Sign In

- Website: `https://goo.gl/forms/FzHSa5INKlavWIJC3`
- Enter the word of the day in the appropriate slot.

Lecture #4: Simple Compression: Huffman Trees

- Strings are composed of characters, which (like everything else in a computer) are represented as bit strings.
- The relationship between characters and their bit representations (*encodings* or *code points*) is arbitrary. Standardization is necessary to prevent chaos.
- Python now uses an international standard known as *Unicode,* which encodes (as of Version 9.0) 128,237 characters, using code points that range from 0–1,114,111.
- These cover 135 scripts (roughly, alphabets), and various sets of symbols: punctuation, control characters (like tab or newline), mathematical symbols, etc.
- A few examples:

Literal	**Glyph**	**Encoding**	**Glyph**	**Encoding**	**Glyph**
`"\u0041"`	A	`"\u00A7"`	§	`"\u0398"`	Θ
`"\u0061"`	a	`"\u00A9"`	©	`"\u2663"`	♣
`"\u0030"`	0	`"\u00E9"`	é	`"\u2639"`	☹
`"\u0040"`	@	`"\u05D0"`	ℵ		

More Efficient Encoding

- If every character in a text is represented by an integer value in the full range, we'd have 3 bytes (24 bits) per character.
- So usually, the code points themselves are encoded.
- One common encoding, *UTF-8*, uses 1–4 bytes per character, depending on the number of significant bits in the code point.

Bits Coded	**Range of code points**	**Byte 1**	**Byte 2**	**Byte 3**	**Byte 4**
7	`0x0000 .. 0x007F`	`0xxxxxxx`			
11	`0x0080 .. 0x07FF`	`110xxxxx`	`10xxxxxx`		
16	`0x0800 .. 0xFFFF`	`1110xxxx`	`10xxxxxx`	`10xxxxxx`	
21	`0x10000 .. 0x10FFFF`	`11110xxx`	`10xxxxxx`	`10xxxxxx`	`10xxxxxx`

- x's mark places containing the bits of the code points. The other bits flag how many bytes are needed.
- Where one-byte characters are common, this saves space.
- One clever feature is that bytes 2–4 (continuation bytes) all start with a distinctive pattern (10), so that if one starts at any byte in an array of bytes, one can find the beginning of the character.

Still More Efficient

- We can, however, do better still by using other variable-length encodings that can use less than a byte per character.
- There's potential problem with this idea, however: ambiguity.
- Suppose we tried an encoding like this, using shorter codes for more common letters:

  ```
  E => 0, T => 1, A => 10, O => 11, I => 100, ...
  ```

- And suppose we receive the bits `100`.
- Is this "TEE", "AE", or "I"? Where does one letter end and the next begin?

Unique Prefix Property

- This ambiguity problem can be solved by choosing a code with the

  > *Unique Prefix Property:* The bit encoding for any character is never a prefix of the encoding of any other character.

- For example, the encoding

  ```
  E => 0, T => 10, A => 1101, O => 1100, I => 1110, ...
  ```

  has this property (at least for the characters shown). No encodings appears at the beginning of any other.

- E.g., "TEE" encodes to `1000`, "AE" to `11010`, and 'I' to `1110`.
- There is never any ambiguity about where a character begins, if one works from the left.
- Starting from a given bit position, p, as soon as one collects bits that match the encoding of character C, we know that C has to be the character that starts at p, since adding more bits can never match another character.

Decoding Using the Unique Prefix Property

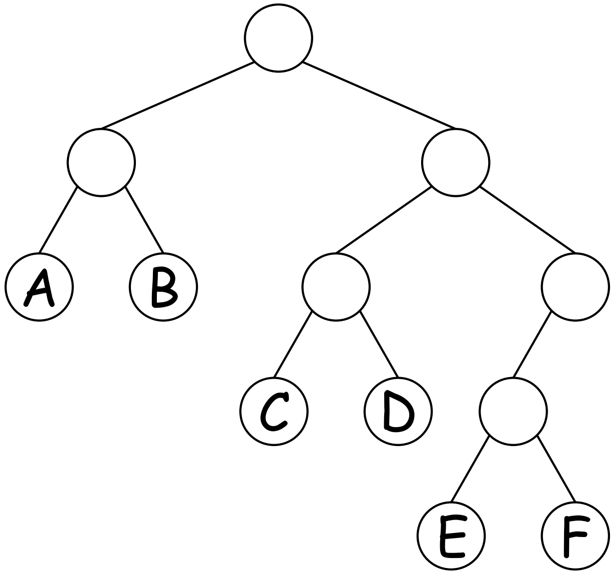
- Given a bit encoding with the unique prefix property, how do we decode?
- Discussion in previous slide gives one solution using a dictionary to map encodings to characters.
- For simplicity, imagine our encoded text as a string of 0s and 1s (not a representation you'd actually use in practice!).
- Suppose D is a dictionary from such strings of 0s and 1s to characters. Then,

```
def decode(msg):
    """Convert encoded message MSG into the character string it represents."""
    ch = ""
    result = ""
    for b in msg:
        ch += b
        if ch in D:
            result += D[ch]
            ch = ""
```

Using Trees

- Binary trees offer a particular way to represent the dictionary from the last slide.

Letter	**Encoding**
A	00
B	01
C	100
D	101
E	1100
F	1101



- Left branches tell what to do when looking at a 0 bit; right branches do the same for 1 bits (result is called a *Patricia tree*.
- To decode, e.g., `1101001011100`,
  - Following bits `1101` (right, right, left, right) takes us to leaf 'F'.
  - Returning to the top, `00` takes us to 'A'.
  - Again from the top, `101` takes us to 'D'.
  - Finally, `1100` gives 'E'. Complete decoding: "FADE".

A Problem

- How, then, do we get an encoding that
  - Minimizes the size of a text, and
  - Satisfies the unique prefix property (so that it can be decoded unambiguously.)
- There is no universal encoding that does this for any text.
- We'd like an algorithn that finds a custom-made optimal encoding for any particular text.
- Idea is to encode more common charcters in fewer bits.

Huffman Coding

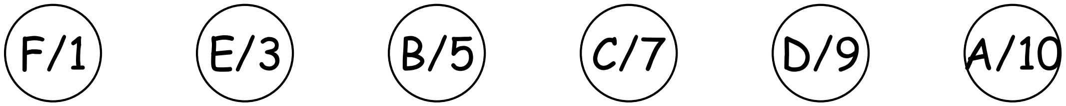
- Huffman coding is named after an MIT student who invented this encoding in response to a class assignment.
- Given an alphabet of symbols to be encoded, with their relative frequencies in a text, it produces the optimal variable-width unique-prefix encoding, assuming that we encode individual characters independently.
- Basic idea is to accumulate trees representing subsets of characters from the bottom up, starting with trivial trees each containing a single character.
- Each time two trees are clustered into one under a new parent node, it represents an additional bit in the coding, so it is best to prefer clustering trees that represent characters with smallest frequency.

Example

- Want to encode string "`AAAAAAAAAABBBBBCCCCCCCDDDDDDDDDEEEF`"
- Here, the frequencies are

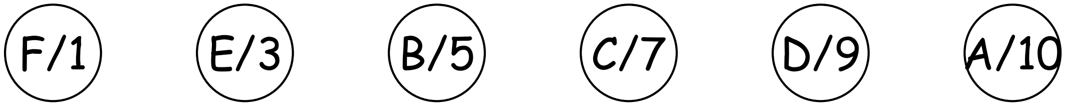
Letter	**Count**
A	10
B	5
C	7
D	9
E	3
F	1

- Represent as 6 one-node trees labeled with letters and their frequencies:

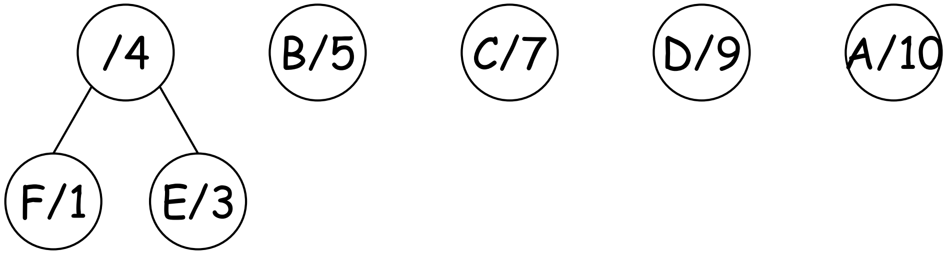


Forming Subtrees

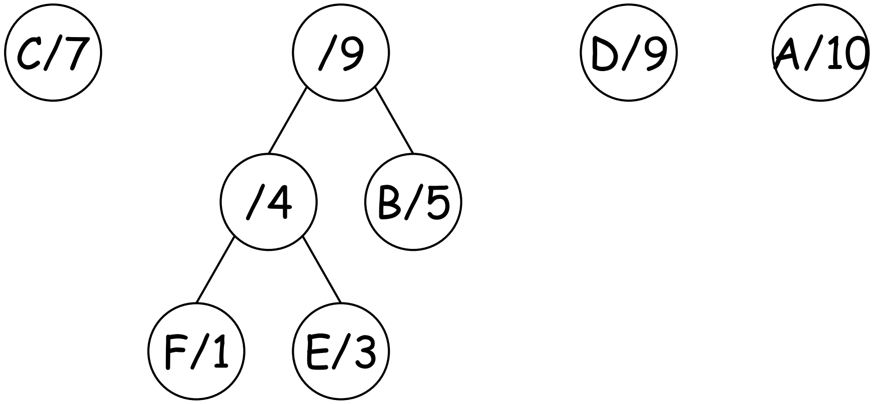
- Starting with



- We combine the two nodes with the smallest frequencies to get a “bigger” node representing both the characters E and F:

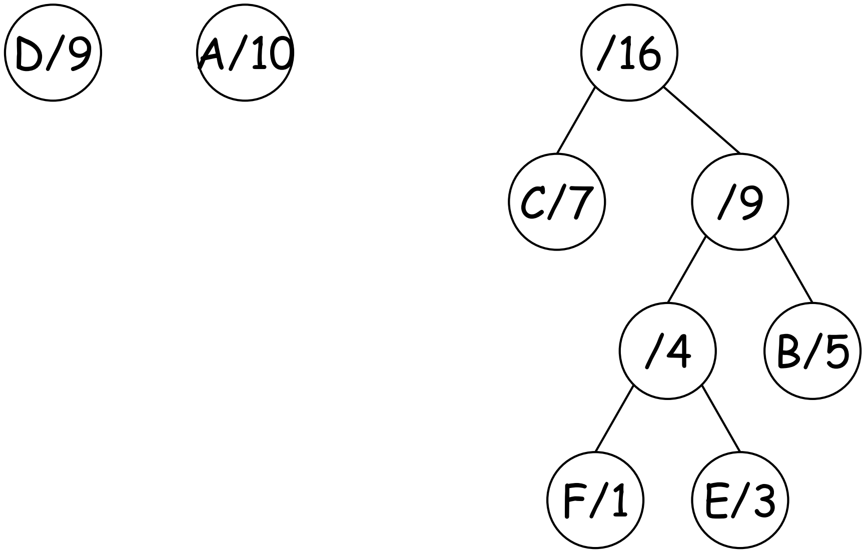


- Keeping the resulting trees in order by frequency, repeat:



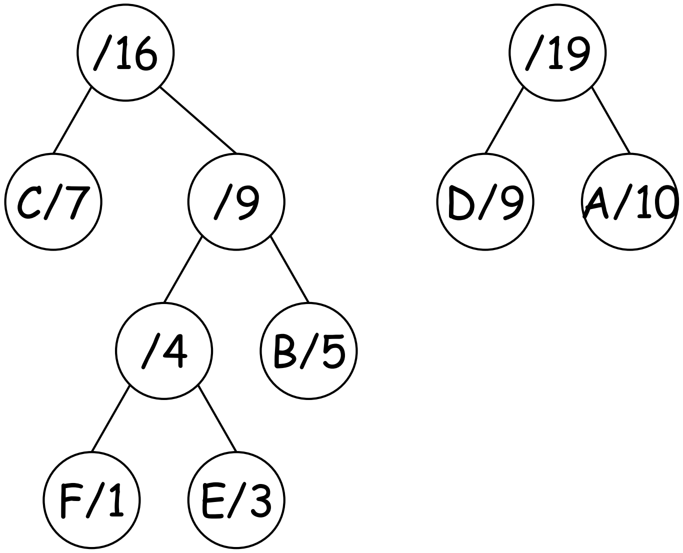
Forming Subtrees (II)

- And again:



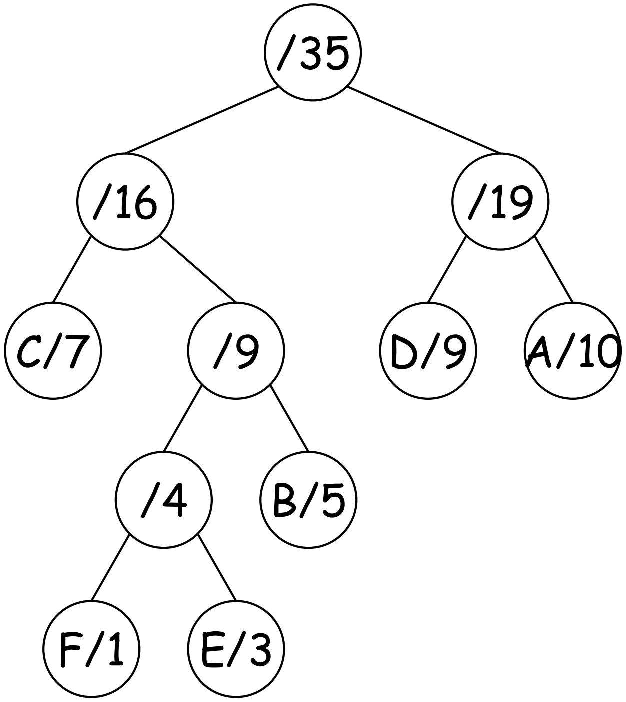
Forming Subtrees (III)

- And yet again:



Forming Subtrees (IV)

- Finally, we get the tree on the left, which corresponds to the encoding table on the right



Letter	**Encoding**
A	11
B	011
C	00
D	10
E	0101
F	0100

- So string "`AAAAAAAAAABBBBBCCCCCCCDDDDDDDDDEEEF`" encodes as "`11111111111111111111011011011011011000000000000001010101010101010100101010101010100`" which is 84 bits as opposed to 94 with our previous unique-prefix encoding from slide 6, and 280 using UTF-8 and Unicode.