Computer Science 210
Data Structures
Siena College
Fall 2019

# Topic Notes: Huffman Trees

We will now briefly consider a *greedy algorithm* concerned with the generation of encodings.

The problem of *coding* is assignment of bit strings to alphabet characters. *Codewords* are the bit strings assigned for characters of alphabet.

We can categorize codes as one of:

1. *fixed-length encoding* (*e.g.*, ASCII)
2. *variable-length encoding* (*e.g.*, Morse code)

The idea here is to reduce the amount of space needed to store a string of characters. Usually, we store characters with 8 bits each, meaning we can store up to $2^8 = 256$ different characters.

However, many strings don't use that many different characters. If we had a string that used only 12 unique characters, we could define patterns of 4 bits each to represent them and save half of the space.

The idea behind variable length encoding is that we can do even better if we use short strings of bits to represent frequently seen characters and infrequent characters with longer strings. This results in a smaller total number of bits needed to encode a message.

To do this, we need to come up with a code and a way to translate text into code and then back again.

But...these variable length codes introduce a problem. If each character's code can have a different length, how do we know when the code for one character has ended and the next has begun?

In Morse code, how can we tell the if the sequence "dot dash dash" is supposed to represent "AT", "ETT", "EM" or just the one character "W"?

This is possible because Morse code is not a binary code at all – it does have dots and dashes (which are one and three time units, respectively, of the sound), but it also has pauses of varying length to separate the individual dots and dashes (a period of silence equal in duration to the sound of a "dot"), to separate letters (silence for the duration of a dash), and to separate words (silence for the duration of 7 dots).

A strictly binary code cannot have these other "markers" to separate letters or words. Therefore, we would construct a *prefix-free code*, one where no codeword is a prefix of another codeword.

This leads to an interesting problem: if the frequencies of the character occurrences for the string to be encoded are known, what is the best binary prefix-free code?

Consider this procedure to generate a translation system, known as a *Huffman coding*.

Count the number of each character in the string to represent and create a single-node binary tree with that character and its count as the value. Repeatedly take the smallest two trees in the collection and combine them to a new tree which has the two trees as subtrees and label the root with the sum of their counts. Continue combining trees (both the original one-element trees and the trees created) in this manner until a single tree remains.

Consider the phrase:

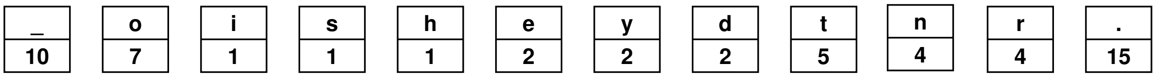
```
no... try not... do... or do not... there is no try...
```

We count the number of times each character occurs in our string:

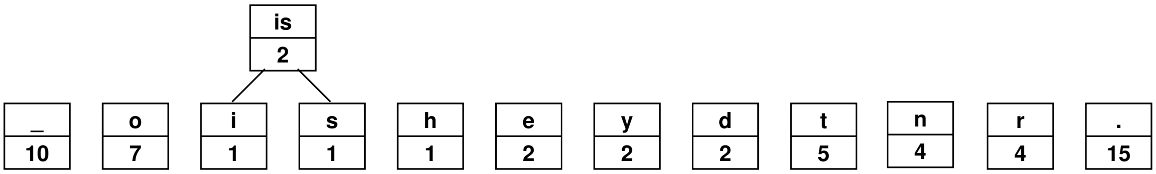
| Character | n | o | . | _ | t | r | y | d | h | e | i | s |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Frequency | 4 | 7 | 15 | 10 | 5 | 4 | 2 | 2 | 1 | 2 | 1 | 1 |

and build the tree.

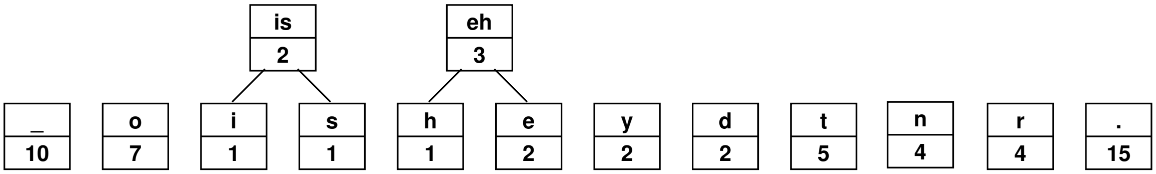
We start with tree nodes for each of our characters with their frequencies:



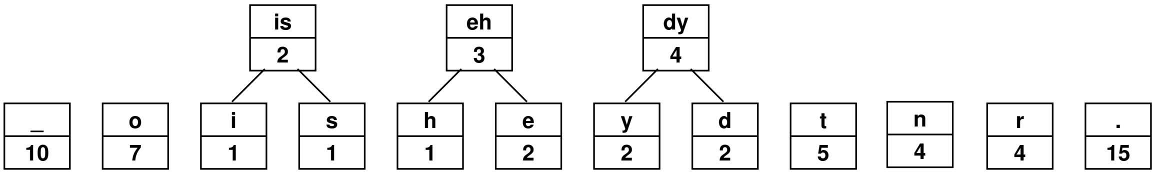
And now combine, in greedy fashion, the "smallest" two nodes without a parent node (*i.e.*, the ones with the smallest two total character frequencies) below a new node, which has all of the characters and their cumulative frequency. In this case, any pair of nodes with frequency 1. For example, we could combine the node for `i` and the node for `s` below a new node for `is` with a total frequency of 2.



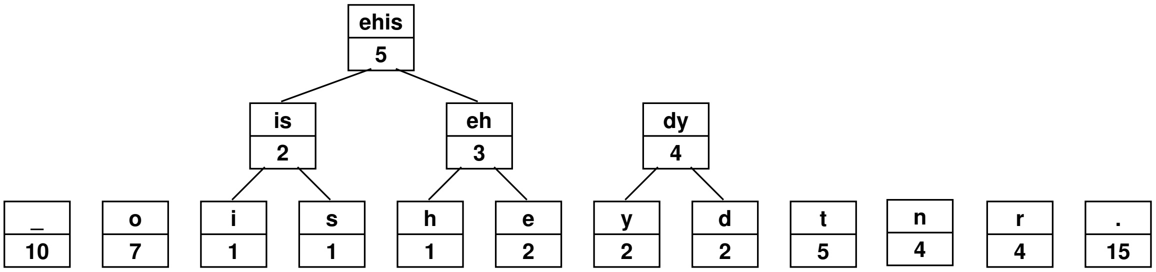
We next need to combine the remaining node with frequency 1 (the `h`) with one of the nodes with frequency 2.



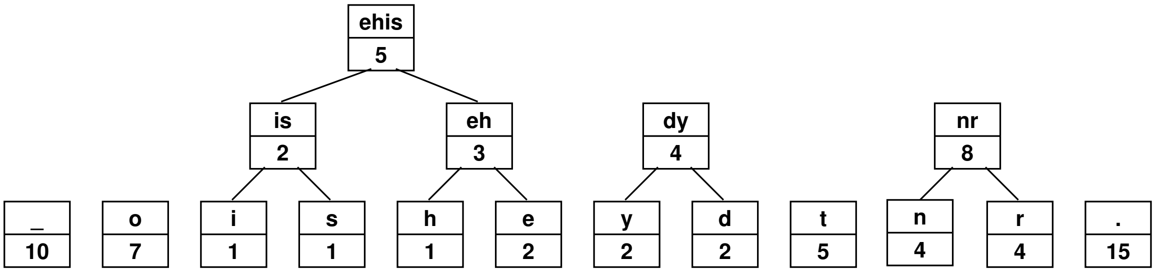
Next, two of the nodes with frequency 2 are combined.



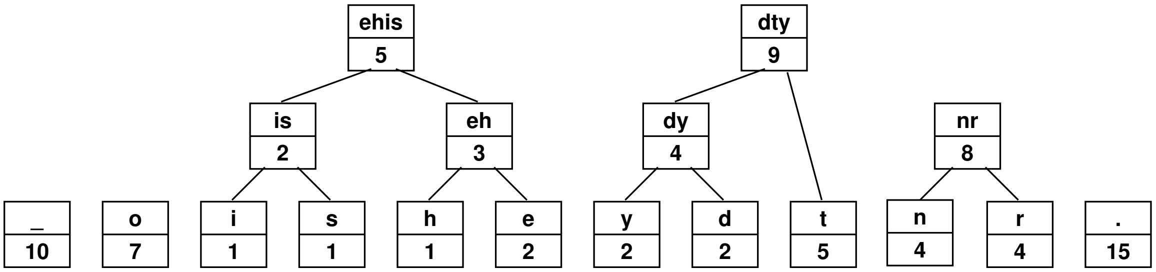
We have no choice on the next step: the frequency 2 and frequency 3 nodes need to be combined.



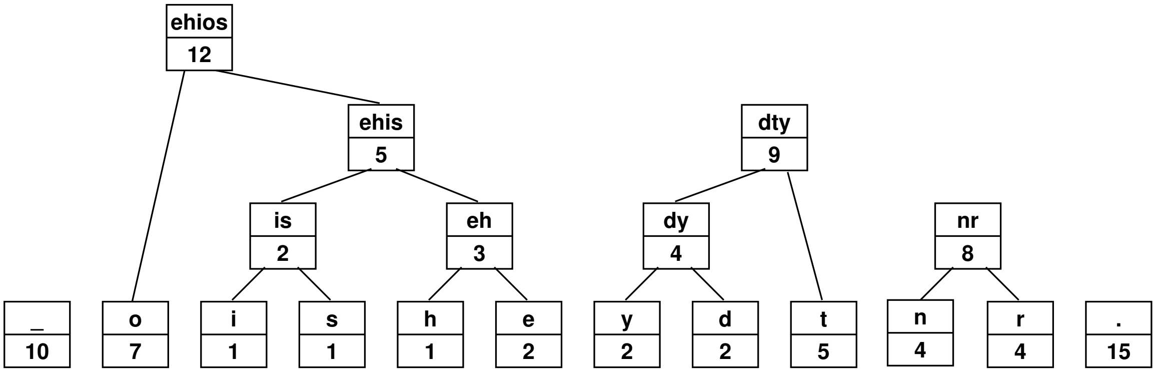
Then, we'll combine two of the nodes with frequency 4.



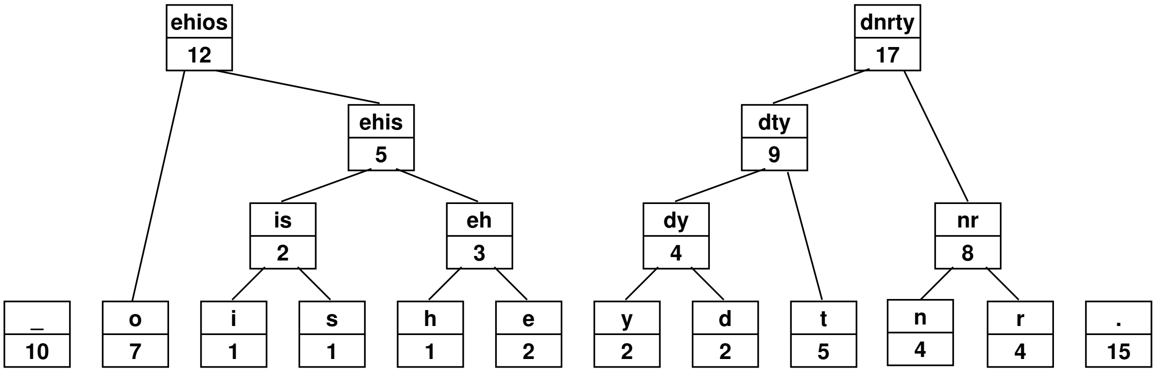
Next up, the node with frequency 4 needs to be combined with one of the nodes with frequency 5.



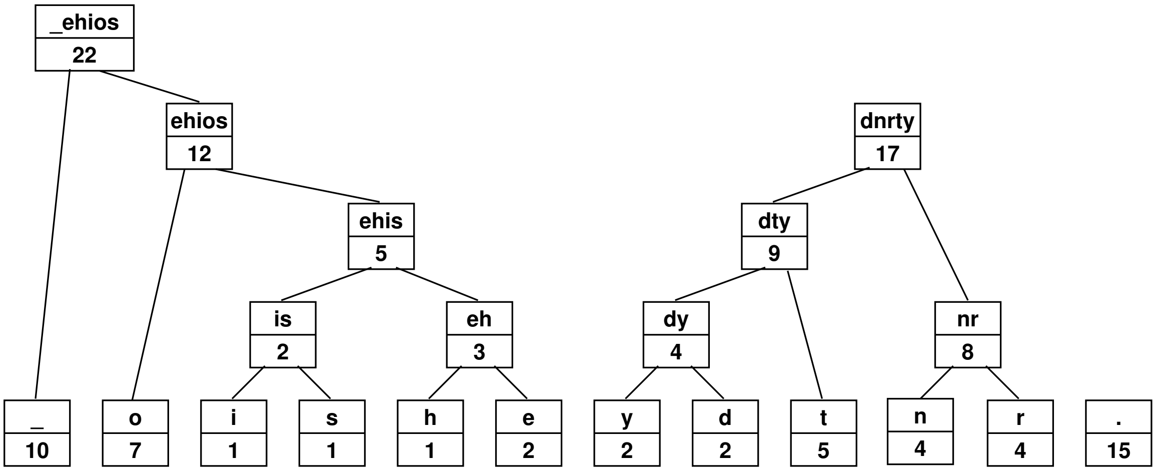
As we continue, the next to be combined are the nodes with frequency 5 and frequency 7.



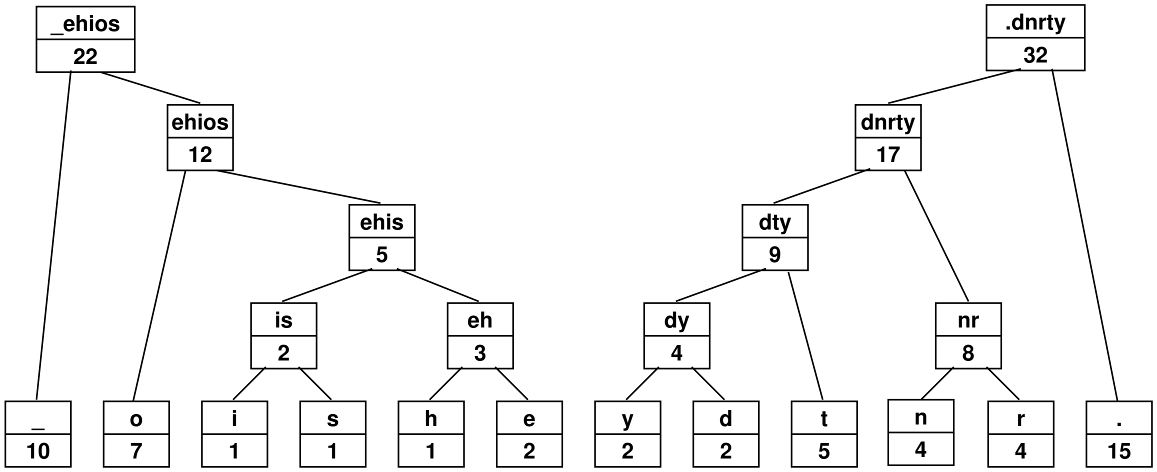
The nodes with frequencies 8 and 9 need to be combined next.



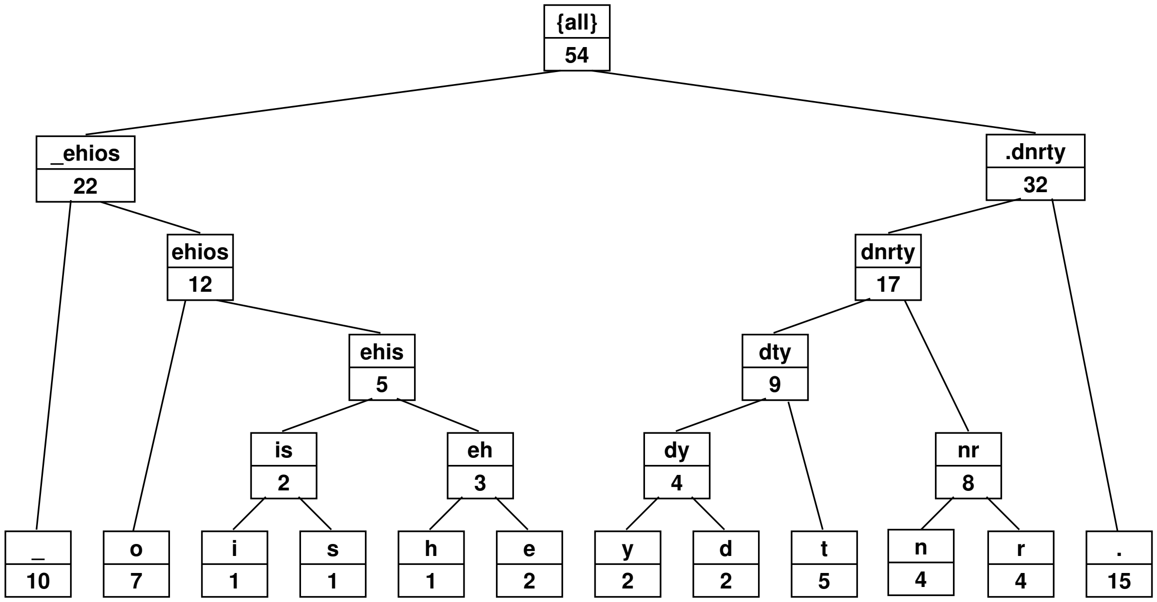
Then it's the nodes with frequences 10 and 12.



Almost there. The smallest frequencies now are the 15 and the 17.

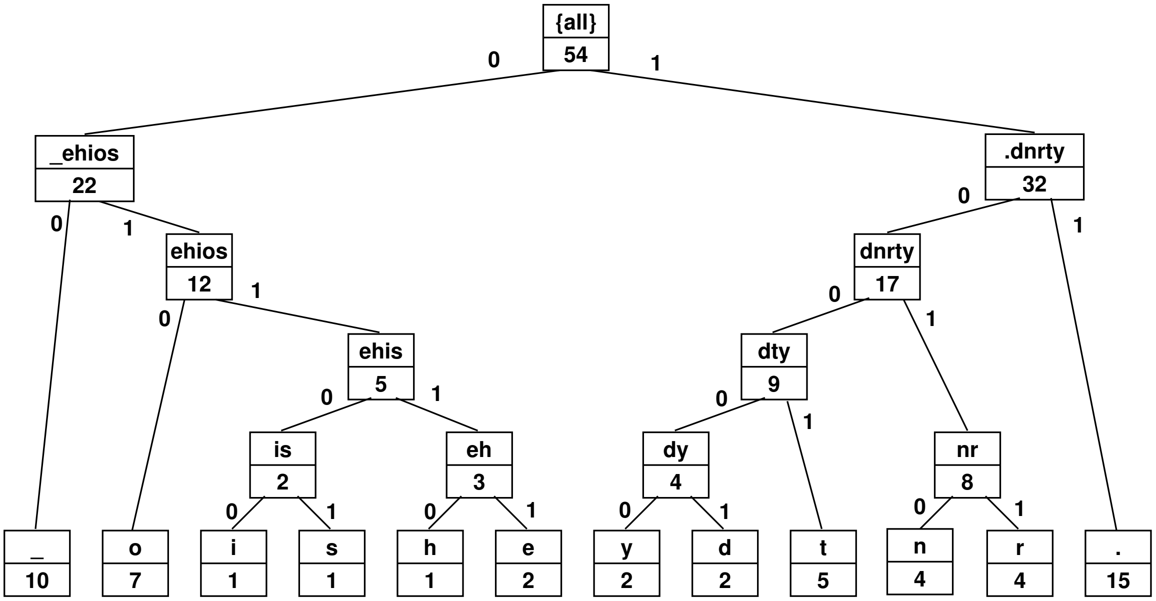


And last, we have just two nodes left to combine, so we do.



To check yourself, make sure the frequency at the root matches the length of your input string.

With the completed tree, we label all the branches with 0 or 1, which determine the bit to use as a tree edge is traversed. We'll put 0's on all left child edges, 1's on the rights.



Note again that the construction is a greedy procedure: we simply take the tree from our collection that has the smallest number of characters represented.

Note also that there are many possible trees that are equally valid and optimal. Any node's right and left subtrees can be swapped, and ties can be broken in any order.

Once we have the tree, we can use it to construct our encoded (compressed) string. For each character, we traverse from the root of the tree to the leaf that contains that character, and add the bit on each edge traversed to our string.

So we start with the letter `n`, whose leaf can be found by traversing to the right (`1`), left (`0`), right (`1`), right (`0`).

```
1010
```

Then, `o`, which is found by traversing left (`0`), right (`1`), left (`0`).

```
1010010
```

and so on for each of the 54 characters in the input string.

```
1010010111111001001101110000001010010100111111100
100010101111110001010110010001010001010010100111111100
10010111001111101101111000110001101001010010001001101110000111111
```

We have stored the 54-character string in 168 bits, as opposed to the 432 it would take in standard ASCII.

Note that some characters have shorter codes than others. This is where the optimality of the algorithm arises. The most frequent characters will be encoded with shorter bit sequences. Rare characters will have longer sequences. Further, there is no special code or sequence that indicates the separation between the bits representing characters. We will see how this is possible when we look at the decoding.

Of course, the encoding is useless if we cannot retrieve our original text. So we would also need to store (and in the case of sending the text over a network, send) the tree itself.

To decode, we just trace the bit patterns through the tree. Start at the root, and move left for each 0 bit, right for each 1 bit in the encoding. When we reach a leaf, we know the next letter is the one in that node. If there is more of the encoded string remaining, we start tracing at the root again.

So to start decoding our original string, we start at the root and see a 1 in the first position of the encoding. So we go right to the `.dnrty` node. Next is 0, so we go left to `dnrty`. Then 1 takes us right to `nr`, and 0 moves us left to the `n` leaf node. So `n` is the first character of the string. We return to the root, and the bit pattern of the encoding has a 0, which we follow left to `_ehios`, 1 right to `ehios`, then 0 left to `o`, which we know is the second character. And so on...

The decoding works because the codes generated will never include one character's code as a prefix of another character's code.

The same tree can be used to encode any string with the same set of characters, but would be guaranteed to be optimal only for the input string that was used to construct the tree.