Huffman Encoding

Huffman Encoding is a data compression scheme for text files. Characters are assigned a bit sequence based on how often they appear in the document. The algorithm makes clever use of some simple data structures to accomplish this goal.

Let's use this simple test String as an example: "go go gophers"

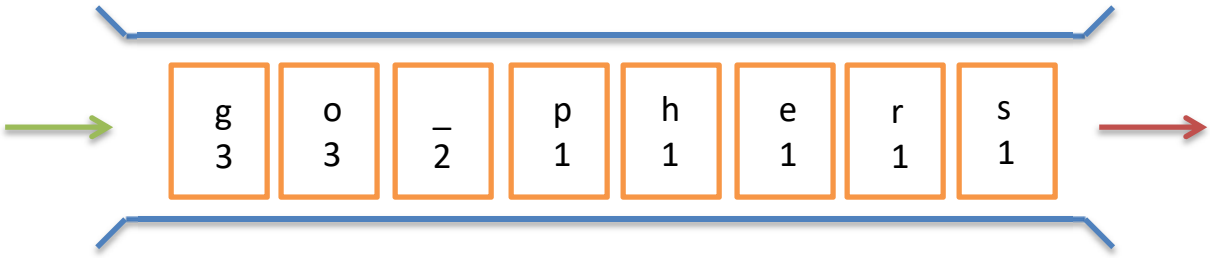
Normally, an ASCII character requires 8 bits, so our 13 character example would require 104 bits. Huffman encoding will allow us to reduce this to about 1/3 of the original bits.

The algorithm follows several steps:

1. Count the frequency of each character:

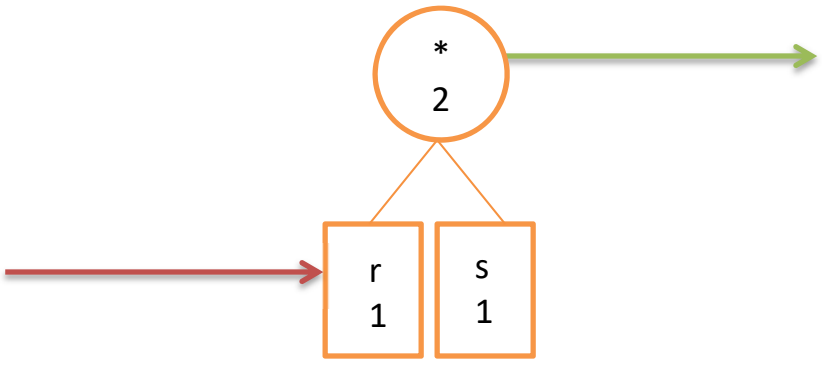
Character	Frequency
g	3
o	3
p	1
h	1
e	1
r	1
s	1
Space	2

2. Make a Binary Tree Node for each character. It will store the character and its frequency. Place these into a minimal Priority Queue:

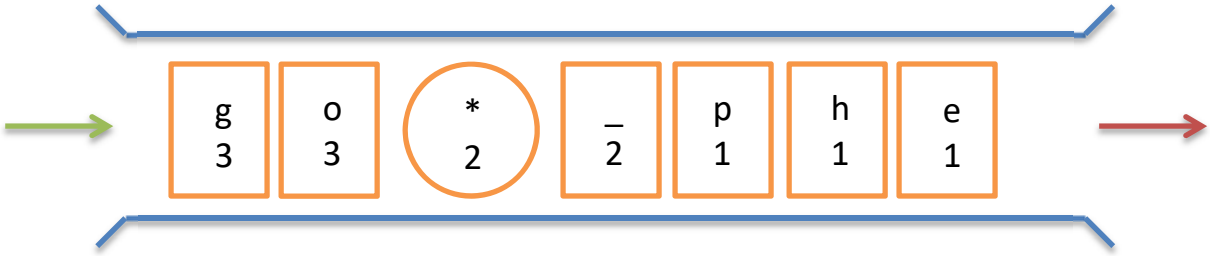


Note that these Nodes aren't connected right now. We have yet to make the actual Tree.

3. Dequeue two nodes. Make a new Tree Node that connects them together. The new Node's frequency will be the sum of the two just dequeued. Enqueue the new Node.



The Queue now looks like this:

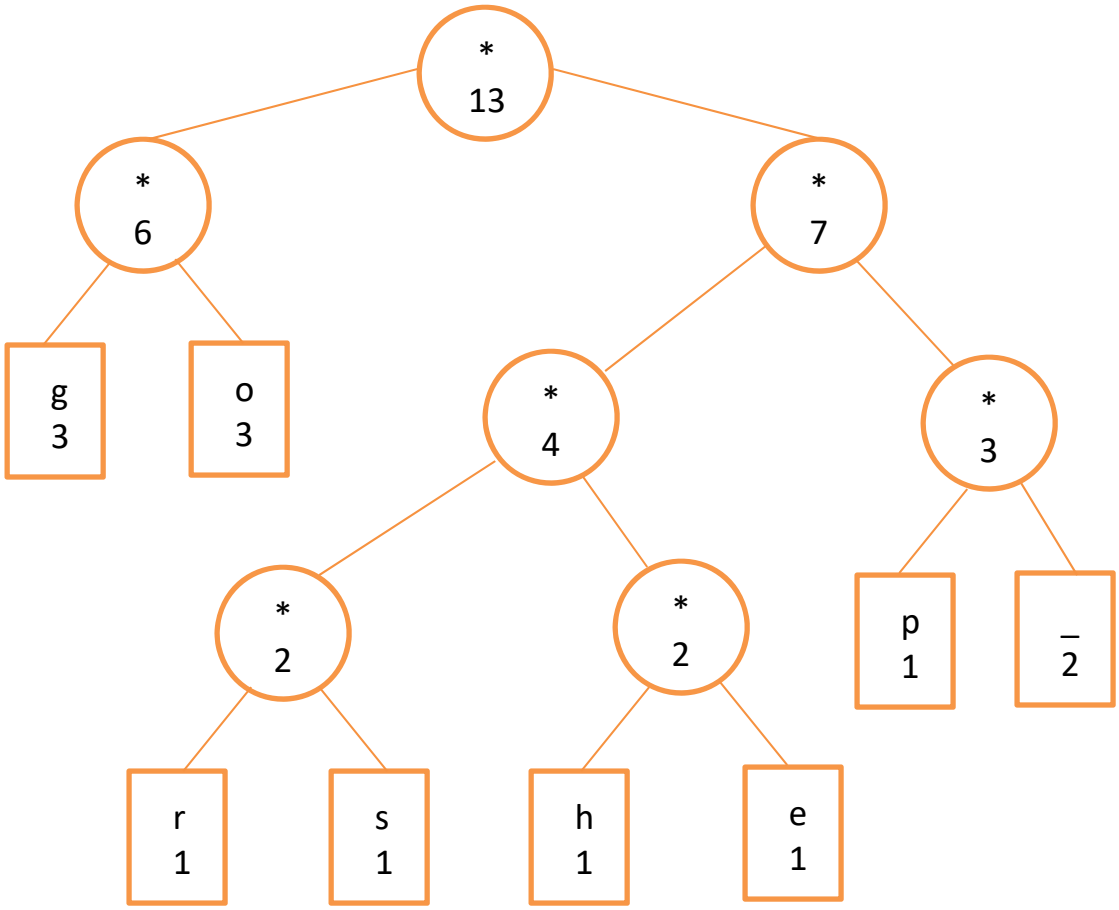


Note that the small, three-Node tree we just made slots in with the other Nodes. Note also that we've reduced the number of Nodes by one. 'r' and 's' haven't disappeared; since the '*' is their parent, we'll be able to find them later.

4. Repeat step 3 until there is only one thing left in the Queue.

   Since we'll have been continually popping two nodes, connecting them and enqueue-ing their new parent Node, eventually the one remaining Node must be the root of a Tree. That Tree must contain all the nodes.

   For "go go gophers", we'll get a Tree that looks something like this:



5. Trace through the Tree down to each leaf. We want to record the path taken to reach the leaf and the character we find there. Record every left turn taken as a 0 and a right turn as a 1:

Character	Path
g	00
o	01
p	110
h	1010
e	1011
r	1000
s	1001
Space	111

And we're done! Now we have much shorter bit representations for the same characters. Where each was eight bits long, now they're only two, three, or four.

I can now use these representations to **encode** and **decode** my text document. Encoding it will make it smaller, and decoding it will return the original document.

To encode it:

- Read through the original document.
- Look up each character's new encoding.
- Append the encoding to an output file or String.

To decode it:

- Read through the encoded document.
- For each character…
    - Start a pointer at the root of the tree.
    - If you encounter a zero, move the pointer to its left child.
    - If you encounter a one, move the pointer to its right child.
    - Once you hit a leaf, append the character there to an output file or String.