# ASCII encoding

## 1 byte (8 bits) per char, 256 distinct chars representable

| | | |
|---|---|---|
| `' '` | 32 | 00100000 |
| `'0'` | 48 | 00110000 |
| `'1'` | 49 | 00110001 |
| `'A'` | 65 | 01000001 |
| `'B'` | 66 | 01000010 |
| `'a'` | 97 | 01100001 |
| `'b'` | 98 | 01100010 |
| `'™'` | 128 | 10000000 |
| `'≠'` | 129 | 10000001 |
| `'}'` | 253 | 11111101 |
| `'~'` | 254 | 11111110 |

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

| A | | S | I |
|---|---|---|---|
| 01000001 | 00100000 | 01010011 | 01001001 |

| M | P | L | E |
|---|---|---|---|
| 01001101 | 01010000 | 01001100 | 01000101 |

**60 characters * 8 bits per char = 480 bits total**

**ASCII encoding:**

+ industry standard
+ encoding is fixed (no specialized table, can encode all chars)
- wasteful if not using all 256 different chars

? all chars use same number of bits

# Compact fixed-length encoding

**N alphabet = 18, use 5 bits per char (32 distinct chars representable)**

| | | |
|---|---|---|
| 'A' | 0 | 00000 |
| ' ' | 1 | 00001 |
| 'S' | 2 | 00010 |
| 'I' | 3 | 00011 |
| 'M' | 4 | 00100 |
| 'P' | 5 | 00101 |
| 'L' | 6 | 00110 |
| 'E' | 7 | 00111 |
| 'T' | 8 | 01000 |
| 'R' | 9 | 01001 |
| 'N' | 10 | 01010 |
| 'G' | 11 | 01011 |
| 'O' | 12 | 01100 |

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

| A | | S | I | M | P |
|---|---|---|---|---|---|
| 00000 | 00001 | 00010 | 00011 | 00100 | 00101 |

**60 characters * 5 bits per char = 300 bits total**

**300/480 = 63% of original size**

**Compact fixed-length encoding:**

**+** small alphabet => fewer bits per char

**-** encoding is custom (table required, can only encode characters in original alphabet)

**?** all chars use same number of bits

# Variable-length encoding

*Must we use same number of bits for each char??*

| | | | |
|---|---|---|---|
| A | • – | U | • • – |
| B | – • • • | V | • • • – |
| C | – • – • | W | • – – |
| D | – • • | X | – • • – |
| E | • | Y | – • – – |
| F | • • – • | Z | – – • • |
| G | – – • | | |
| H | • • • • | | |
| I | • • | | |
| J | • – – – | | |
| K | – • – | 1 | • – – – – |
| L | • – • • | 2 | • • – – – |
| M | – – | 3 | • • • – – |
| N | – • | 4 | • • • • – |
| O | – – – | 5 | • • • • • |
| P | • – – • | 6 | – • • • • |
| Q | – – • – | 7 | – – • • • |
| R | • – • | 8 | – – – • • |
| S | • • • | 9 | – – – – • |
| T | – | 0 | – – – – – |

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

| A | | S | I | M | P | L | E |
|---|---|---|---|---|---|---|---|
| 1010 | 11 | 0100 | 100 | 0101 | 001110 | 01110 | 0000 |

243 bits total

243/480 = 51% of original size

**Huffman encoding:**

- encoding is custom (table required, can only encode characters in original alphabet)

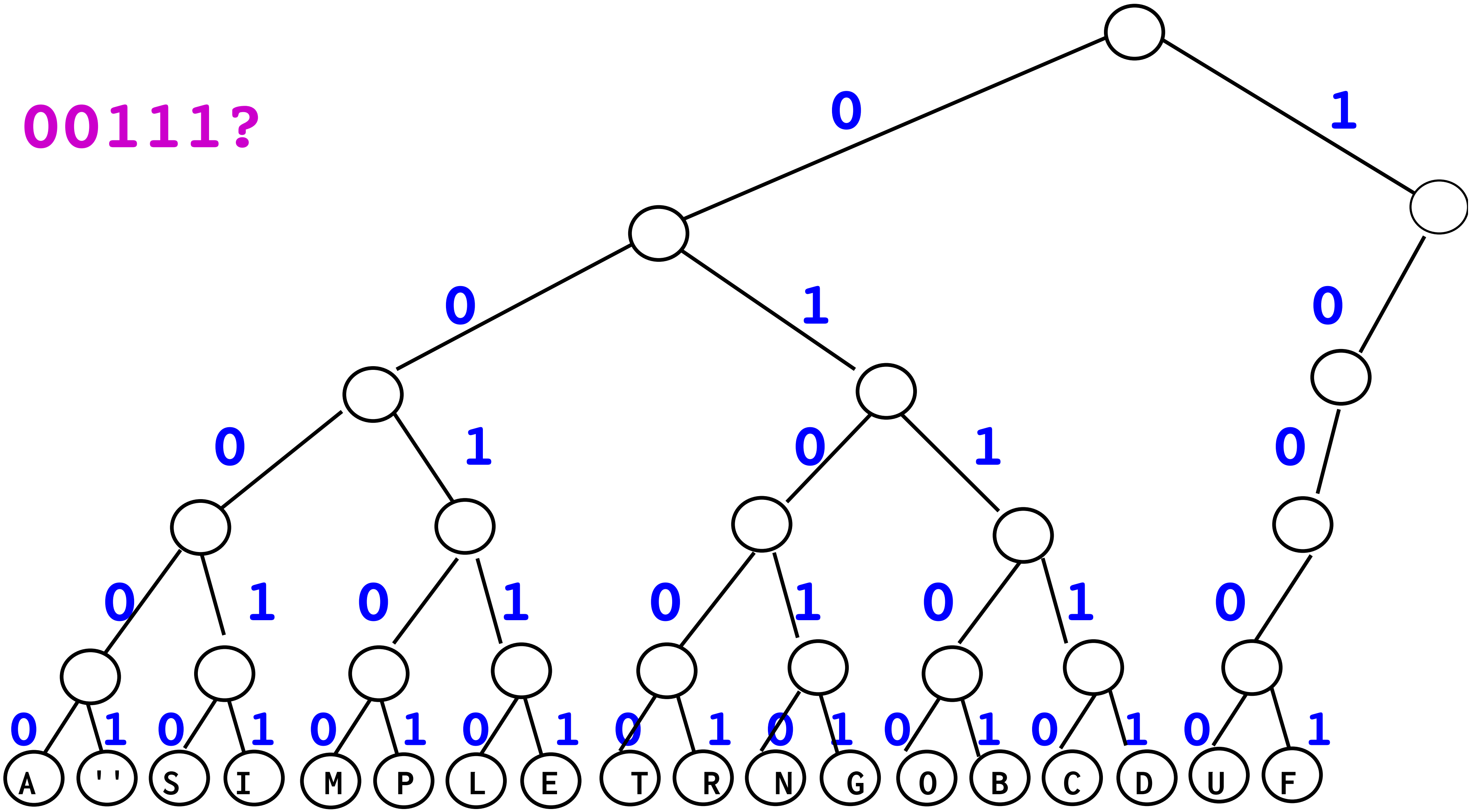
? chars can use different number of bits

\+ **prefix code:** no bit pattern is prefix of any other

\+ encoding is <u>optimal</u> for given input

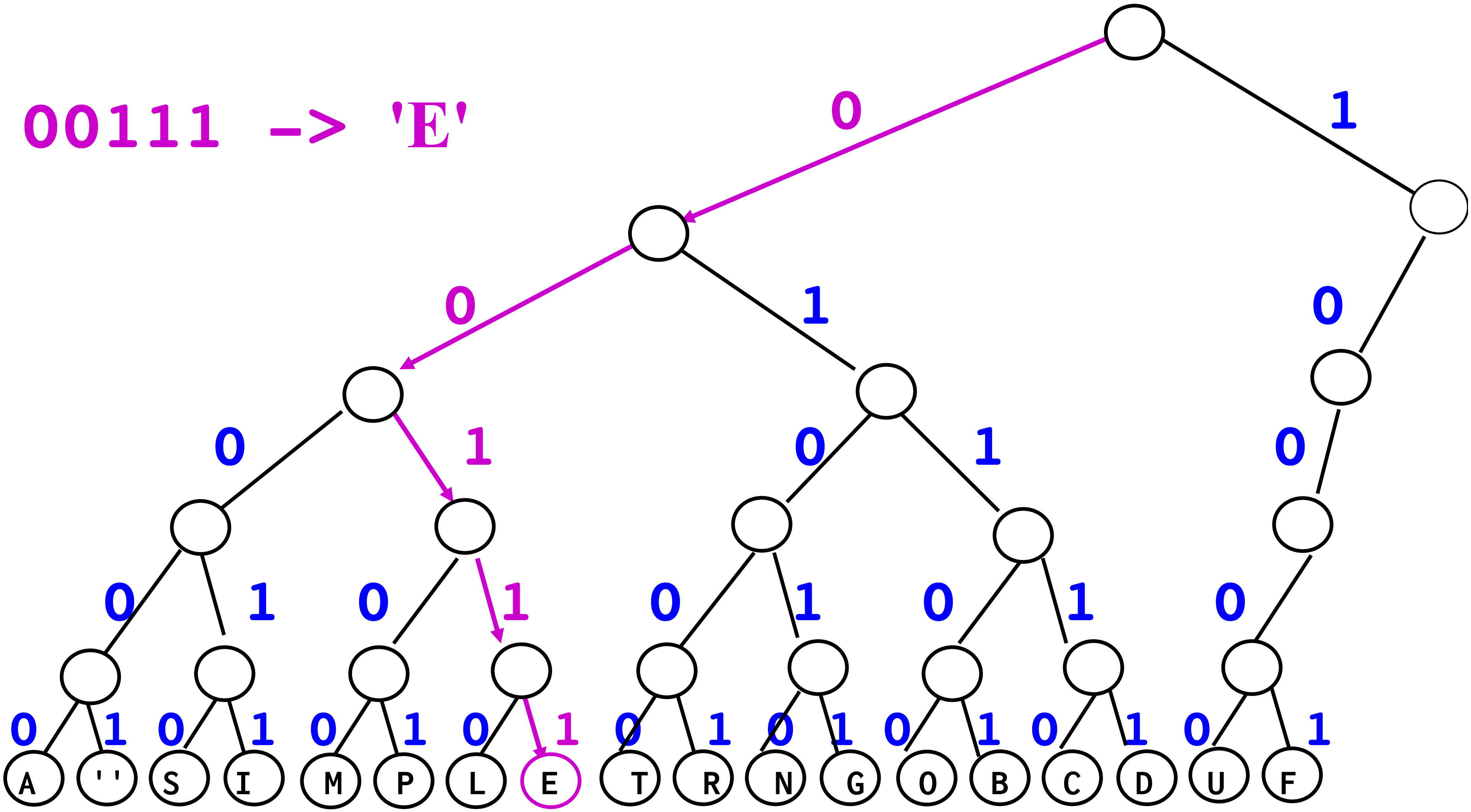
# Encoding trees

00111?



# Encoding trees

00111 -> 'E'

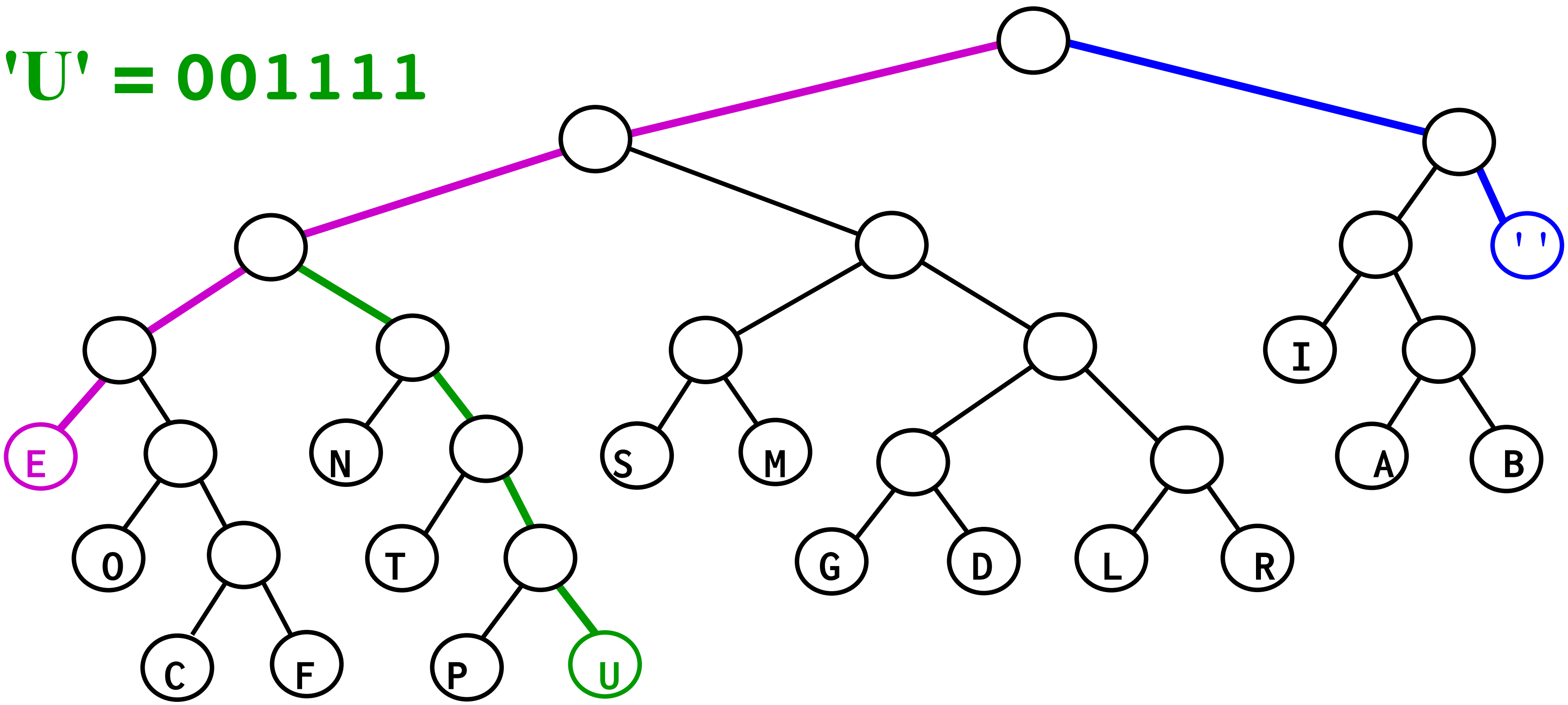


# Huffman encoding tree

'E' = 0000

' ' = 11

'U' = 001111



# Building an optimal tree

**Start:** Create leaf node for each char, weight = frequency
$n$ trees to start ($n$ is num characters in alphabet)

**Step:** Repeatedly join two trees with smallest weights
into tree with weight equal to sum
$\rightarrow$ $n$-1 trees left

**Stop:** When all joined into one combined tree

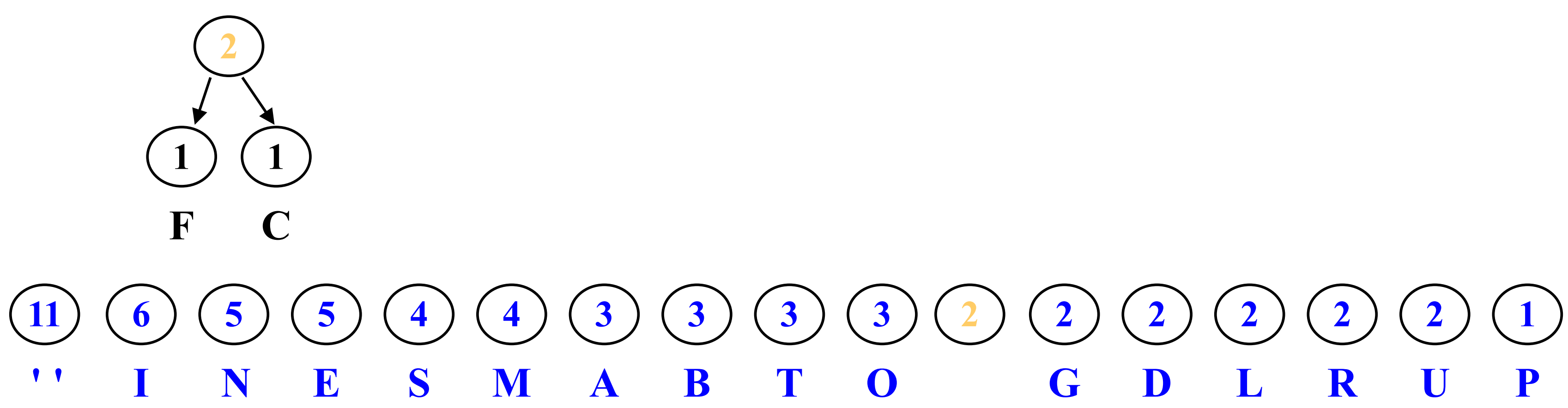
# Building an optimal tree

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

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | 6 | 5 | 5 | 4 | 4 | 3 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 |
| ' ' | I | N | E | S | M | A | B | T | O | G | D | L | R | U | P | F | C |

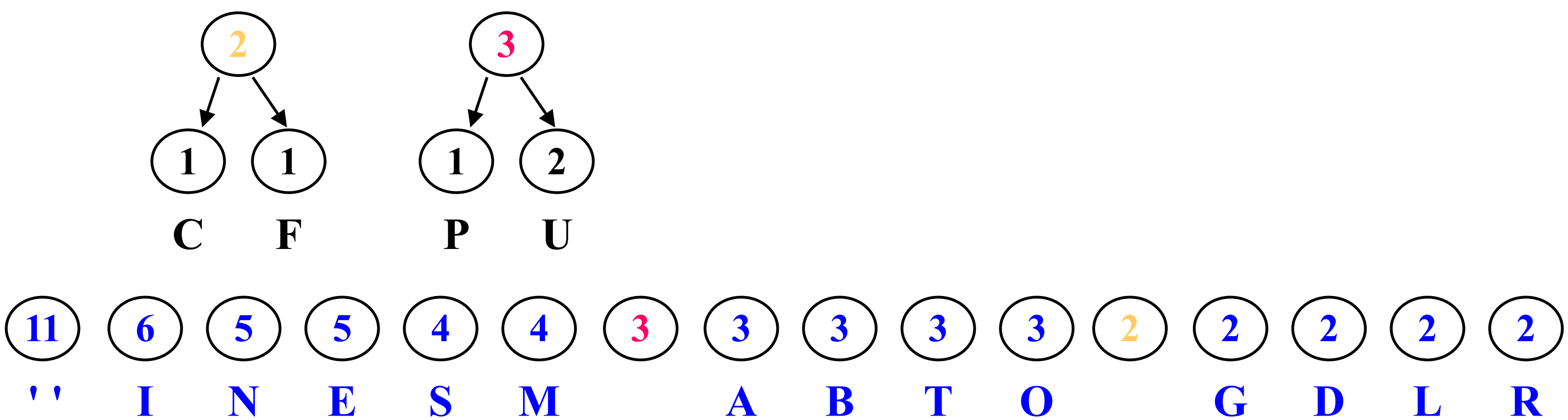
# Building an optimal tree

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



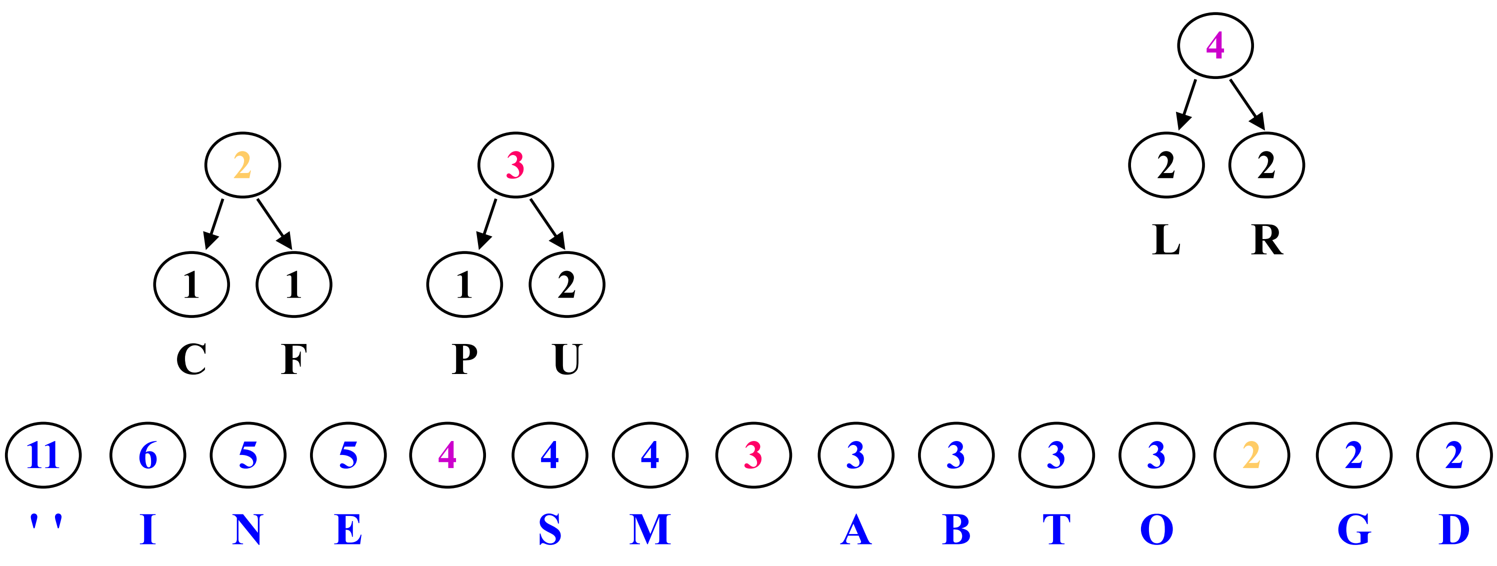
# Building an optimal tree

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



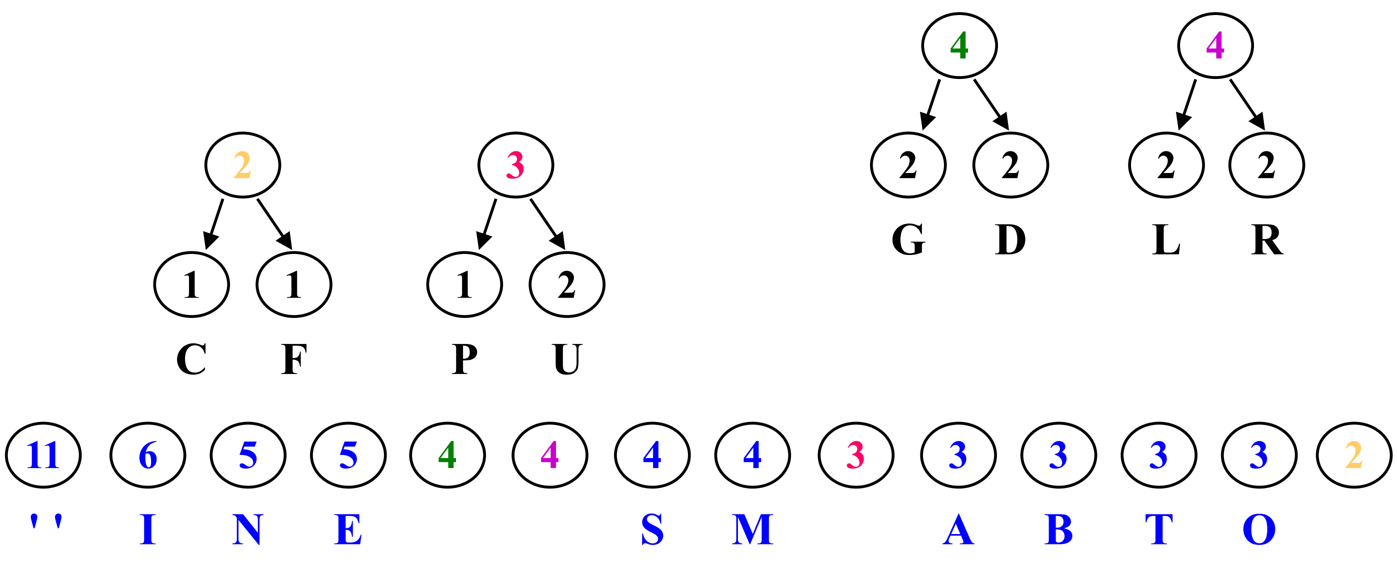
# Building an optimal tree

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



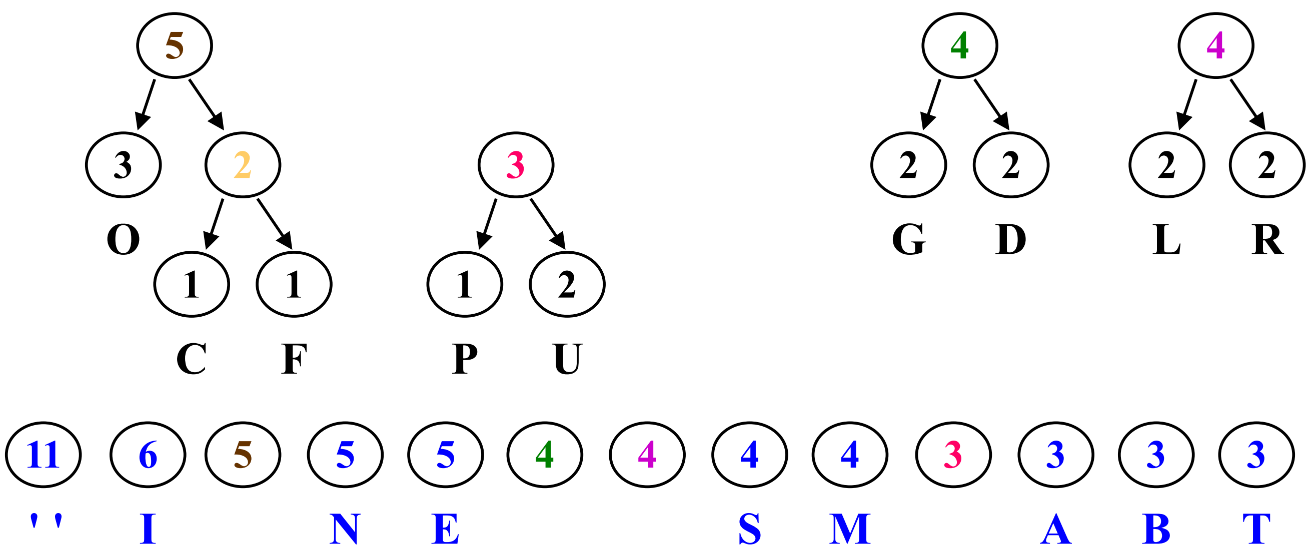
# Building an optimal tree

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



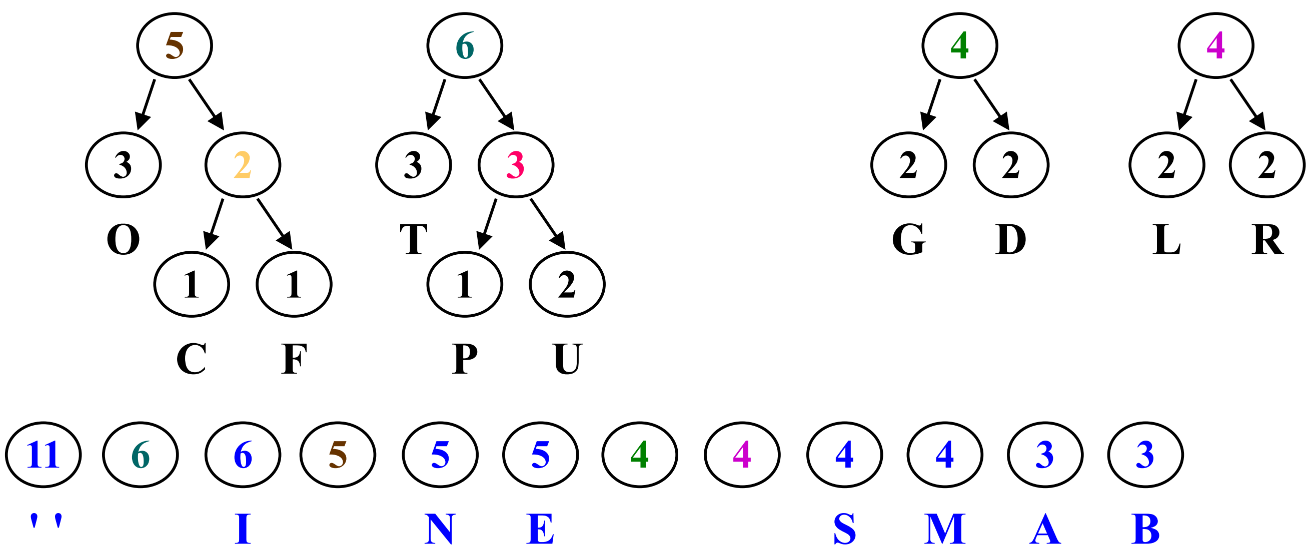
# Building an optimal tree

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



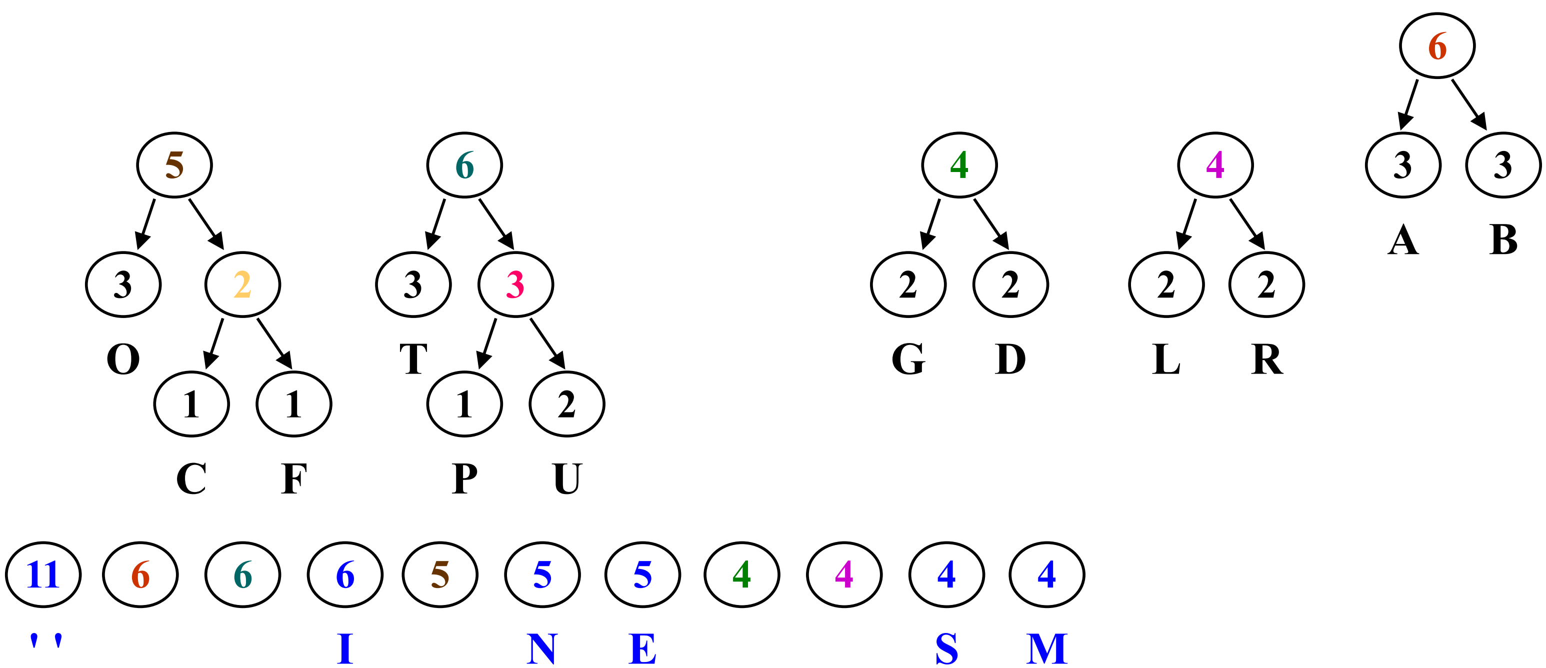
# Building an optimal tree

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



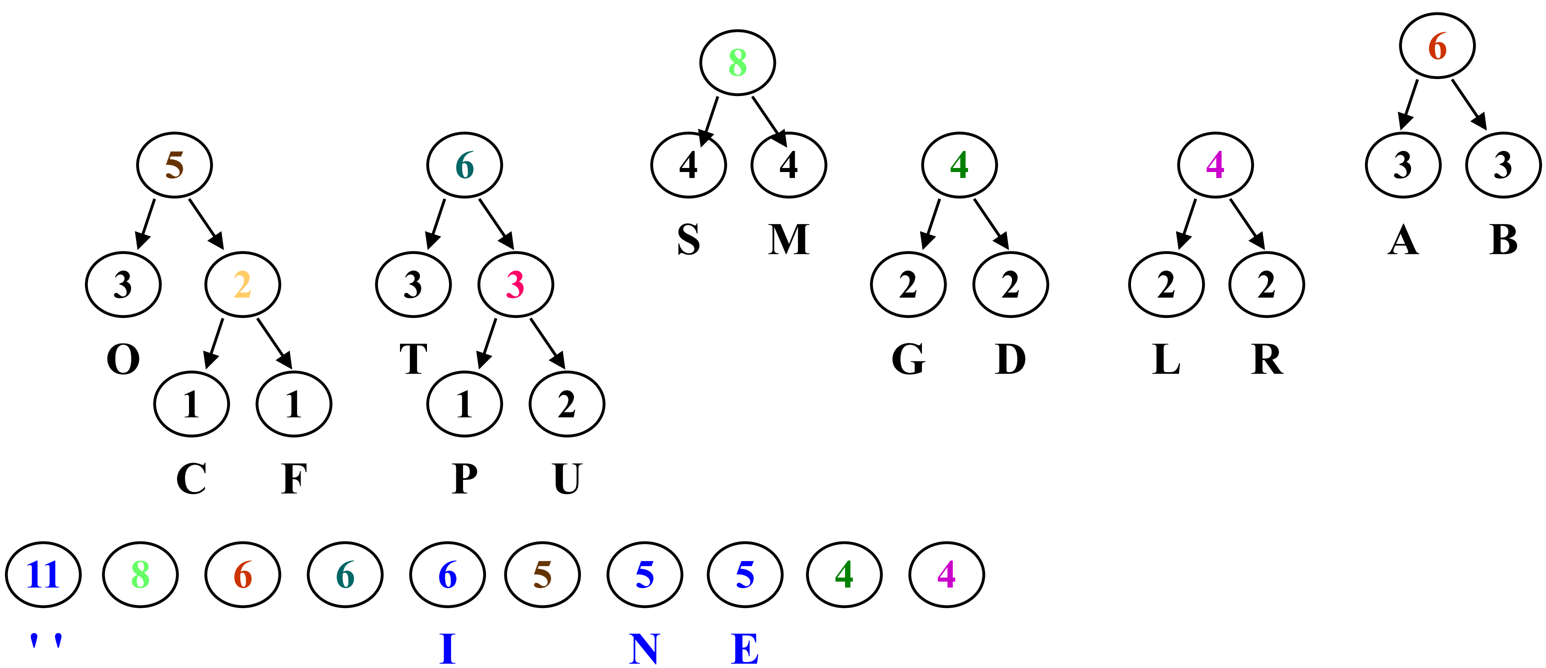
# Building an optimal tree

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



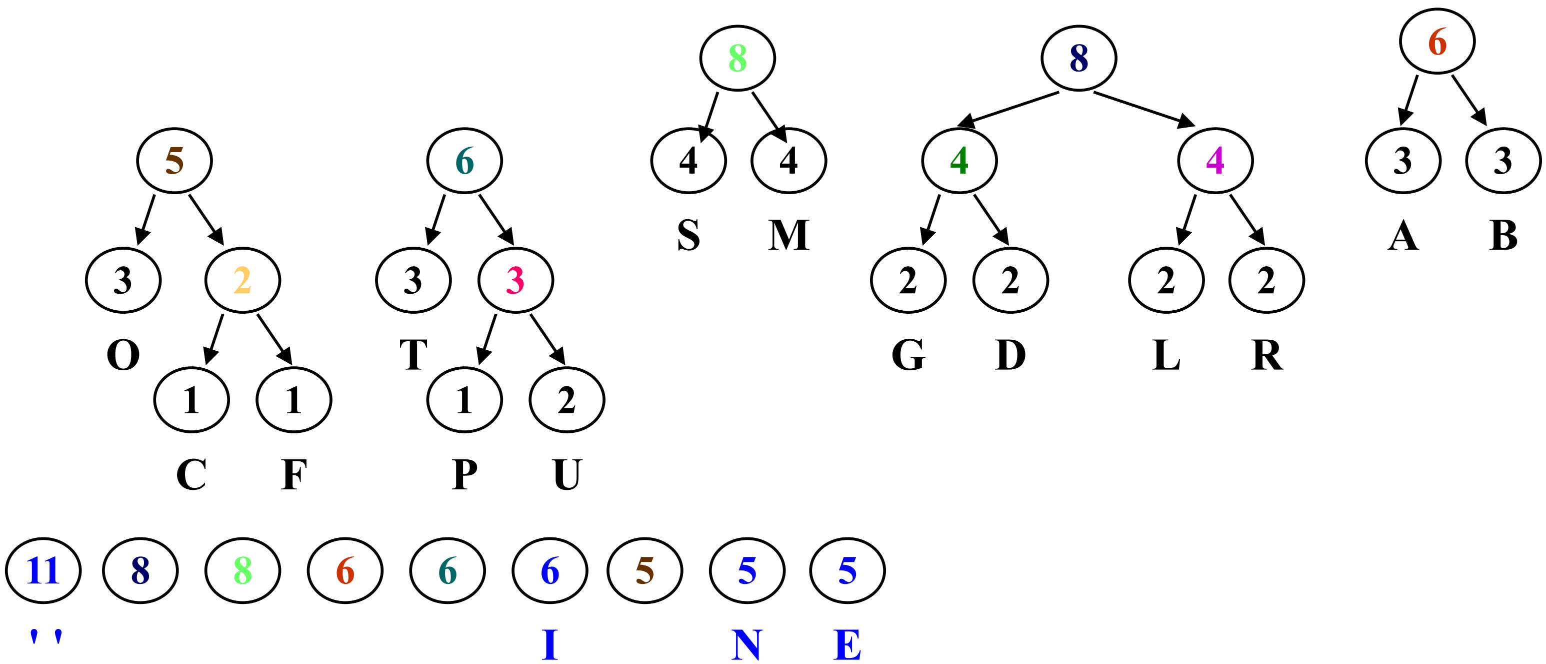
# Building an optimal tree

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



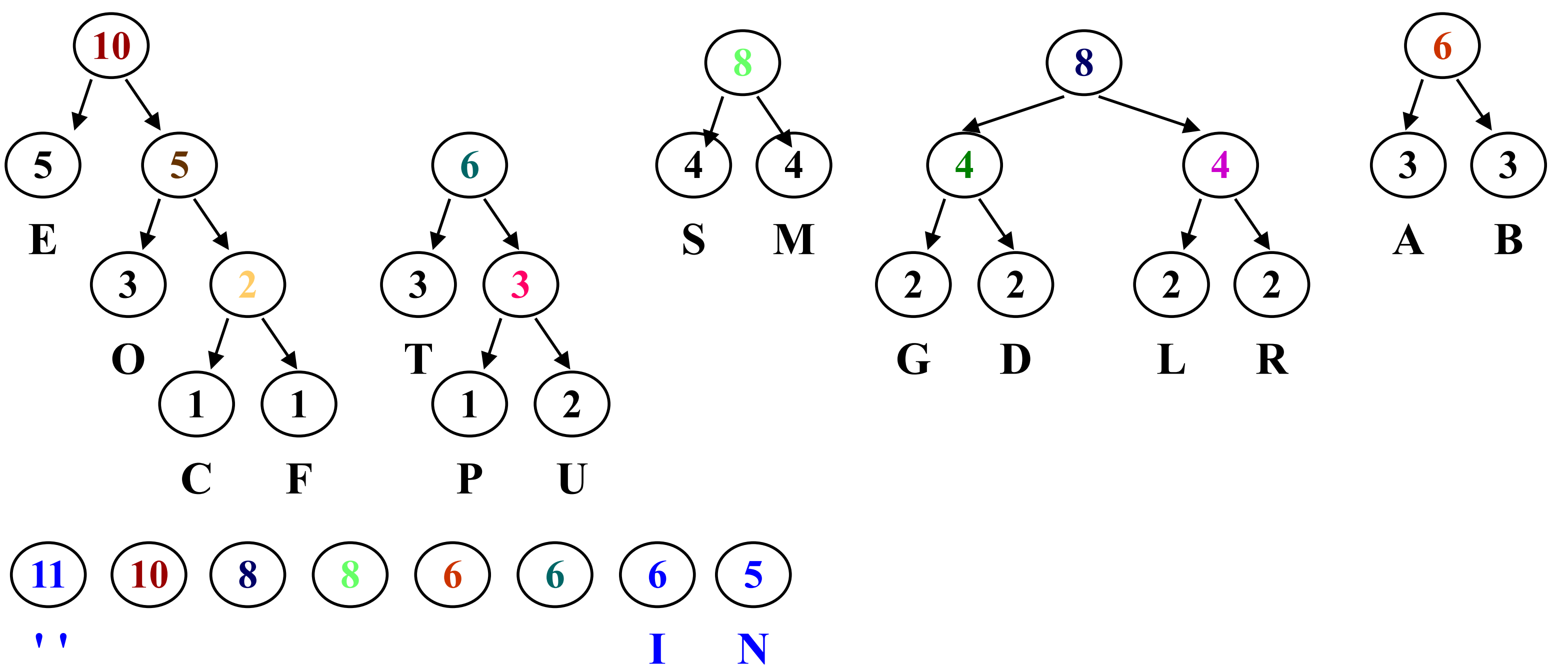
# Building an optimal tree

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



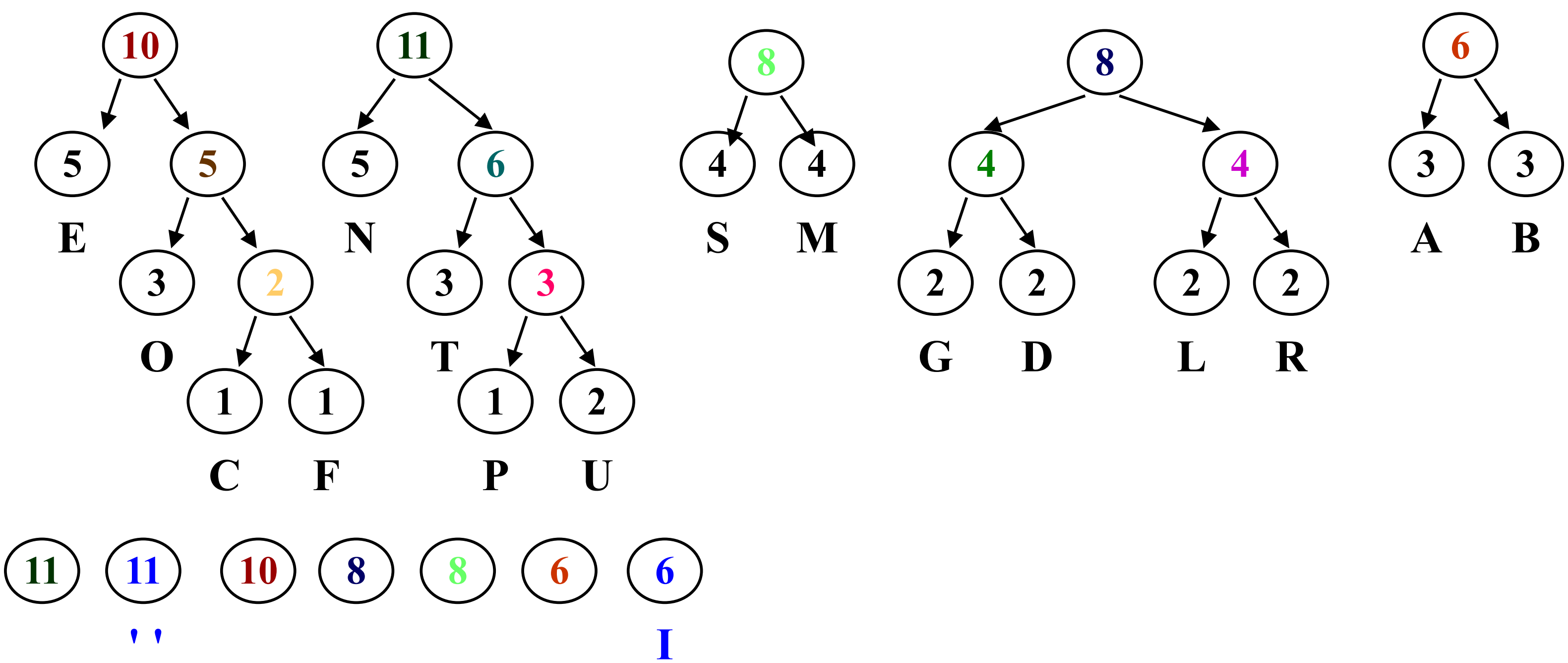
# Building an optimal tree

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



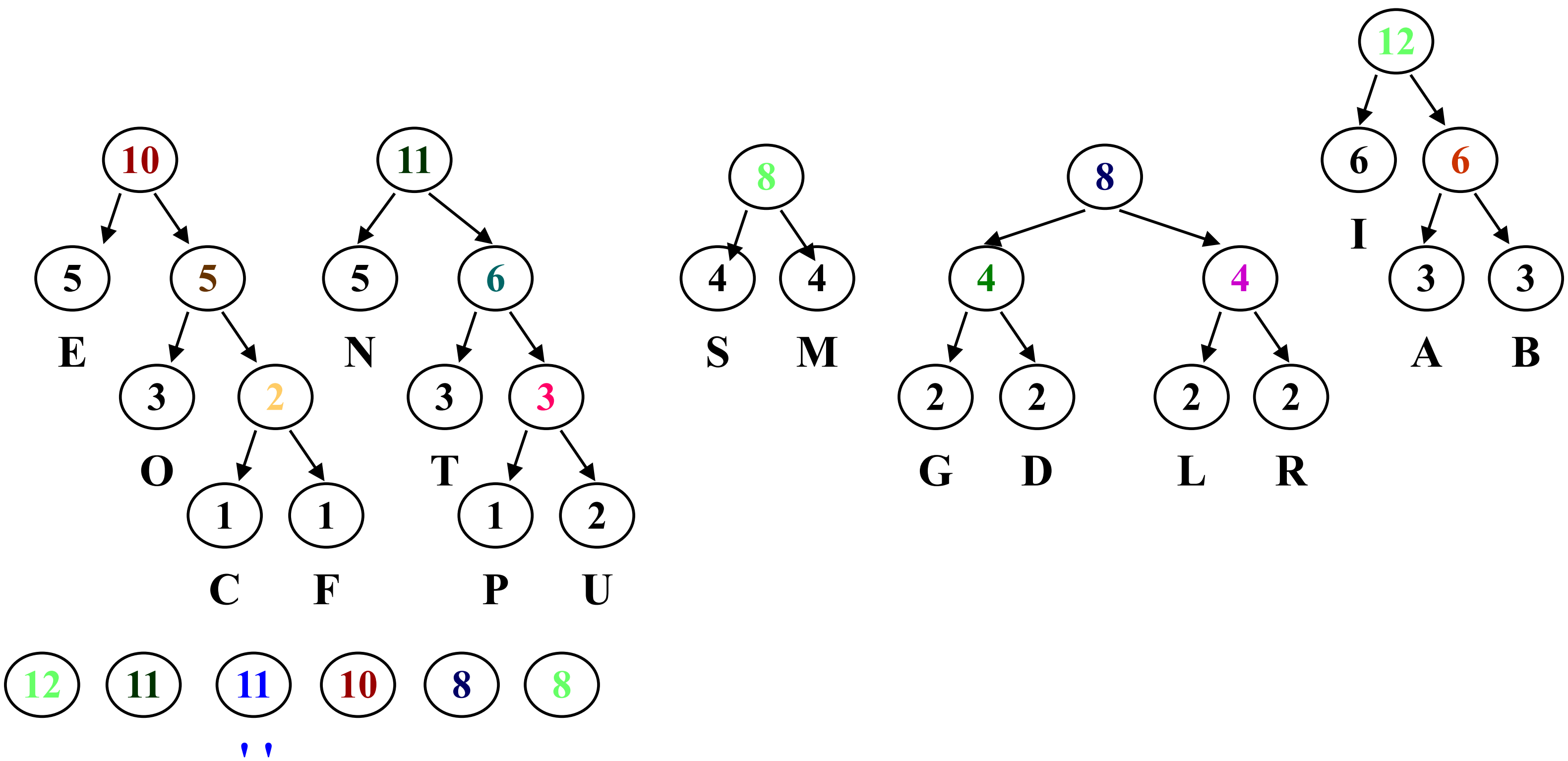
# Building an optimal tree

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



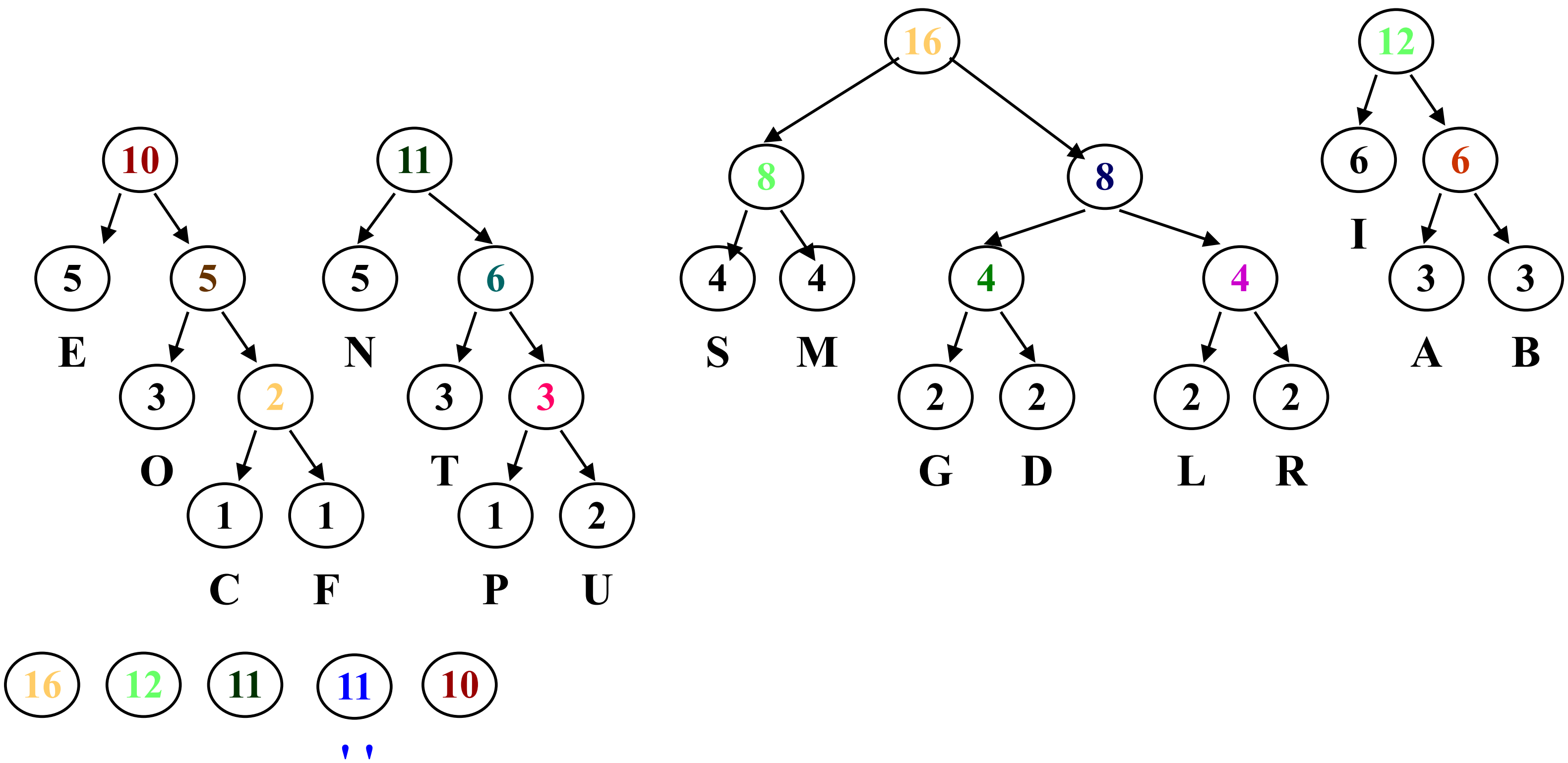
# Building an optimal tree

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



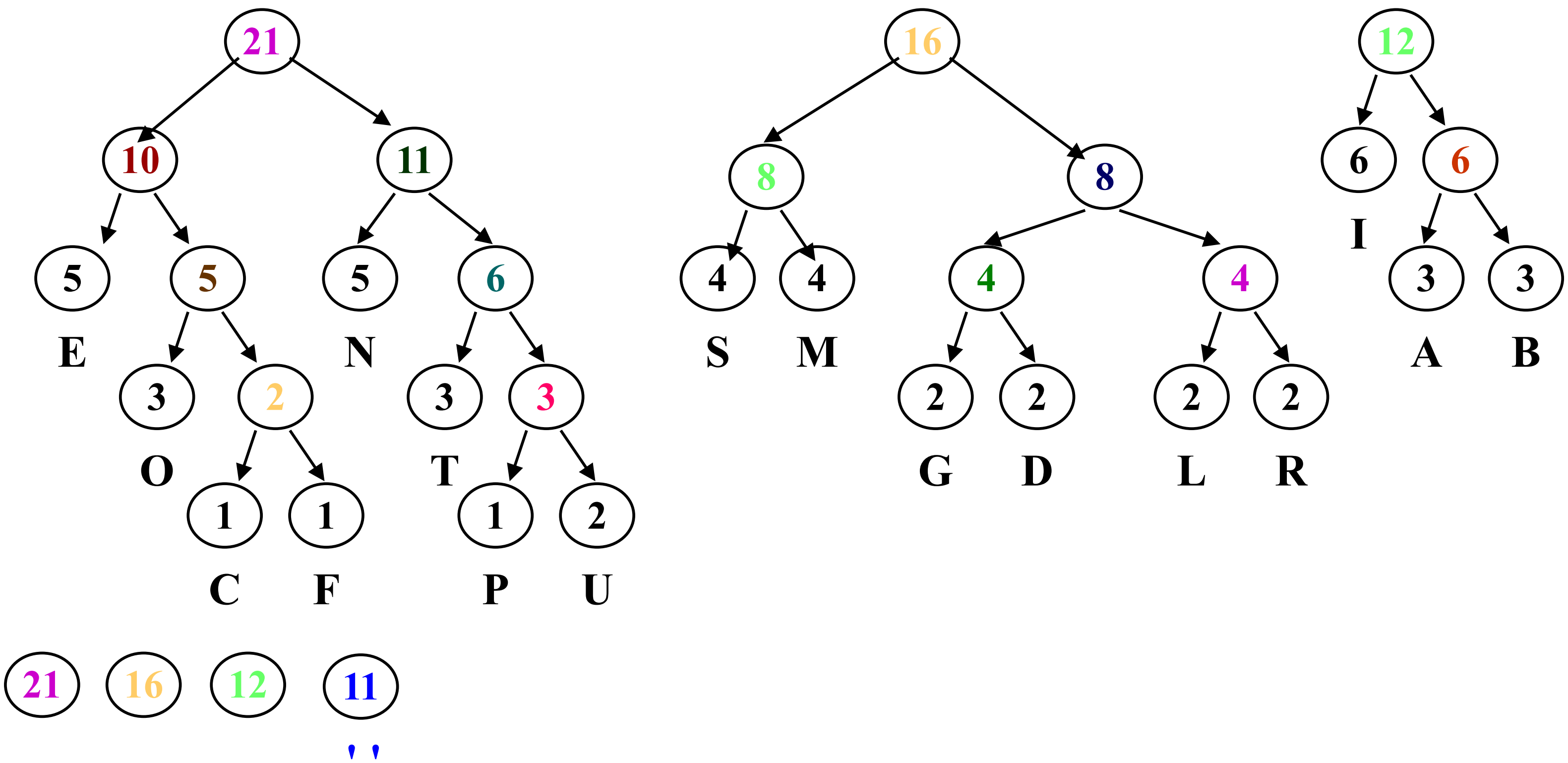
# Building an optimal tree

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



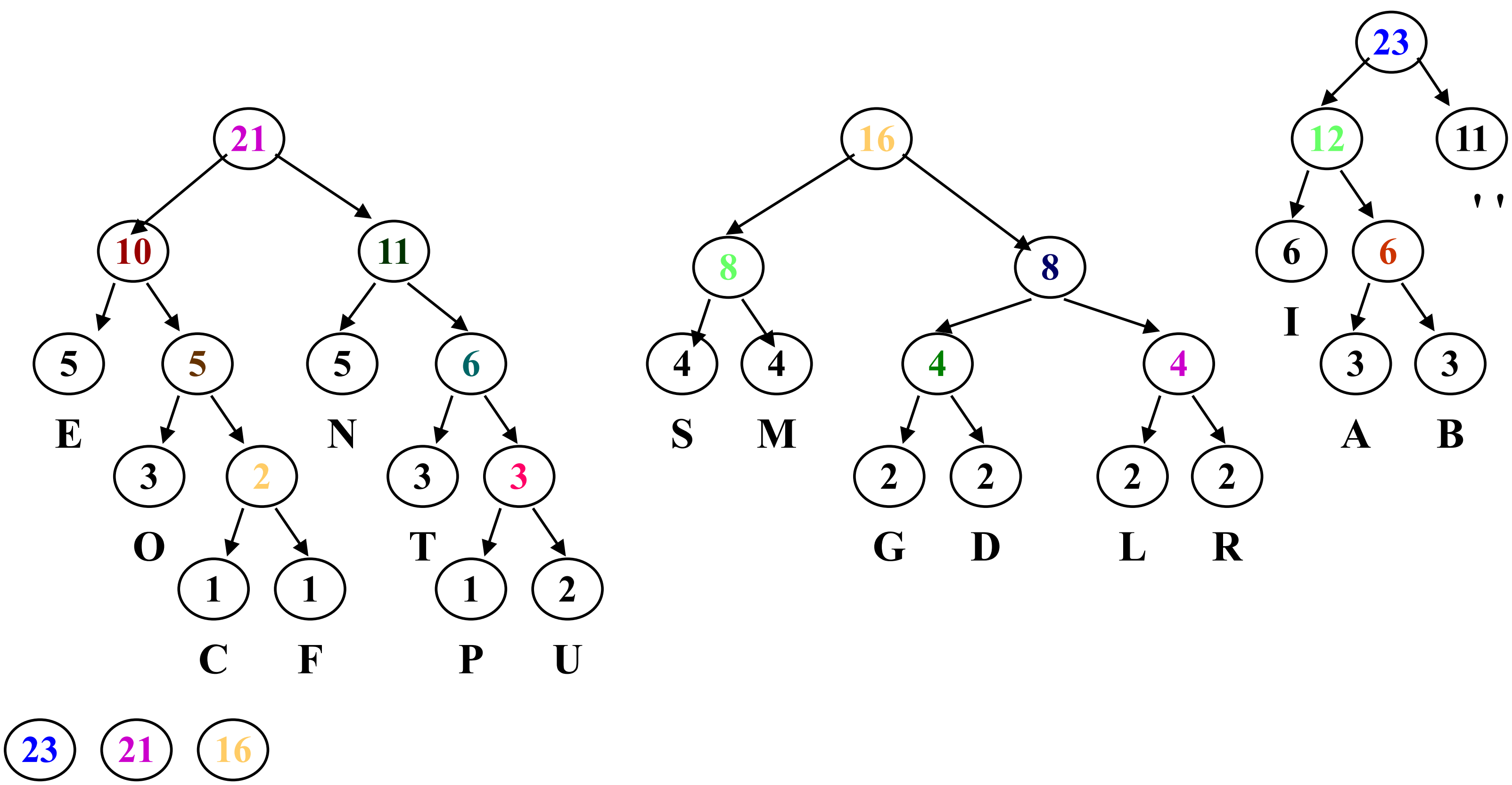
# Building an optimal tree

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



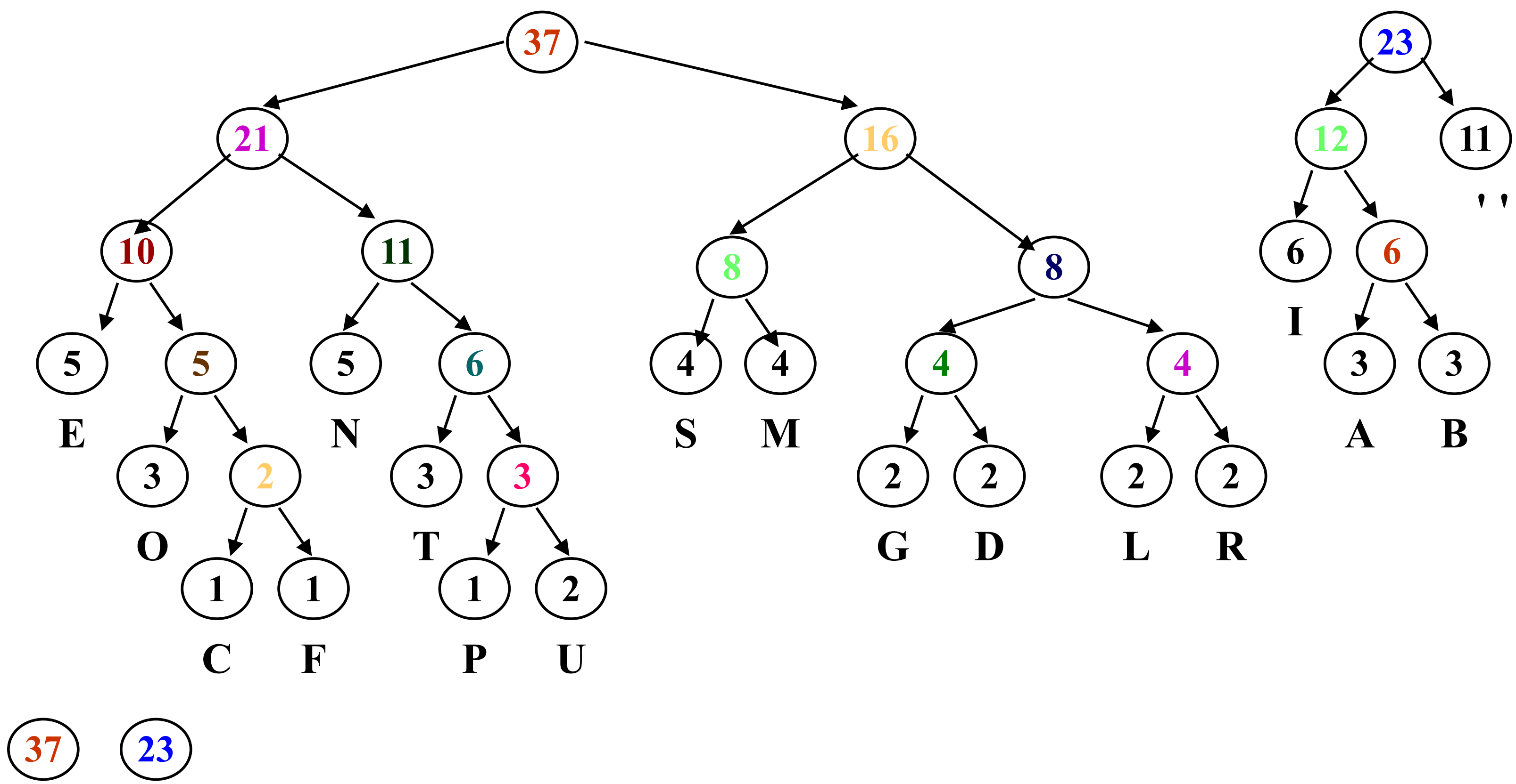
# Building an optimal tree

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



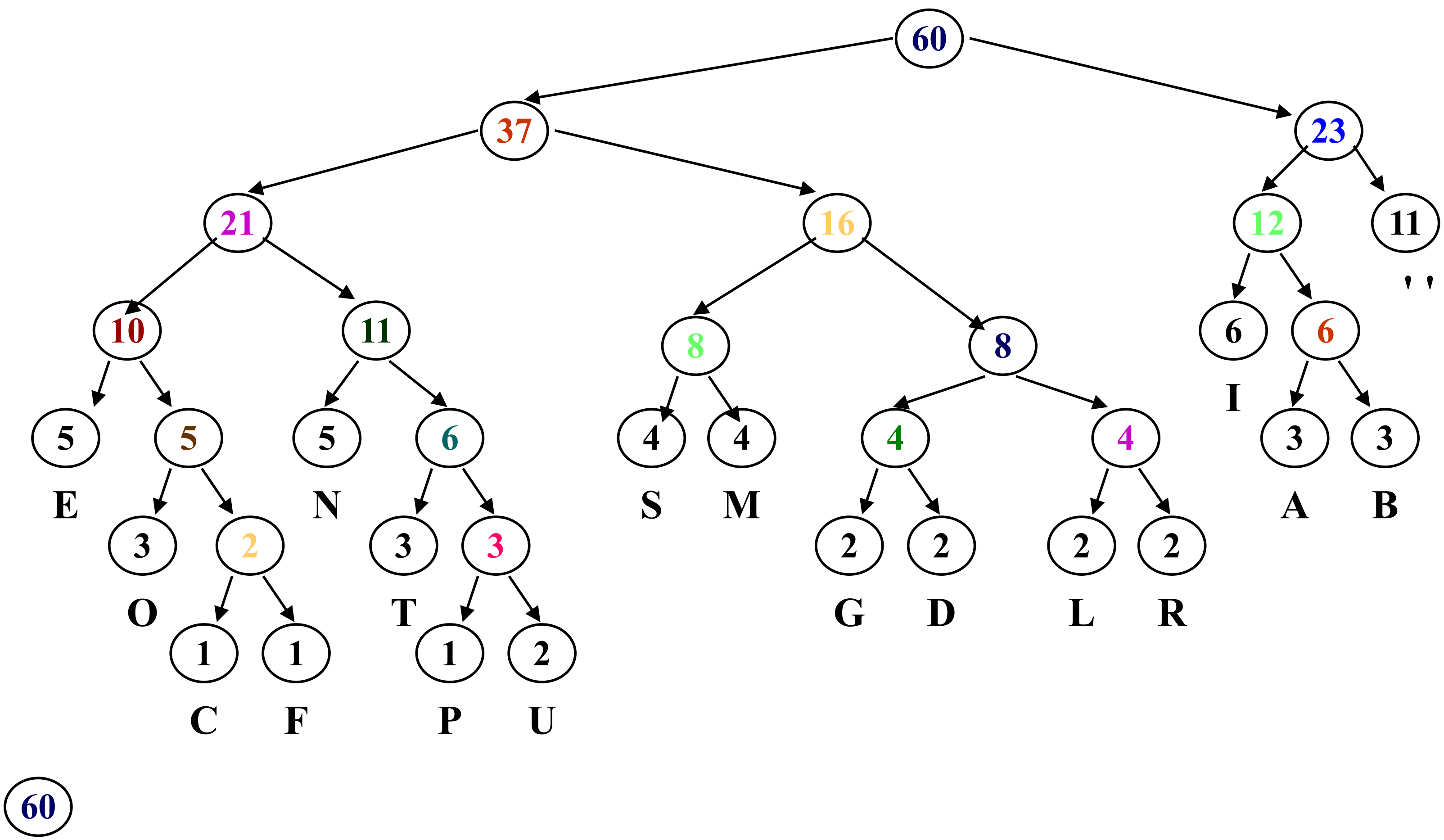
# Building an optimal tree

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



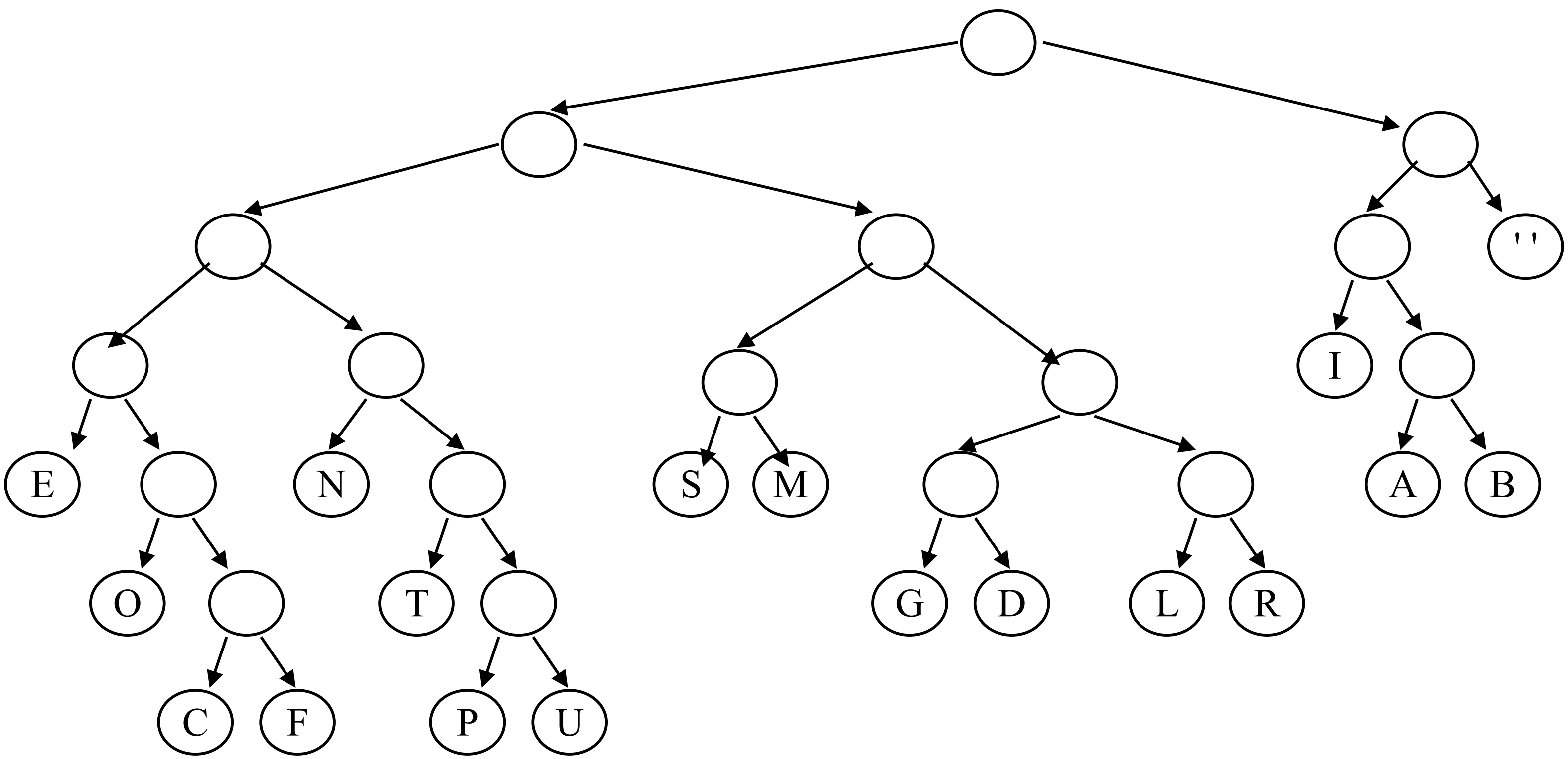
# Building an optimal tree— done!

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



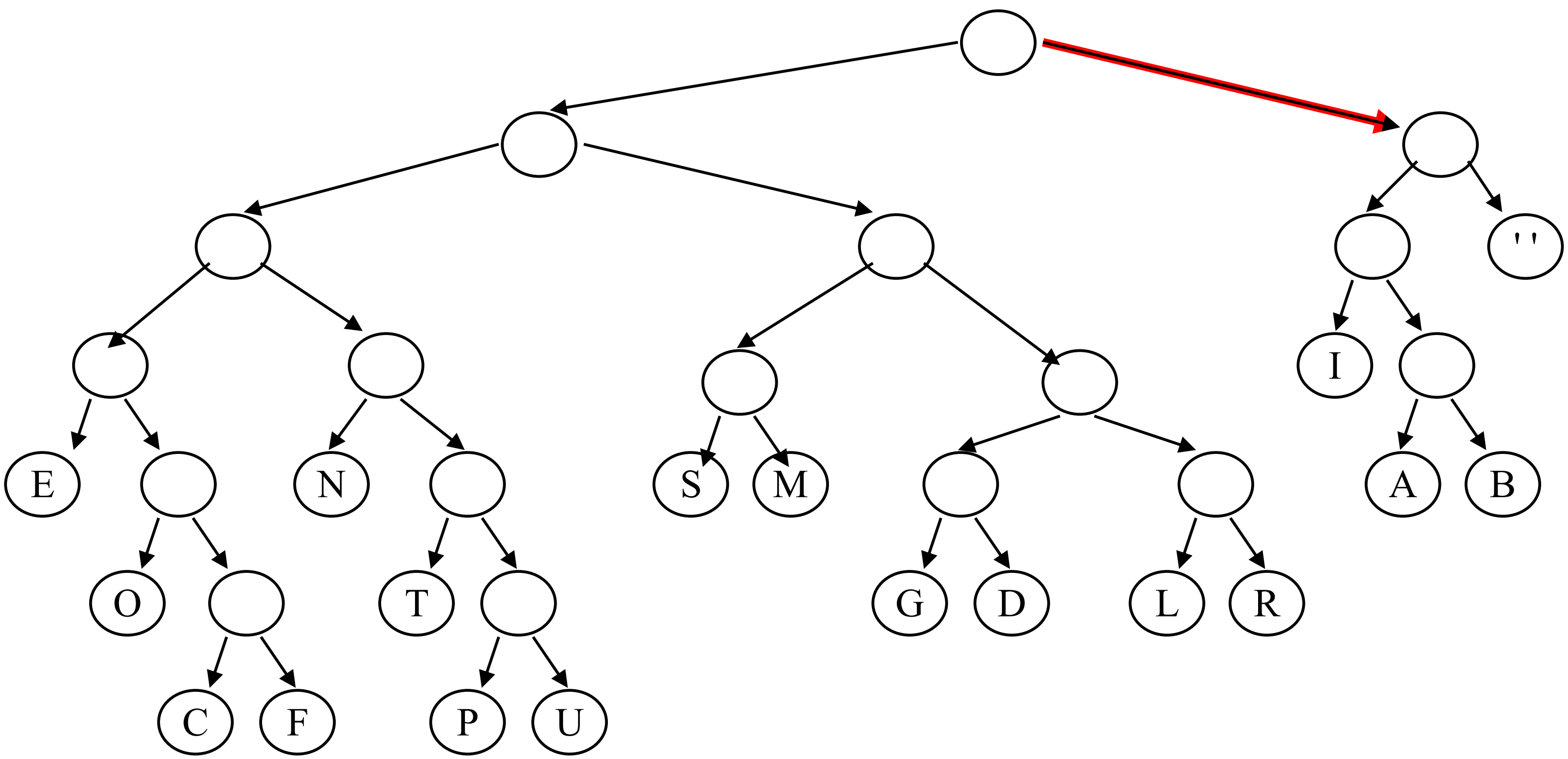
# Decoding a message

**1011000010100011011000110101001110**



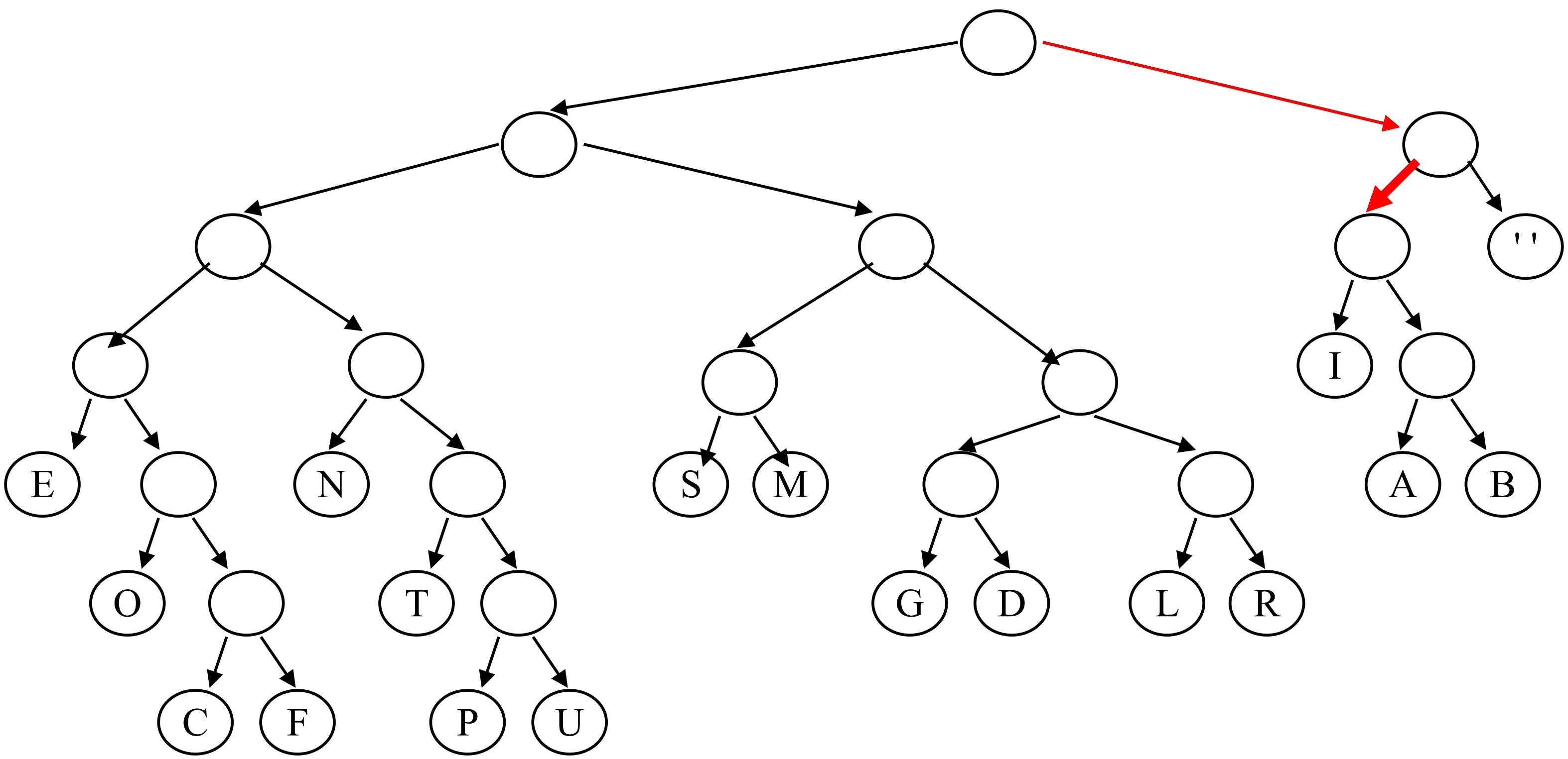
# Decoding a message

**1011000010100011011000110101001110**



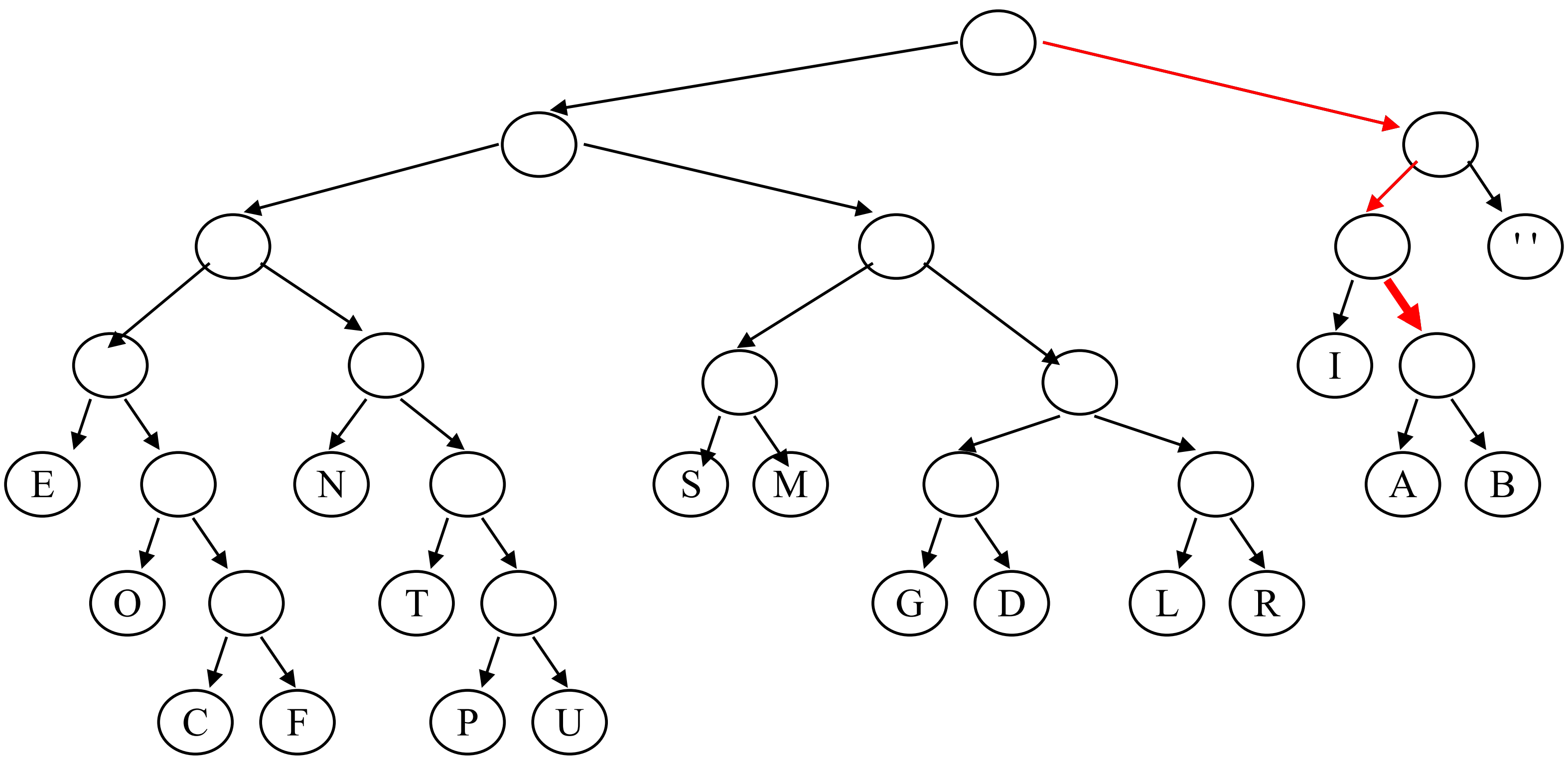
# Decoding a message

**10110000101000110110001101010011110**



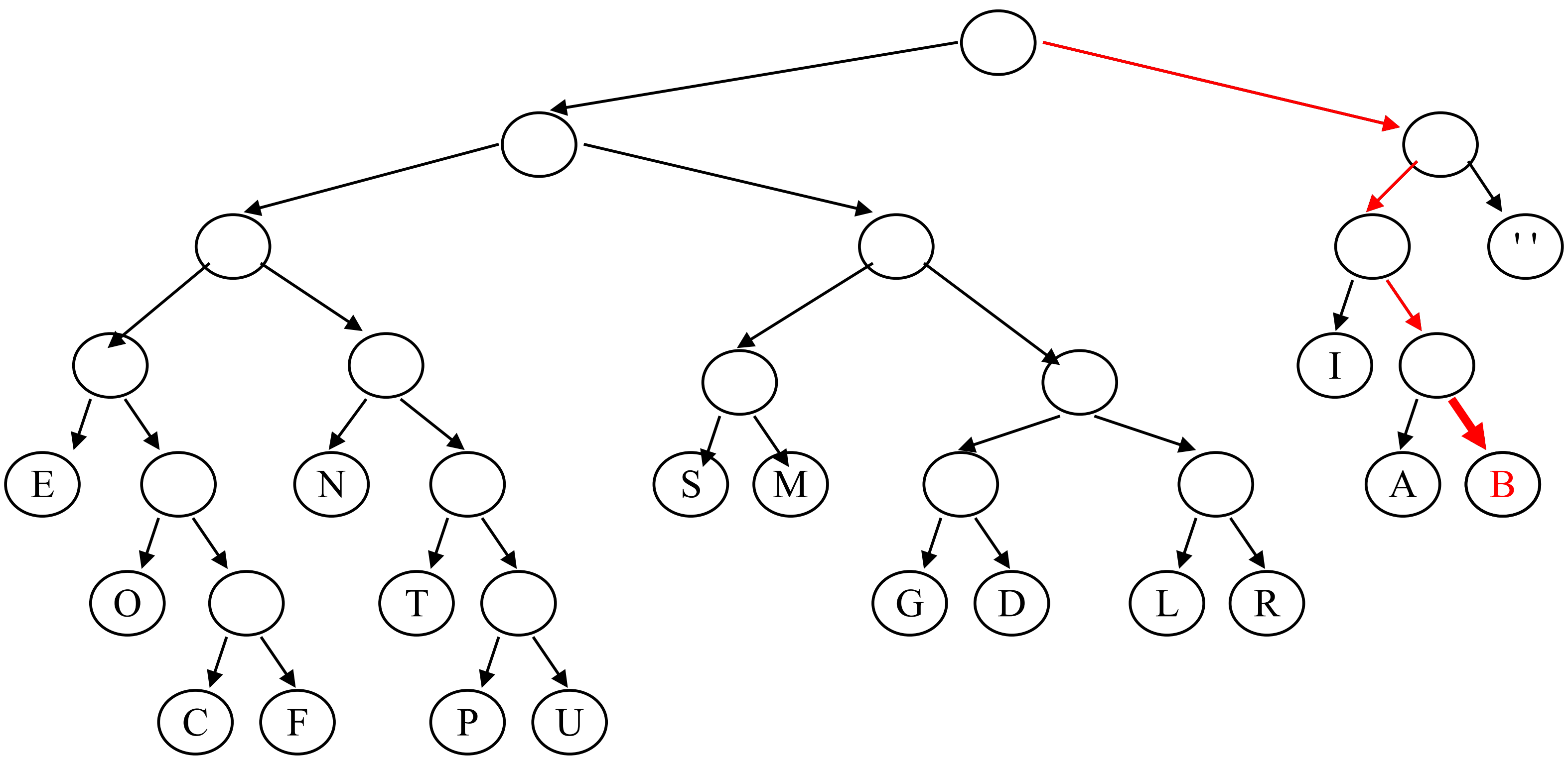
# Decoding a message

**1011000010100011011000110101001110**



# Decoding a message

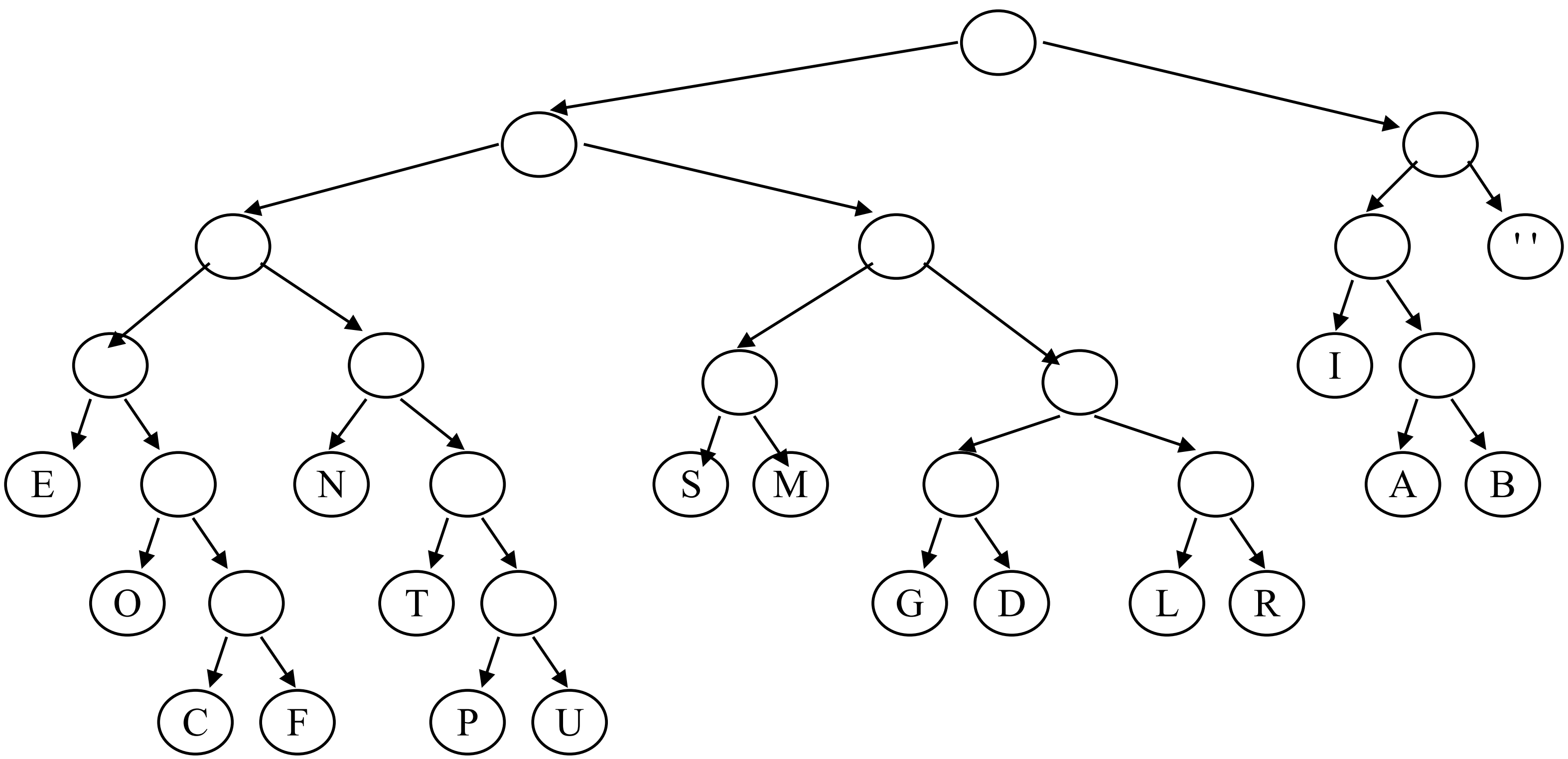
**1011000010100011011000110101001110**




B

# Decoding a message

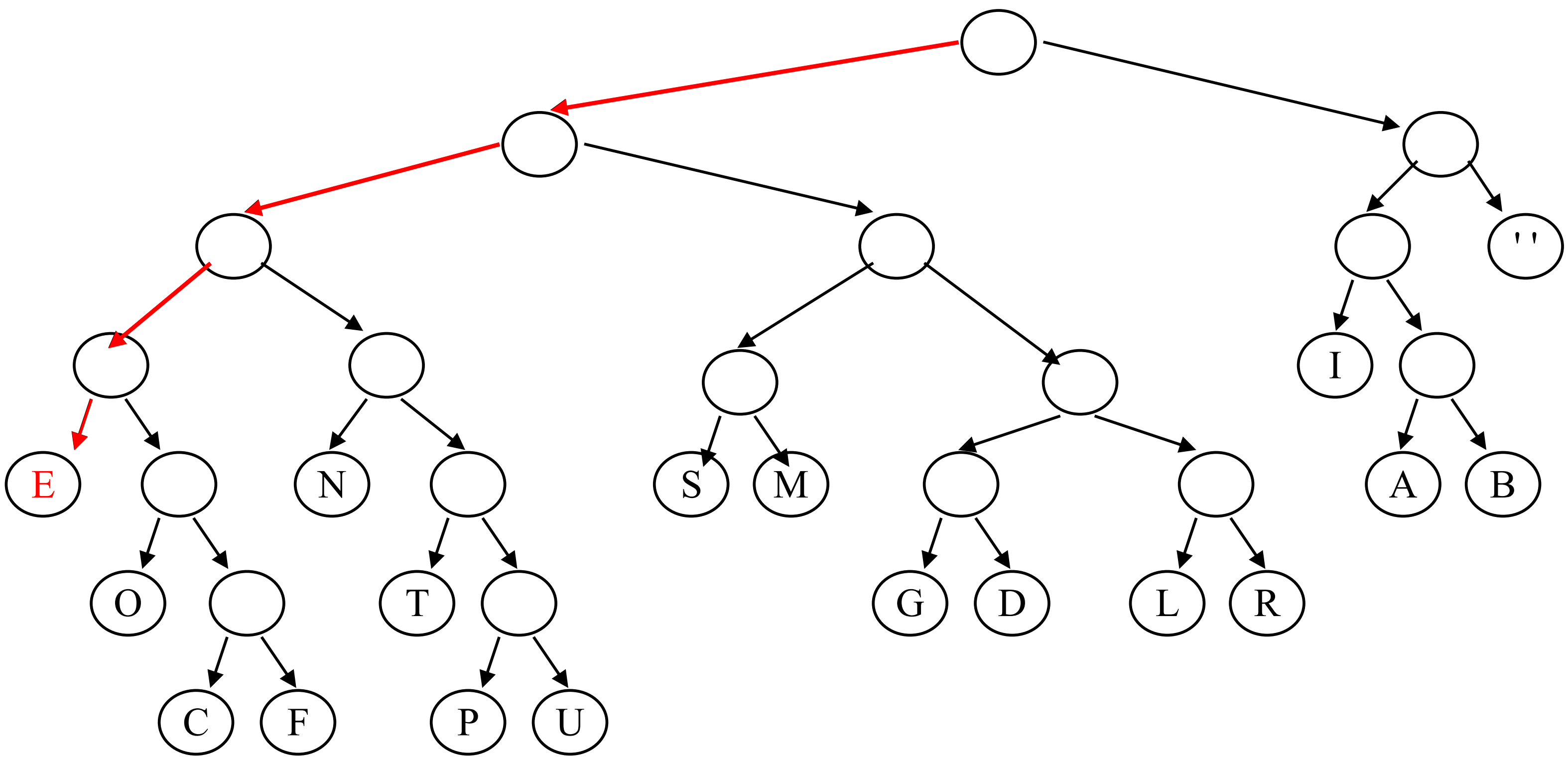
**1011000001010001101100011010100111 0**




B

# Decoding a message

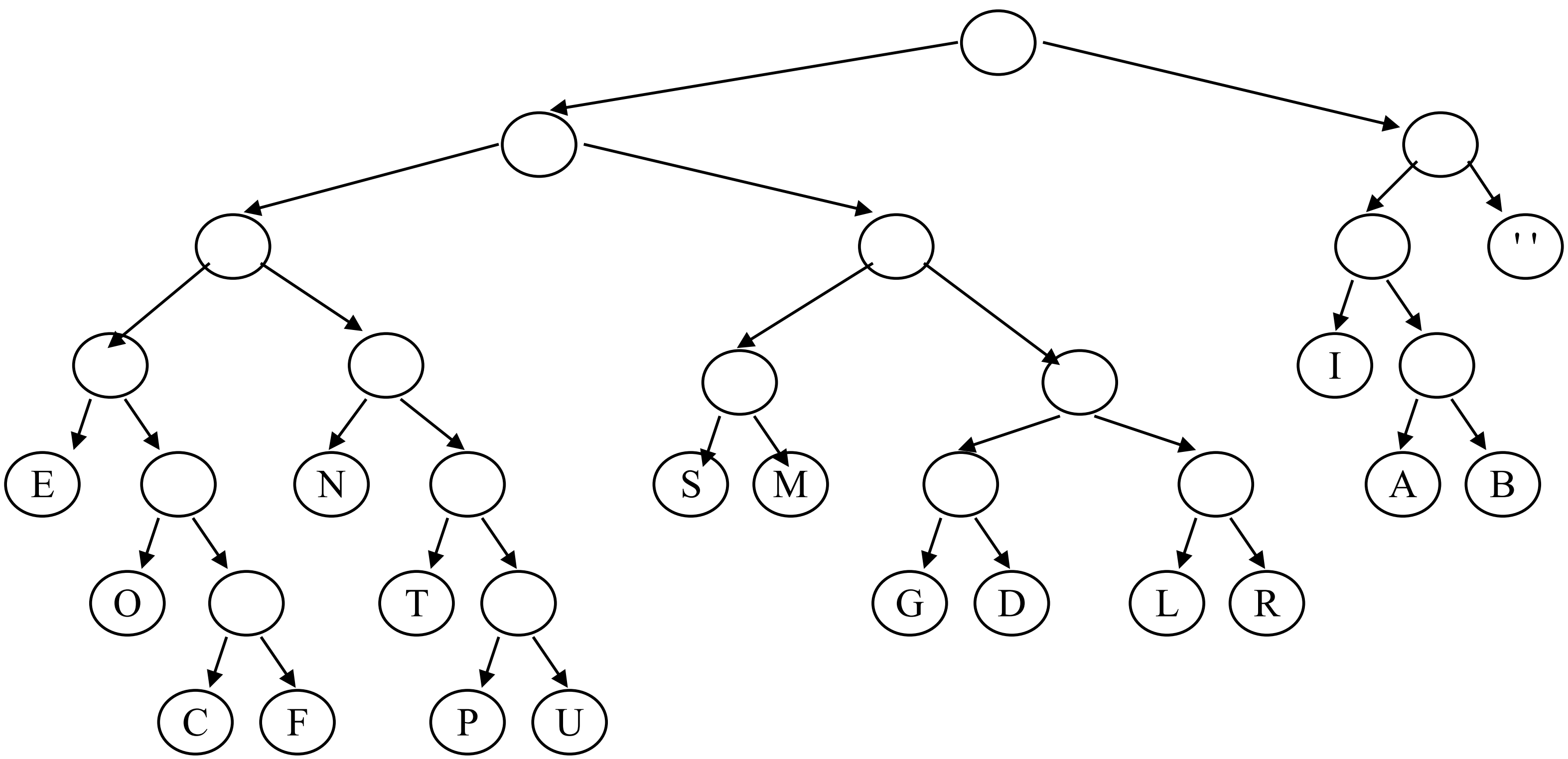
**1011000010100011011000110101001110**




BE

# Decoding a message

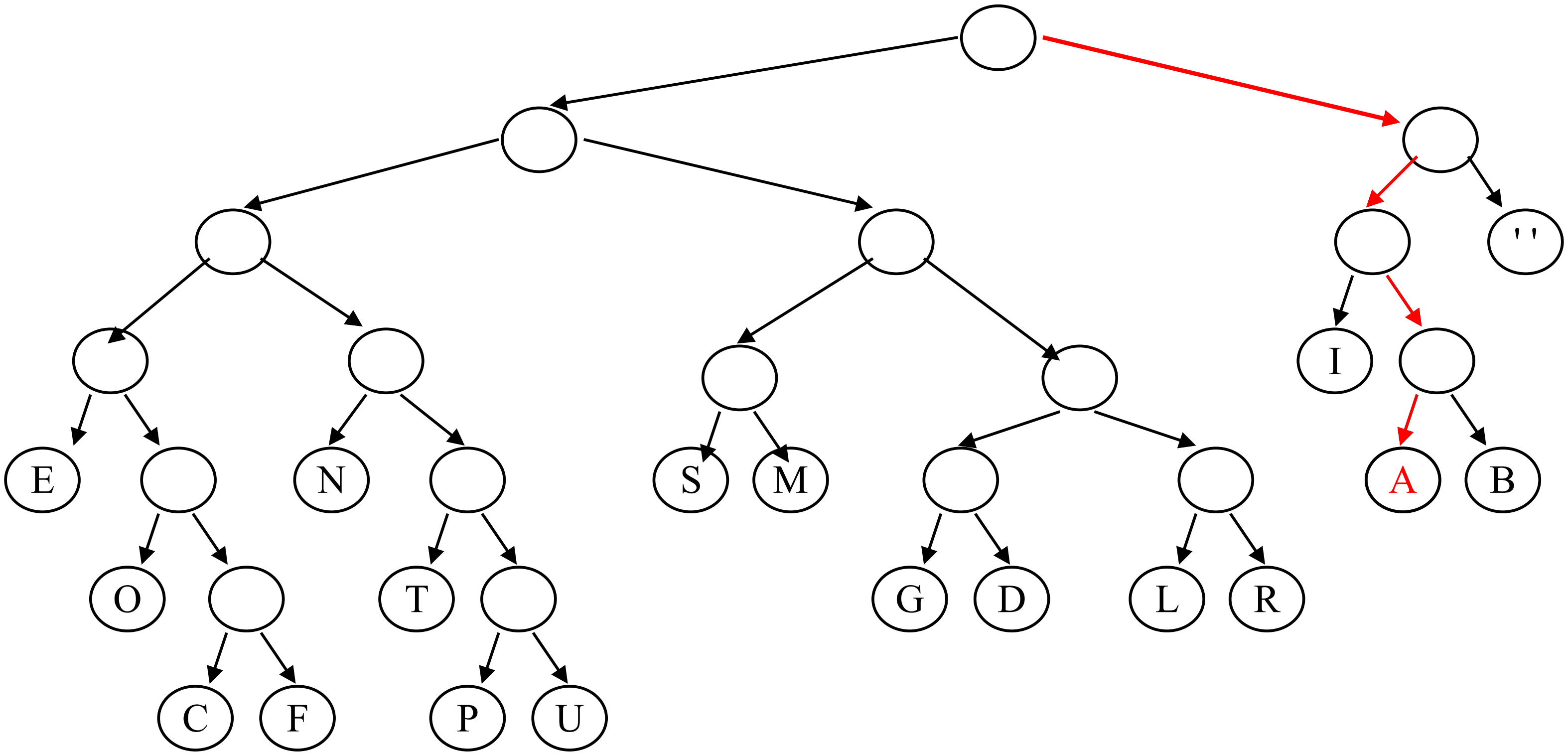
1011000010100011011000110101001110




BE

# Decoding a message

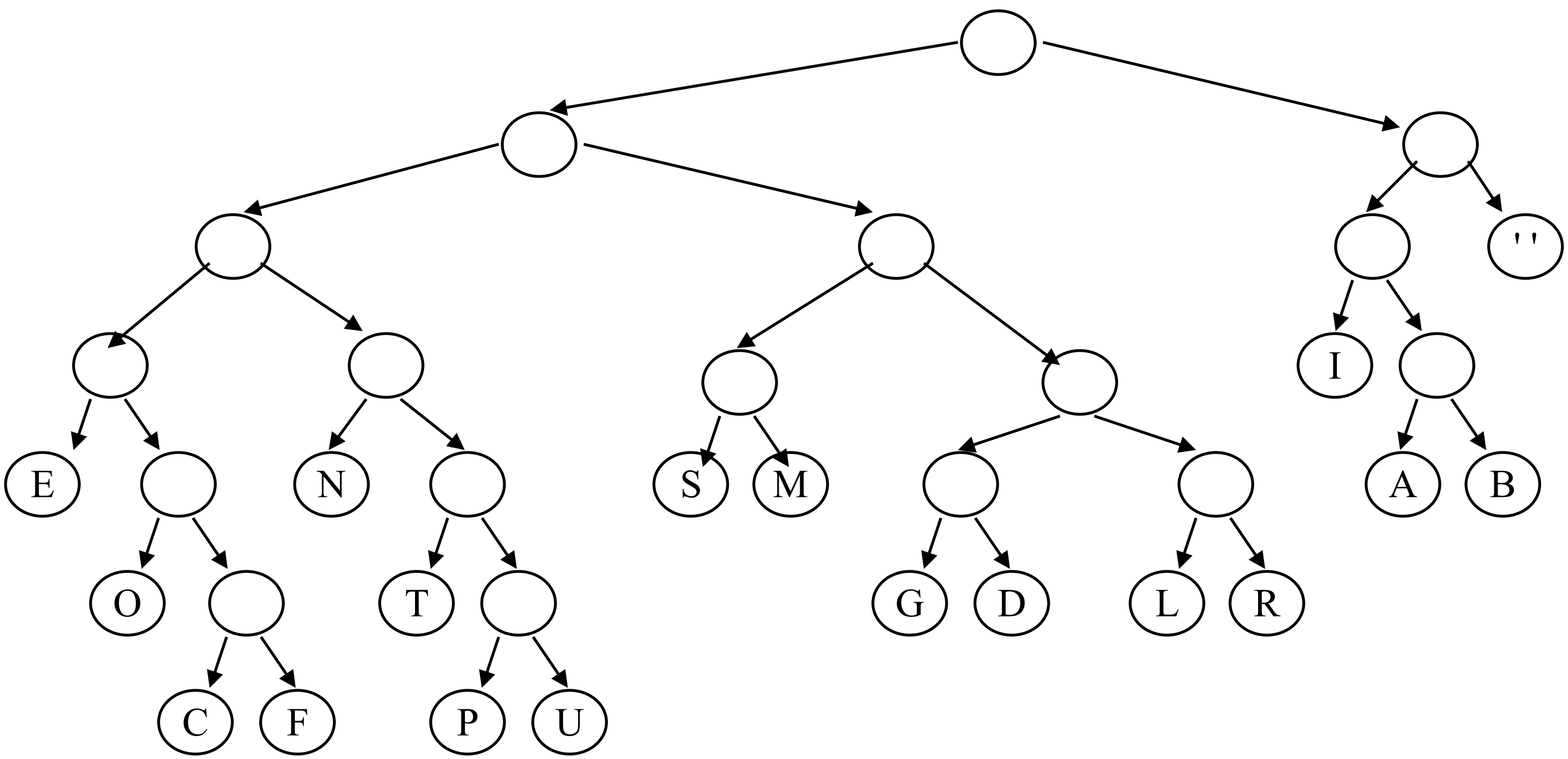
**10110000101000110110001101010011110**




BEA

# Decoding a message

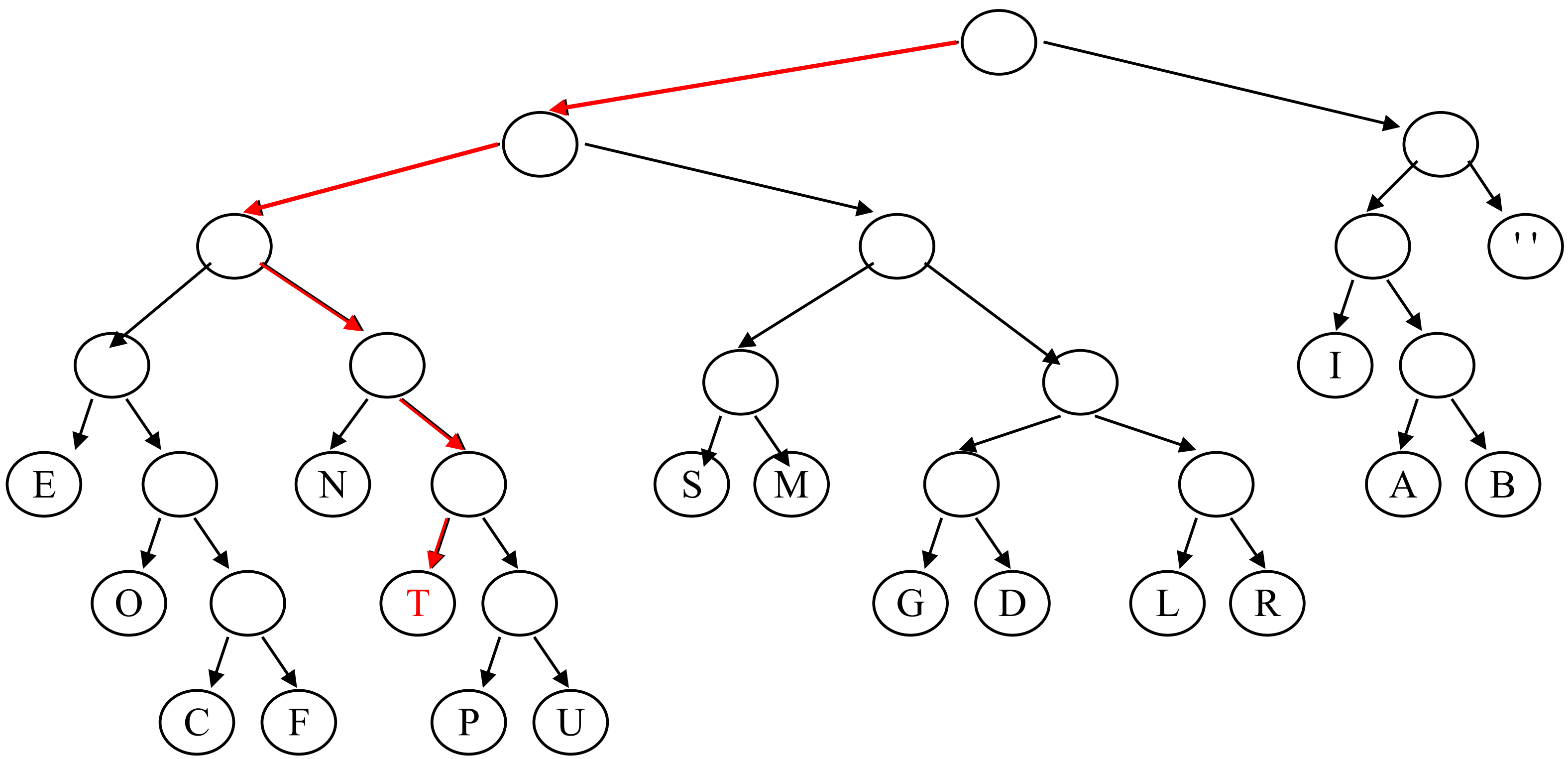
101100001010**0011011000110101001110**




BEA

# Decoding a message

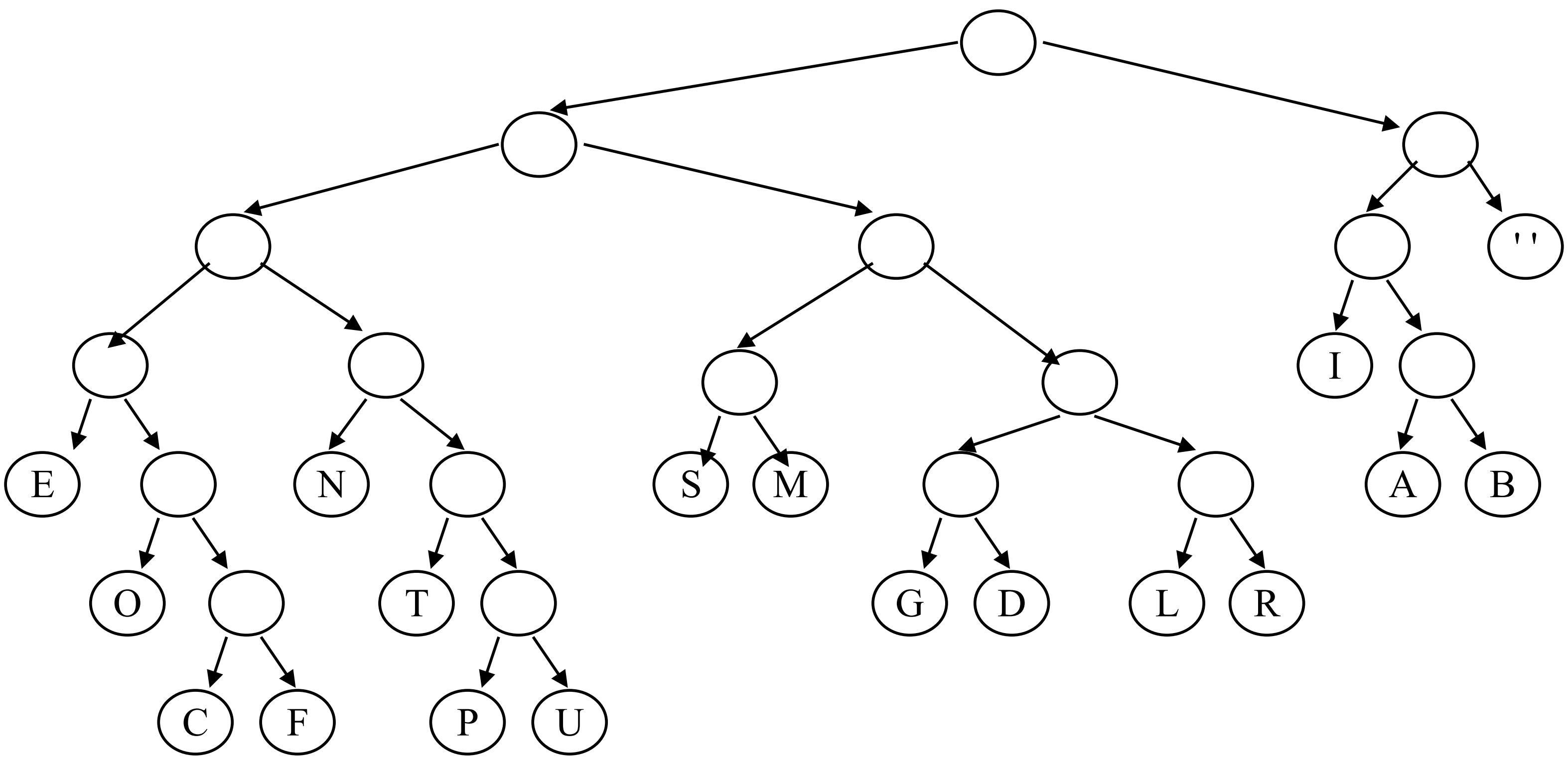
1011000010100011011000110101001110




BEAT

# Decoding a message

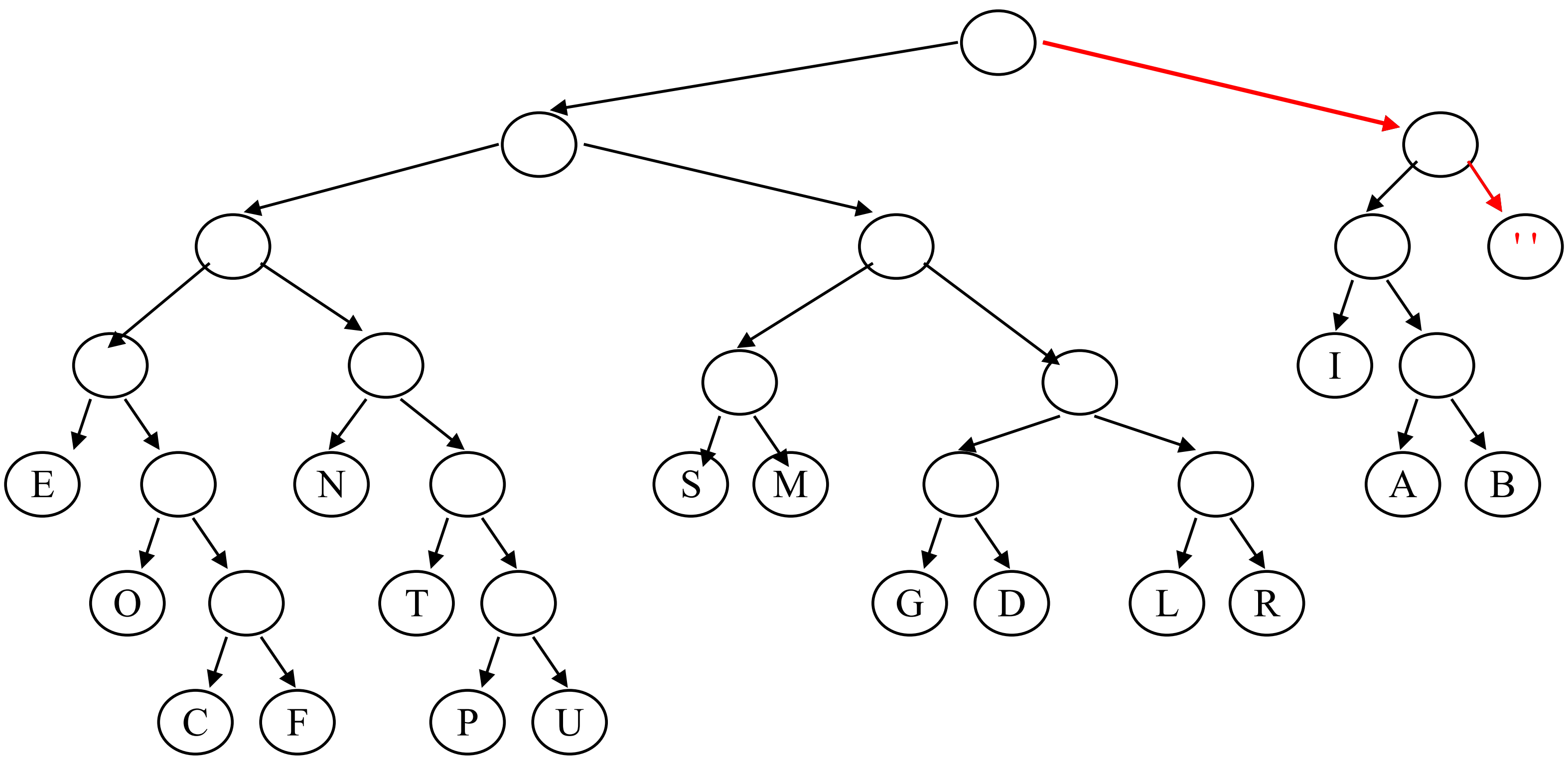
1011000001010001101100011010100 1110




BEAT

# Decoding a message

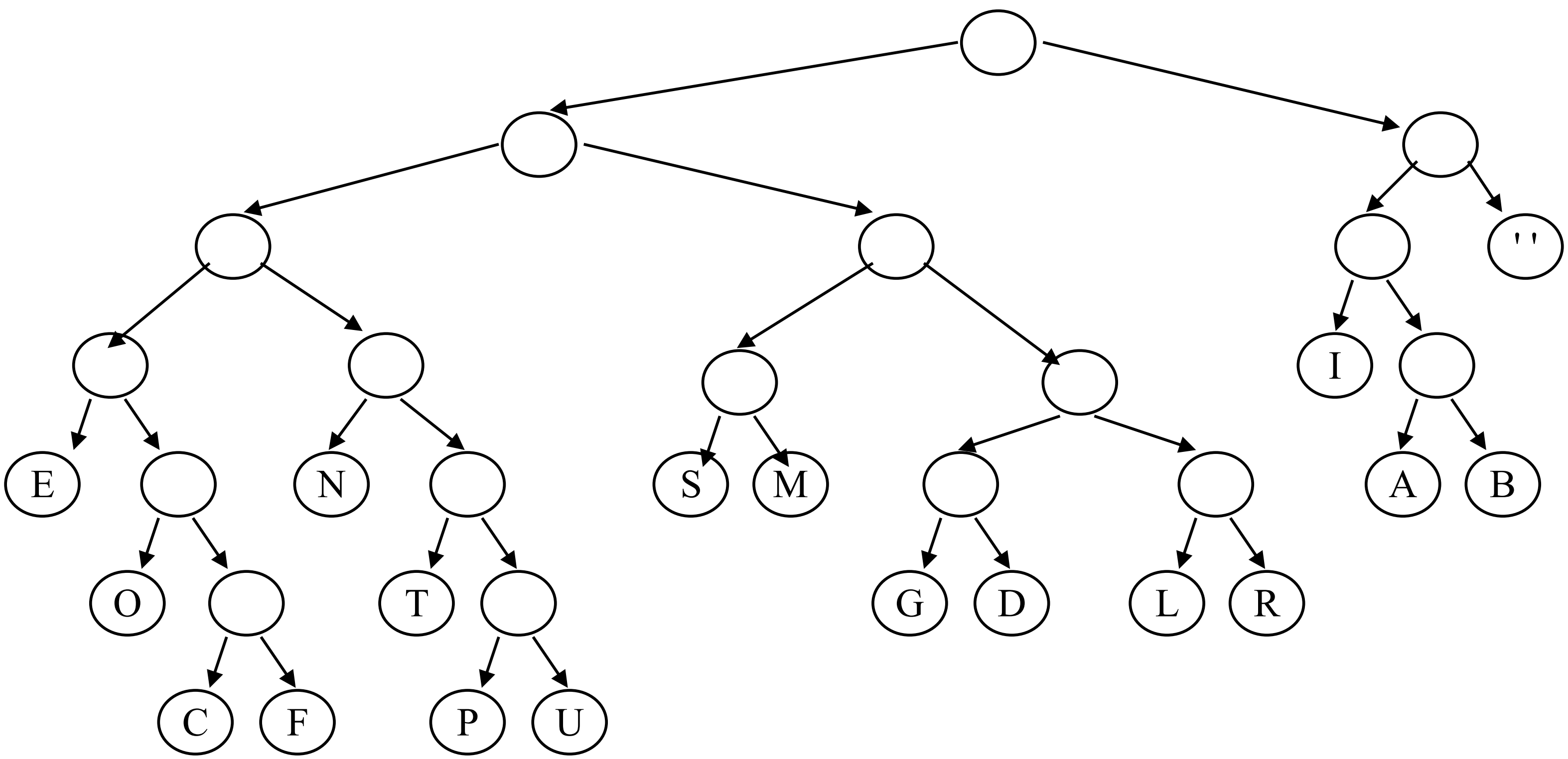
101100000101000110110001101010011 10




BEAT

# Decoding a message

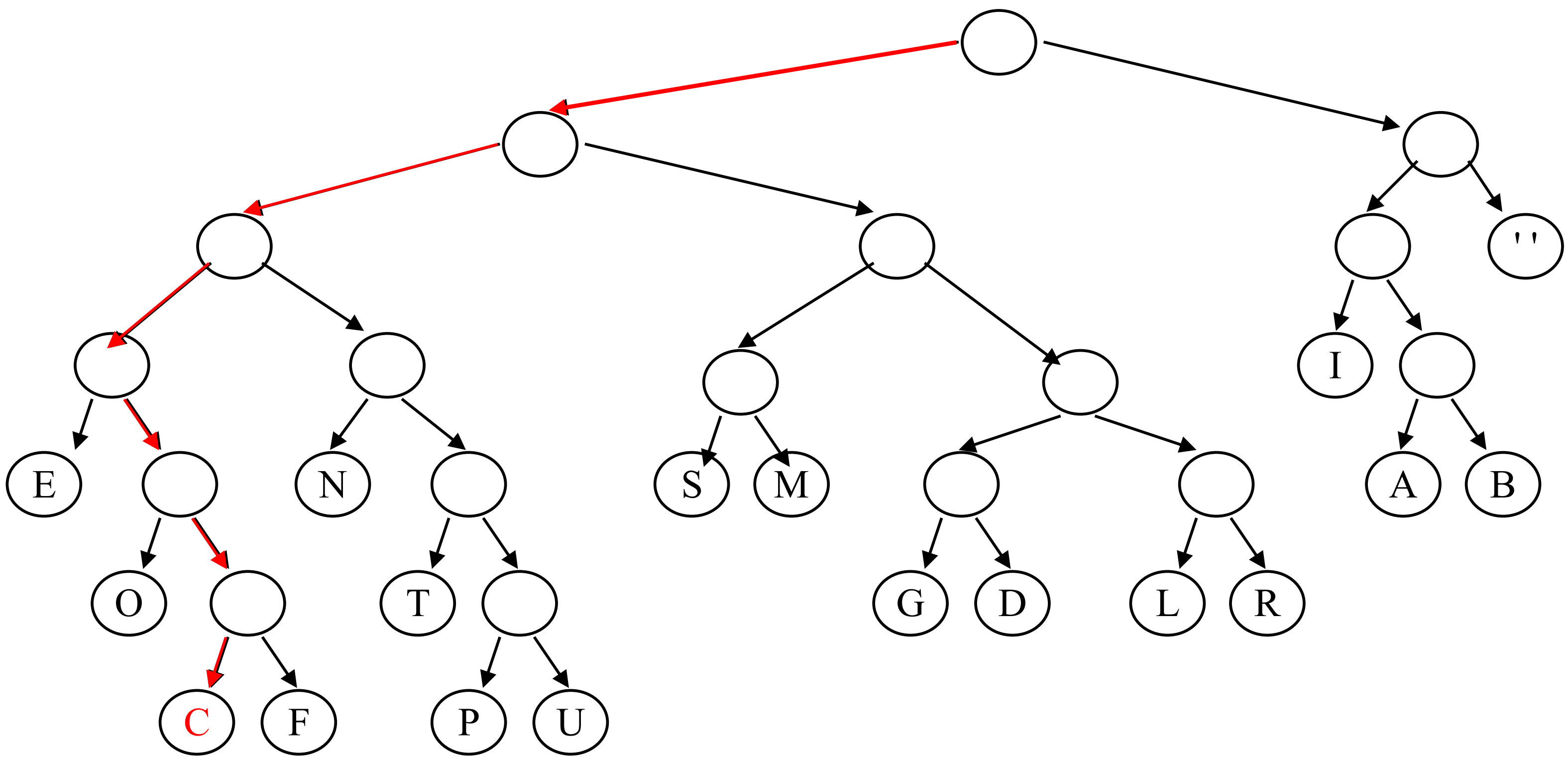
1011000001010001101100011010100111O




BEAT

# Decoding a message

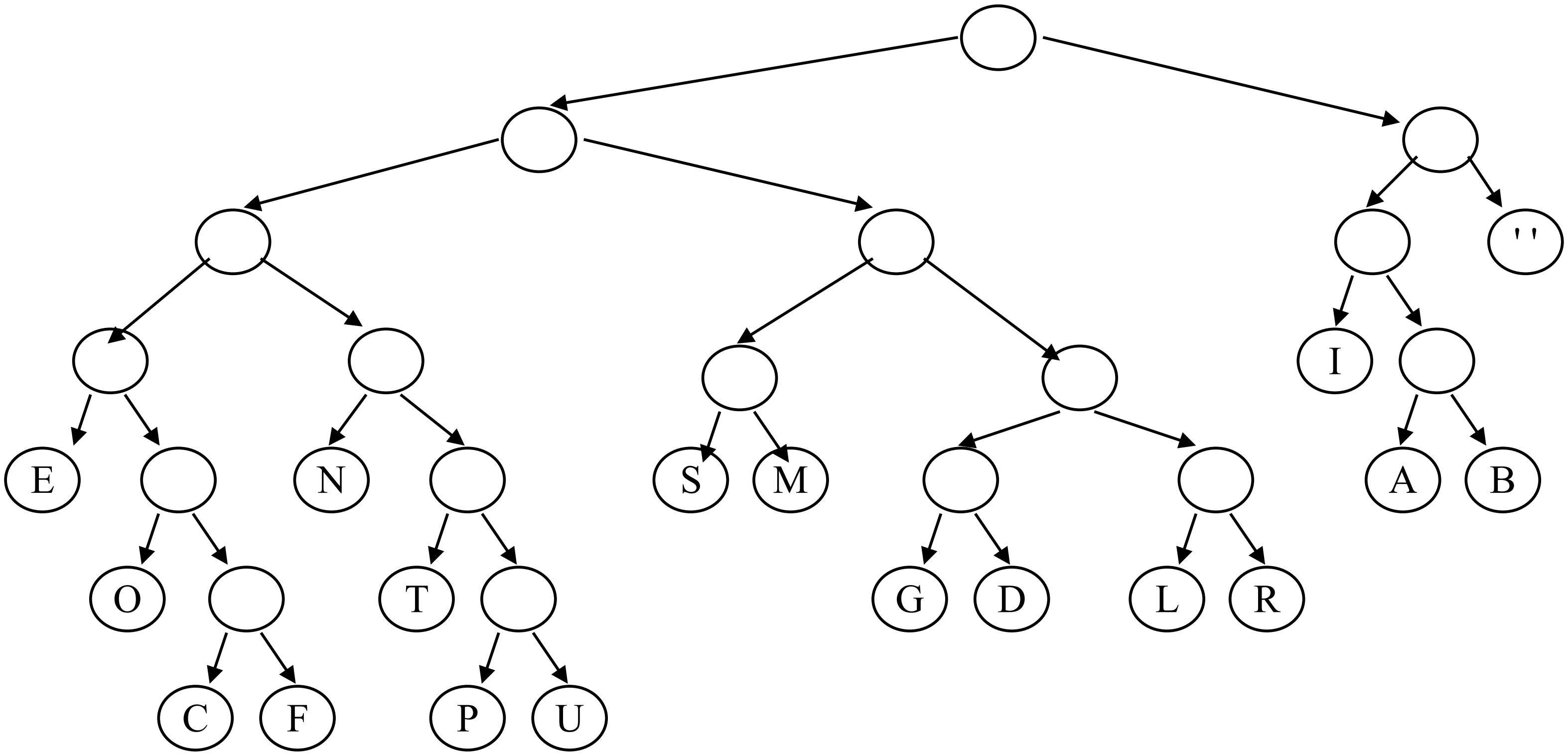
10110000101000110110001101010011110




BEAT C

# Decoding a message

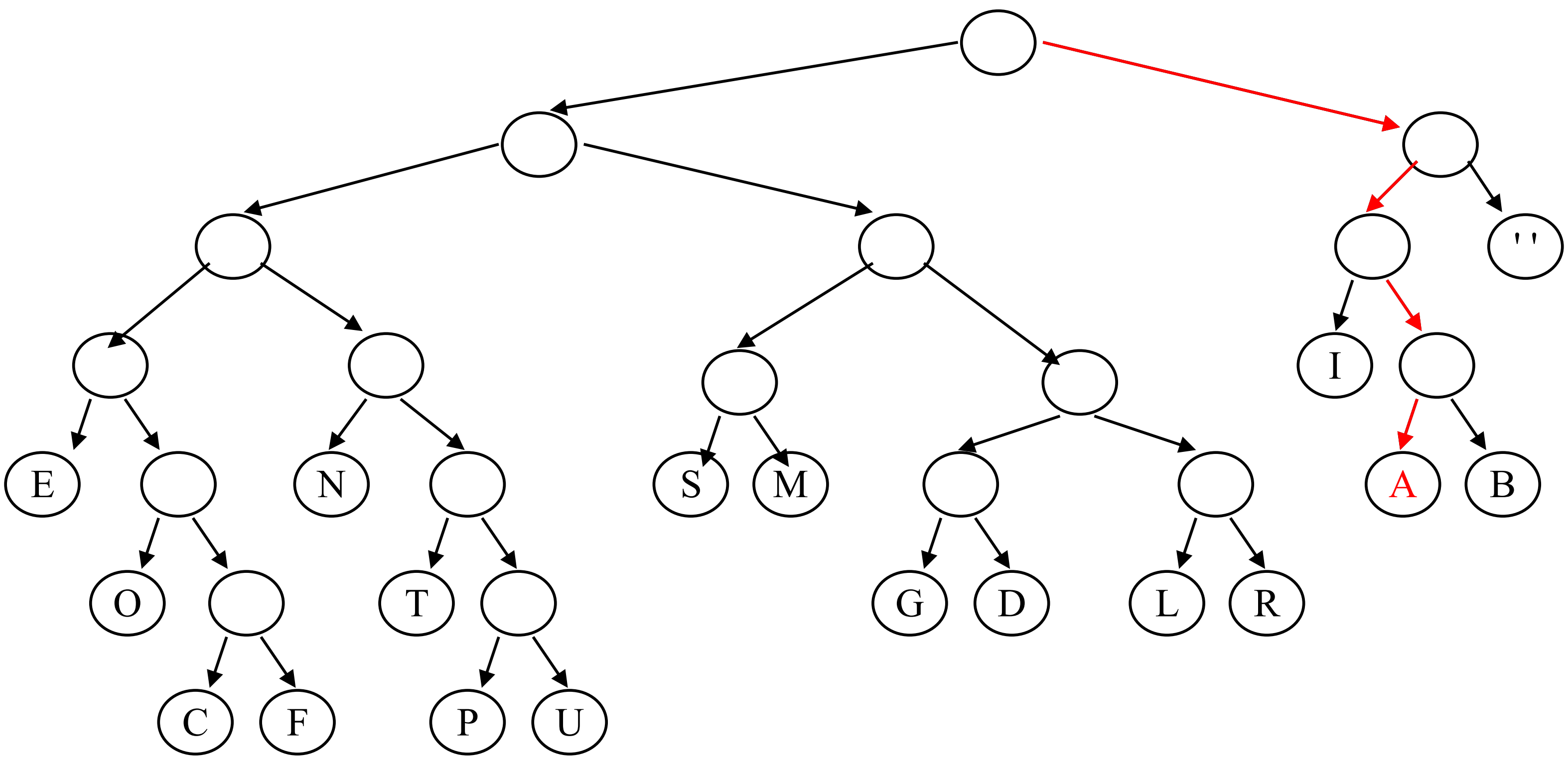
1011000010100011011000110**101001110**




BEAT C

# Decoding a message

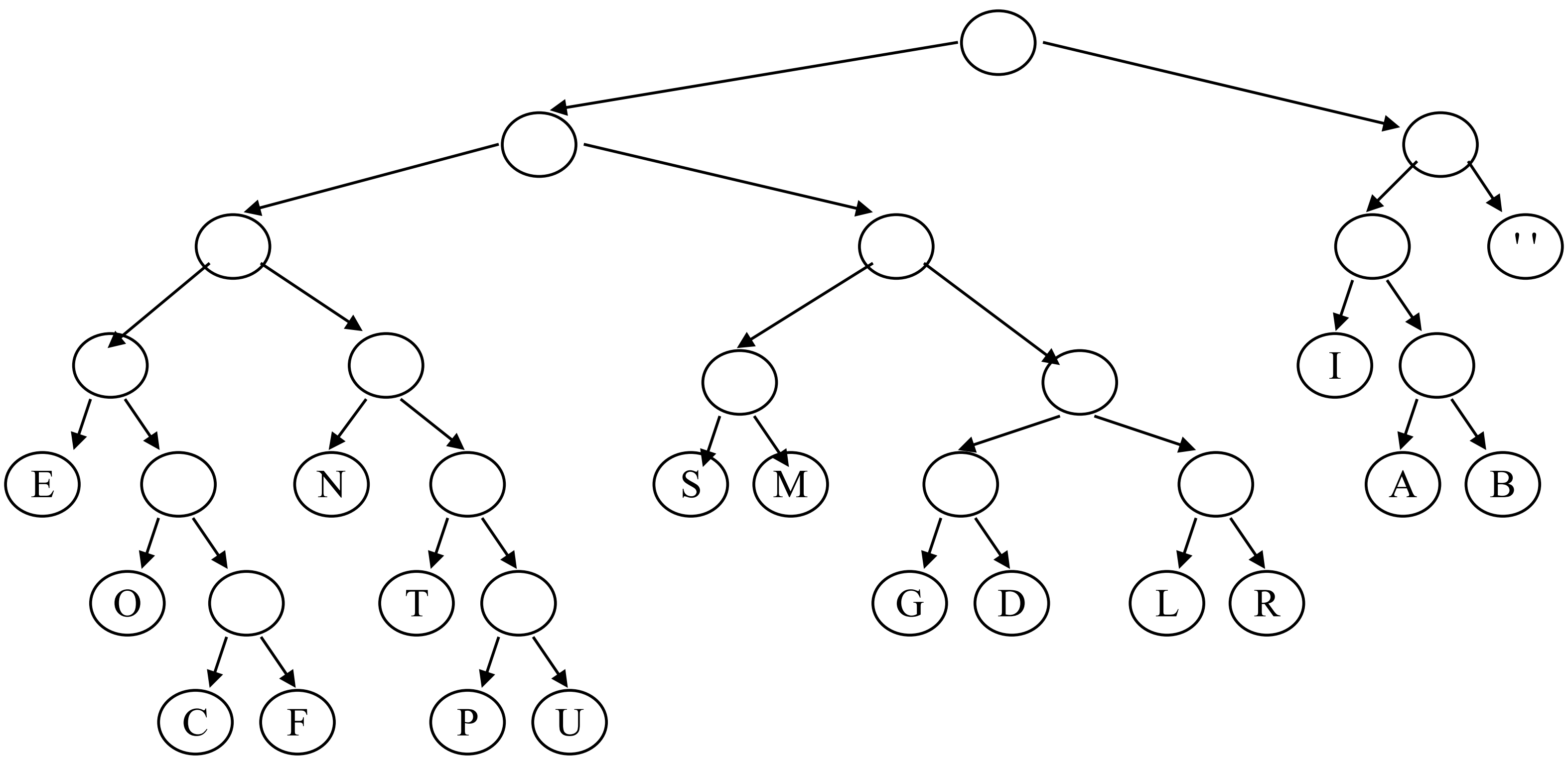
101100001010001101100011010100 1110




BEAT CA

# Decoding a message

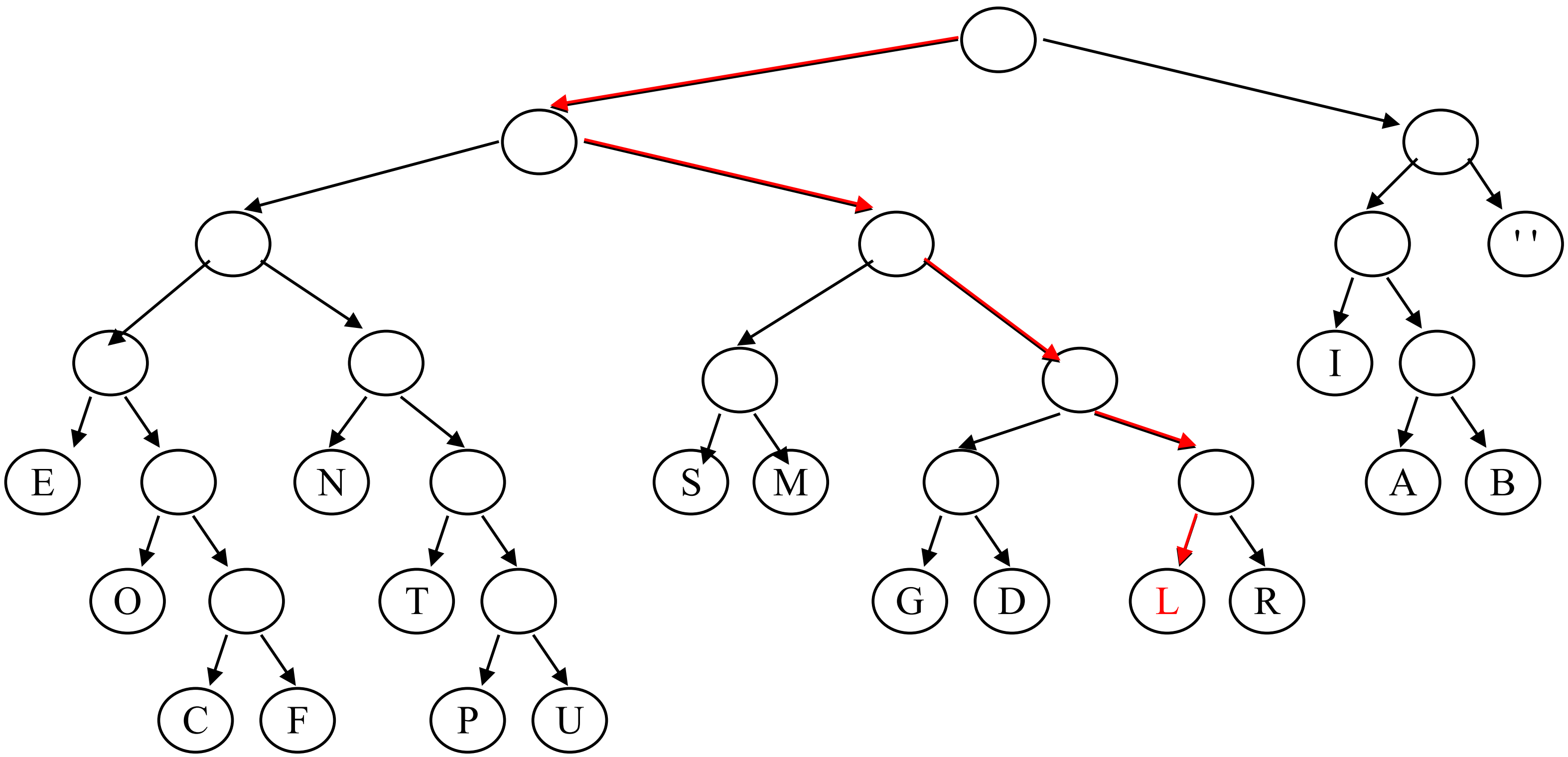
1011000010100011011000110101001110




BEAT CA

# Decoding a message

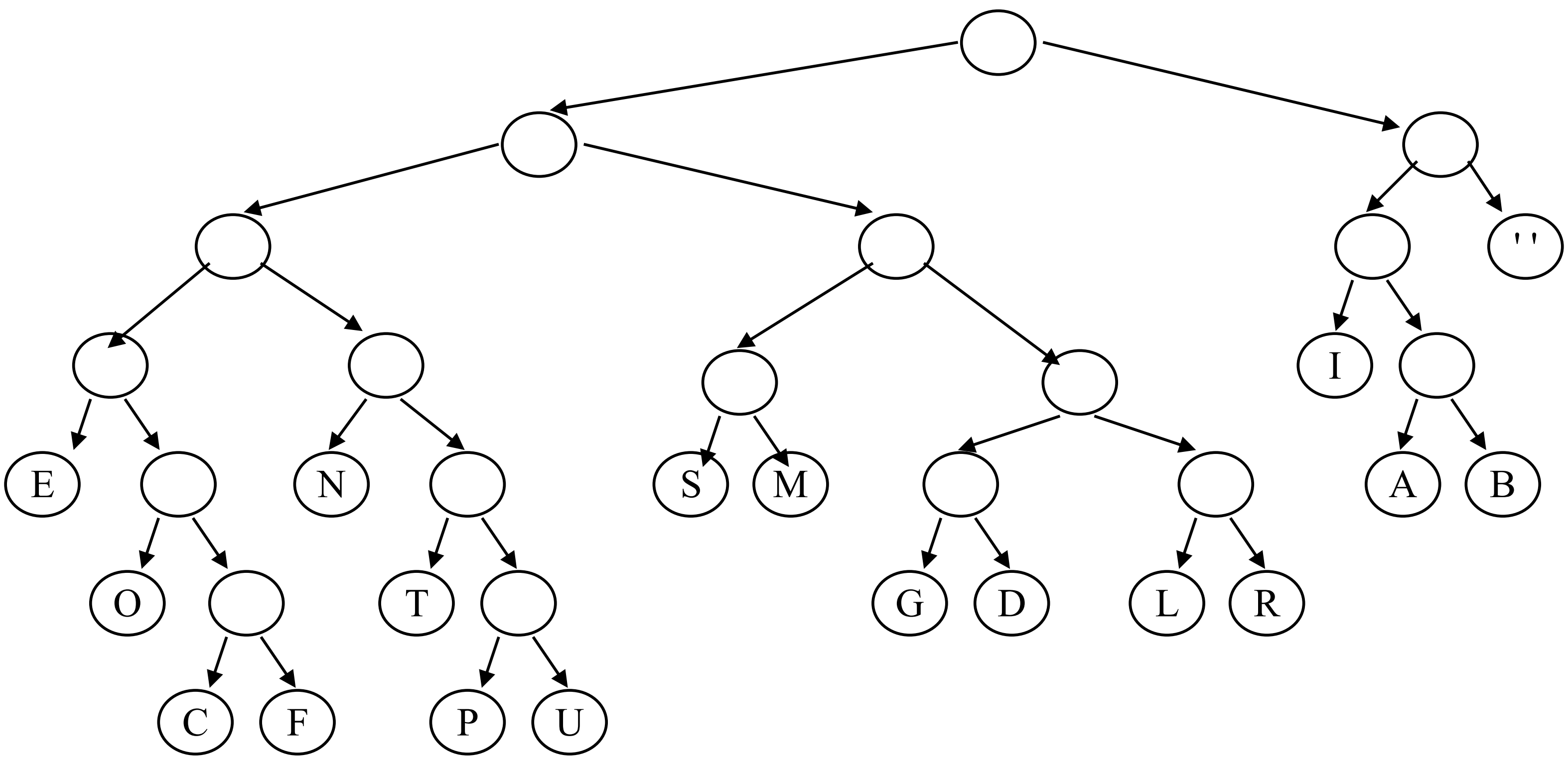
1011000010100011011000110101001110




BEAT CAL

# Decoding a message

1011000010100011011000110101001110




BEAT CAL

# Thoughts on Huffman coding

- Tightest encoding will minimize sum of weighted path lengths
  - (i.e minimizes number of bits used overall)
- "Greedy" algorithm
  - Makes locally optimal decision in search of globally optimal result
- Good and bad trees
  - What kind of trees result in lots of compression?
  - Which don't?
  - What kind of input produces a good tree? What about a bad one?