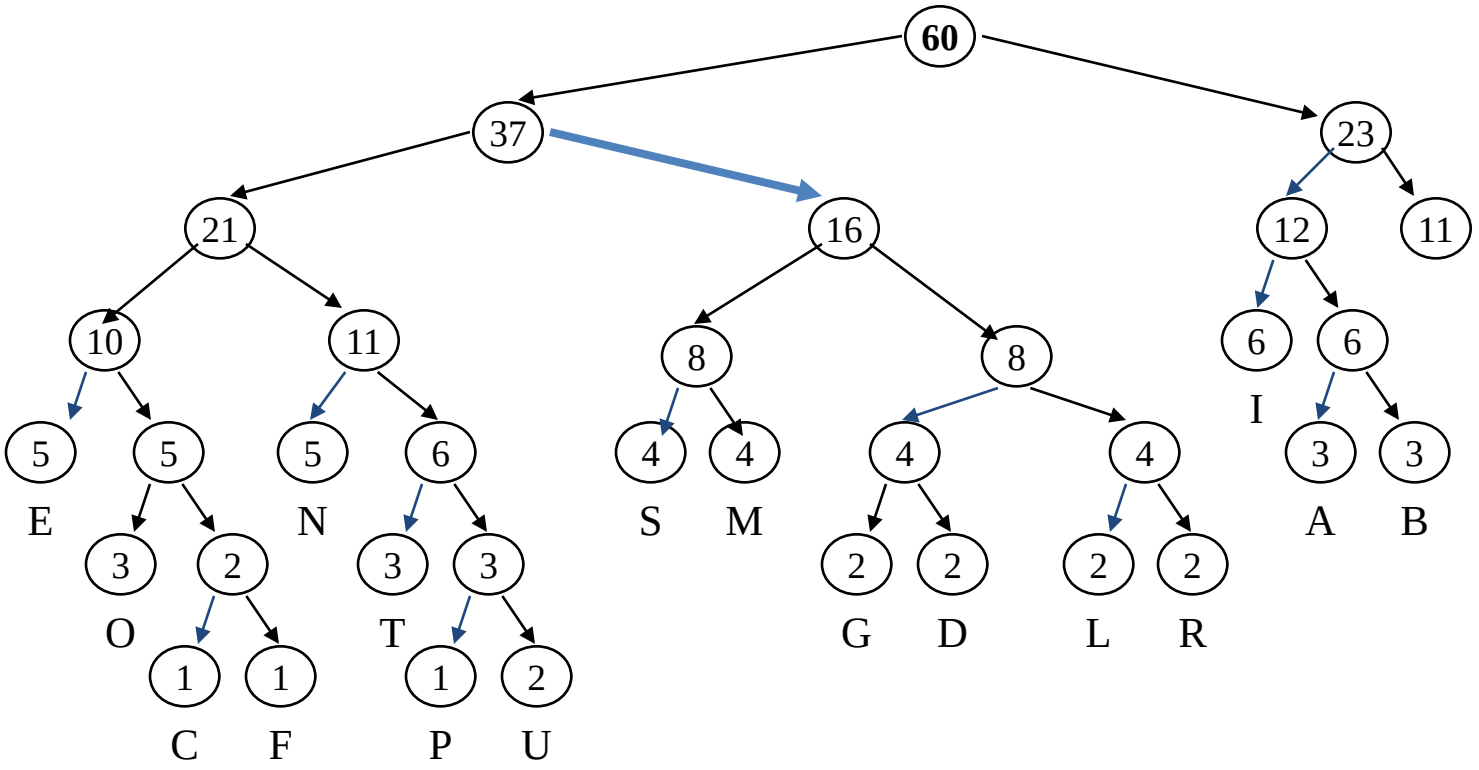
01101




GOO

# Decoding a message

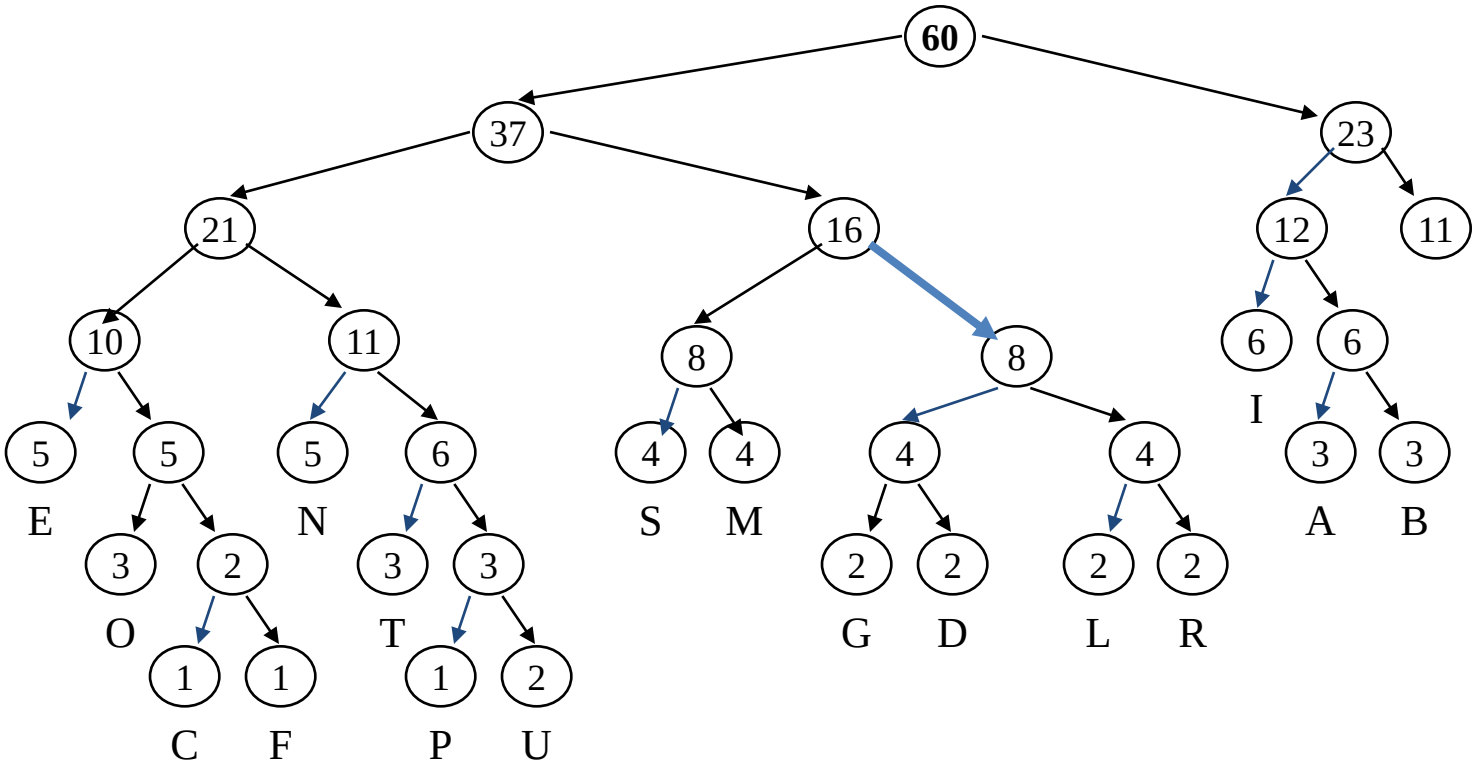
1101




GOO

# Decoding a message

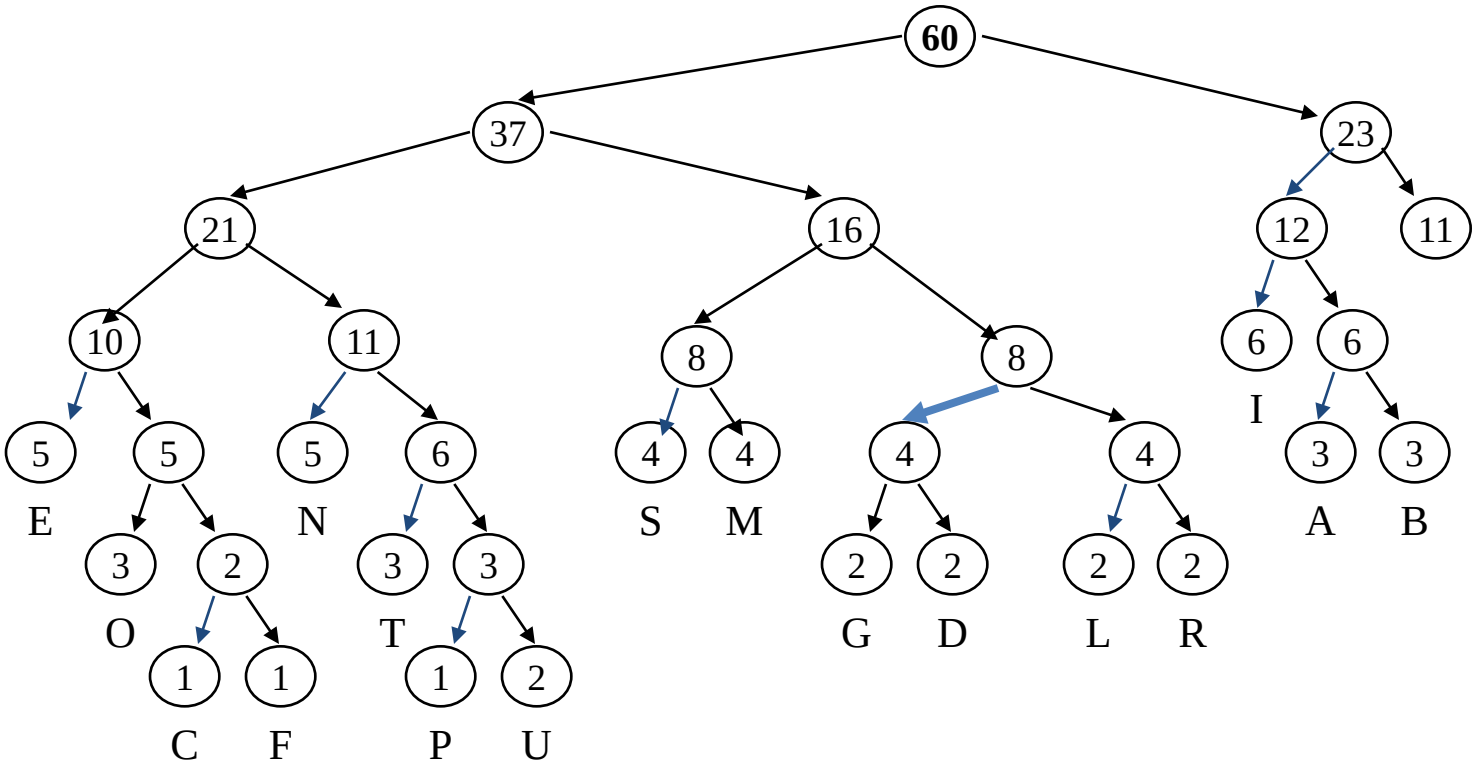
101




GOO

# Decoding a message

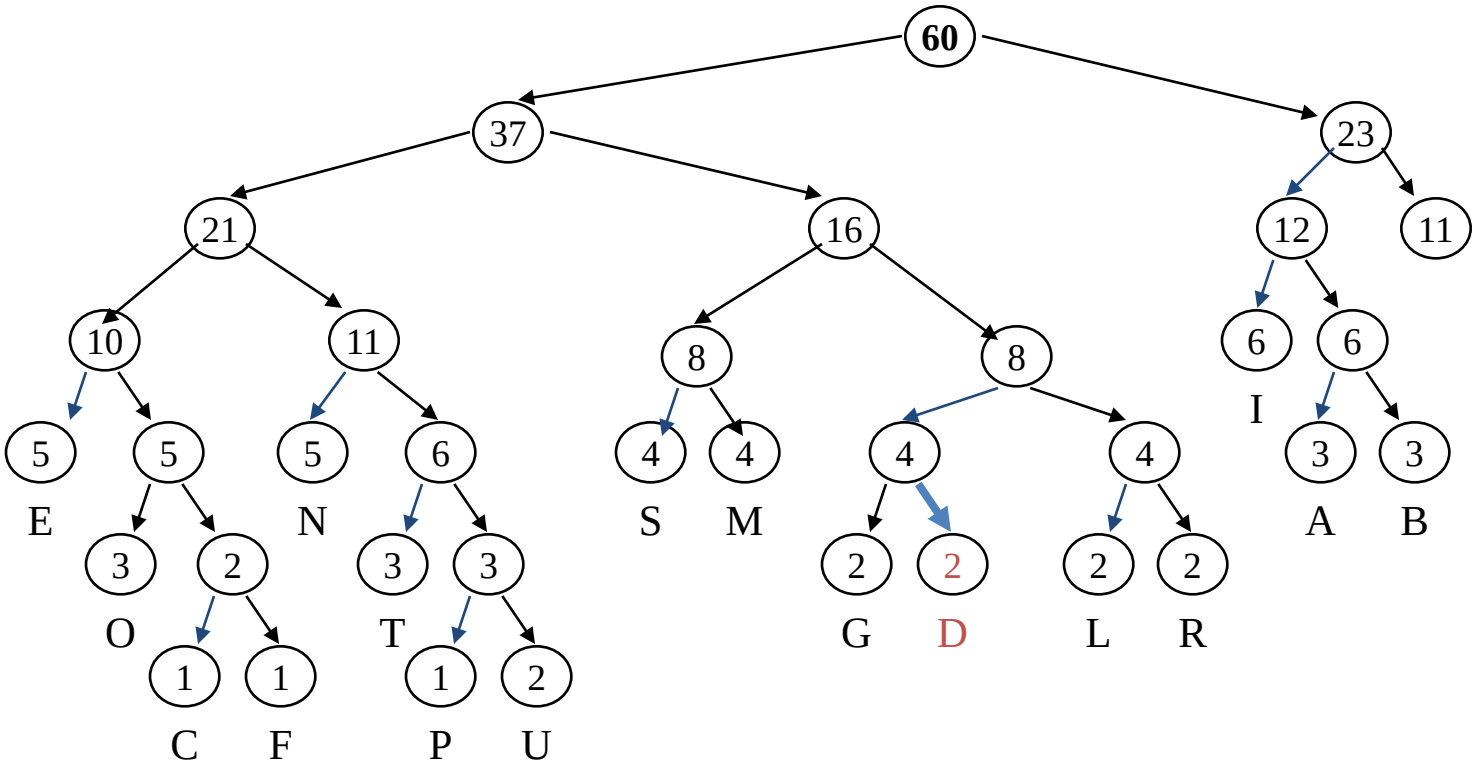
01




GOO

# Decoding a message

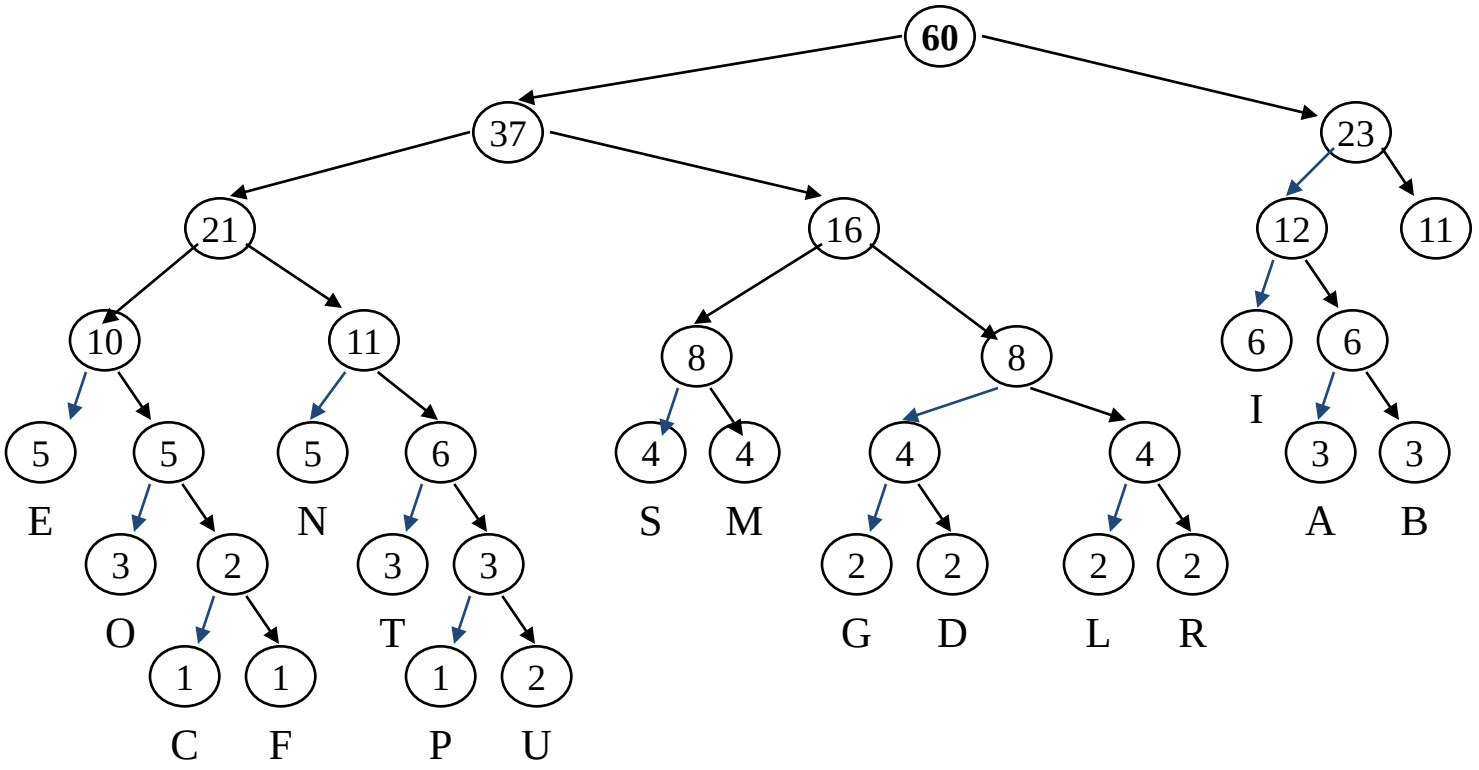
1




GOOD

# Decoding a message

01100000100001001101




GOOD

# Other Huffman Issues

- What do we need to decode?
  - How did we encode? How will we decode?
  - What information needed for decoding?

- Reading and writing bits: chunks and stopping
  - Can you write 3 bits? Why not? Why?
  - PSEUDO_EOF
  - BitInputStream and BitOutputStream: API

- What should happen when the file won't compress?
  - Silently compress bigger? Warn user? Alternatives?

# Huffman Complexities

- How do we measure? Size of input file, size of alphabet
  - Which is typically bigger?

- Accumulating character counts: _____
  - How can we do this in O(1) time, though not really
- Building the heap/priority queue from counts ____
  - Initializing heap guaranteed
- Building Huffman tree ____
  - Why?
- Create table of encodings from tree ____
  - Why?
- Write tree and compressed file _____

# Good Compsci 100 Assignment?

- Array of character/chunk counts, or is this a map?
  - Map character/chunk to count, why array?
- Priority Queue for generating tree/trie
  - Do we need a heap implementation? Why?
- Tree traversals for code generation, uncompression
  - One recursive, one not, why and which?
- Deal with bits and chunks rather than ints and chars
  - The good, the bad, the ugly
- Create a working compression program
  - How would we deploy it? Make it better?
- Benchmark for analysis
  - What's a *corpus*?

# Other methods

- Adaptive Huffman coding
- Lempel-Ziv algorithms
  - Build the coding table on the fly while reading document
  - Coding table changes dynamically
  - Protocol between encoder and decoder so that everyone is always using the right coding scheme
  - Works well in practice (`compress`, `gzip`, etc.)
- More complicated methods
  - Burrows-Wheeler (`bunzip2`)
  - PPM statistical methods

# Data Compression

| Year | Scheme | Bit/Char |
|---|---|---|
| 1967 | ASCII | 7.00 |
| 1950 | Huffman | 4.70 |
| 1977 | Lempel-Ziv (LZ77) | 3.94 |
| 1984 | Lempel-Ziv-Welch (LZW) – Unix compress | 3.32 |
| 1987 | (LZH) used by zip and unzip | 3.30 |
| 1987 | Move-to-front | 3.24 |
| 1987 | gzip | 2.71 |
| 1995 | Burrows-Wheeler | 2.29 |
| 1997 | BOA (statistical data compression) | 1.99 |

- Why is data compression important?
- How well can you compress files losslessly?
  - Is there a limit?
  - How to compare?
- How do you measure how much information?

# From bit to byte to char to int to long

- Ultimately everything is stored as either a 0 or 1
  - Bit is binary digit  a byte is a binary term (8 bits)
  - We should be grateful we can deal with Strings rather than sequences of 0's and 1's.
  - We should be grateful we can deal with an int rather than the 32 bits that comprise an int

- If we have 255 values for R, G, B, how can we pack this into an int?
  - Why should we care, can't we use one int per color?
  - How do we do the packing and unpacking?

# More information on bit, int, long

- int values are stored as two's complement numbers with 32 bits, for 64 bits use the type long, a char is 16 bits
  - Standard in Java, different in C/C++
  - Facilitates addition/subtraction for int values
  - We don't need to worry about this, except to note:
    - Infinity + 1 = - Infinity (see `Integer.MAX_VALUE`)
    - Math.abs(-Infinity) > Infinity
- Java byte, int, long are signed values, char unsigned
  - What are values for 16-bit char? 8-bit byte?
  - Why will this matter in Burrows Wheeler?

# Signed, unsigned, and why we care

- Some applications require attention to memory-use
  - Differences: one-million bytes, chars, and int
    - First requires a megabyte, last requires four megabytes
    - When do we care about these differences?
  - Memory is cheaper, faster, …But applications expand to use it
- Java signed byte: `-128..127, # bits?`
  - What if we only want 0-255? (Huff, pixels, …)
  - Convert negative values or use char, trade-offs?
- Java char unsigned: `0..65,536 #` bits?
  - Why is char unsigned? Why not as in C++/C?

# More details about bits

- How is 13 represented?
  - … 0 0 1 1 0 1
    $2^4$ $2^3$ $2^2$ $2^1$ $2^0$
  - Total is 8+4+1 = 13
- What is bit representation of 32? Of 15? Of 1023?
  - What is bit-representation of $2^n - 1$?
  - What is bit-representation of 0? Of -1?
    - Study later, but -1 is all 1's, left-most bit determines < 0
- Determining what bits are on? How many on?
  - Understanding, problem-solving

# How are data stored?

- To facilitate Huffman coding we need to read/write one bit
  - Why do we need to read one bit?
  - Why do we need to write one bit?
  - When do we read 8 bits at a time? 32 bits?
- We can't actually write one bit-at-a-time. We can't really write one char at a time either.
  - Output and input are buffered,minimize memory accesses and disk accesses
  - Why do we care about this when we talk about data structures and algorithms?
    - Where does data come from?

# How do we buffer char output?

- Done for us as part of InputStream and Reader classes
  - InputStreams are for reading bytes
  - Readers are for reading char values
  - Why do we have both and how do they interact?

```
Reader r = new
  InputStreamReader(System.in);
```

  - Do we need to flush our buffers?
- In the past Java IO has been notoriously slow
  - Do we care about I? About O?
  - This is changing, and the java.nio classes help
    - Map a file to a region in memory in one operation

# Buffer bit output

- To buffer bits we store bits in a buffer (duh)
  - When the buffer is full, we write it.
  - The buffer might overflow, e.g., in process of writing 10 bits to 32-bit capacity buffer that has 29 bits in it
  - How do we access bits, add to buffer, etc.?
- We need to use bit operations
  - Mask bits -- access individual bits
  - Shift bits – to the left or to the right
  - Bitwise and/or/negate bits

# Representing pixels

- Pixel typically stores RGB and alpha/transparency values
  - Each RGB is a value in the range 0 to 255
  - The alpha value is also in range 0 to 255

```
Pixel red = new Pixel(255,0,0,0);
Pixel white = new Pixel(255,255,255,0);
```

- A picture is simply an array of int values

```
void process(int pixel){
   int blue = pixel & 0xff;
   int green = (pixel >> 8) & 0xff;
   int red = (pixel >> 16) & 0xff;
}
```

# Bit masks and shifts

```
void process(int pixel){
   int blue  = pixel & 0xff;
   int green = (pixel >> 8) & 0xff;
   int red   = (pixel >> 16) & 0xff;
}
```

- Hexadecimal number: 0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f
  - f is 15, in binary this is 1111, one less than 10000
  - The hex number 0xff is an 8 bit number, all ones
- Bitwise & operator creates an 8 bit value, 0—255
  - Must use an int/char, what happens with byte?
  - 1&1 == 1, otherwise we get 0 like *logical and*
  - Similarly we have |, bitwise or