Huffman Codes

Huffman codes are used for data compression. Huffman encoding is one of the earliest data compression algorithms; popular programs like Pkzip and Stuffit use their own techniques but are based on the original schemes such as Huffman or LZW. Compression is useful for archival purposes and for data transmission, when not much bandwidth is available.

Idea: Let's say you want to compress your file, and your file only contains 6 characters, ABCDEF. If you store these using an 8-bit ascii code, you will need space 8N bits, where n is the numbers of characters in your file. If n=1000, this is 8000 bits.

One way to do better: Since you only have six characters, you can represent these in fewer bits. You really only need three bits to represent these characters:

000	A
001	B
010	C
011	D
100	E
101	F

Immediately, we can reduce the storage necessary to 3N bits. If n=1000, this is 3000 bits.

What if we count the frequency of each letter and have something like the following?

A: 45% B:10% C:10% D:20% E:10% F:5%

Now if we assign the following codes:

0	A	45%
100	B	10%
101	C	10%
111	D	20%
1100	E	10%
1101	F	5%

Notice we need 4 bits to represent F now, but the most common character, A, is represented by just 1 bit.
Also note that since we are using a variable number of bits, this messes up the counting somewhat. I can't use 110 to represent D, since then if a 110 popped up we can't tell if this is referring to D or E, since both start with 110 – we need unique prefixes.

For example to store ABFA is: 010011010

Now, to store 1000 characters with this frequency and encoding scheme requires:
450*1 + 3*100 + 3*100 + 3*200 + 4*100 + 4*50 = 2250 bits. 25% improvement over before.

Question: We can find frequencies easily in O(n) time by linearly scanning and counting up the number of occurences of each token. How do we determine what codes should be assigned each character?

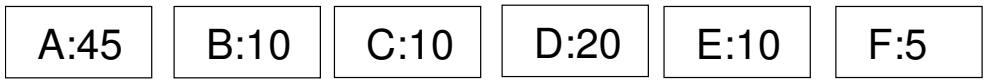
Idea: Count up frequencies, and build up trees by extracting minimum.

```
Huffman(S,f)        ; S = string of characters to encode.
                    ; F=frequences of each char
    n ← |S|         ; Make each character a 'node'
    Q ← S           ; Priority queue using the frequency as key
    for j ← 1 to n-1 do
        z ← Allocate-Node()
        x ← left[z] ← Extract-Min(Q)
        y ← right[z] ← Extract-Min(Q)
        f[z] ← f[x]+f[y]                    ; update frequencies
        Insert(Q,z)
    return Extract-Min(Q)
```

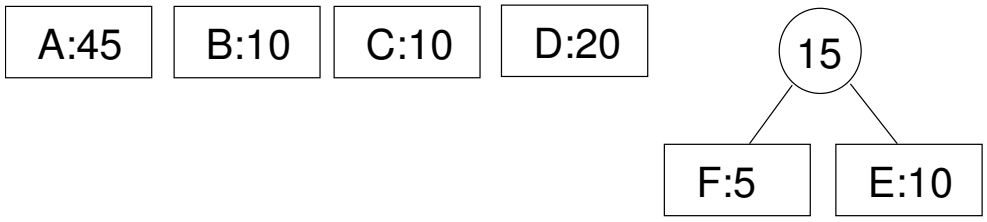
Example:

First make each character a node by itself.

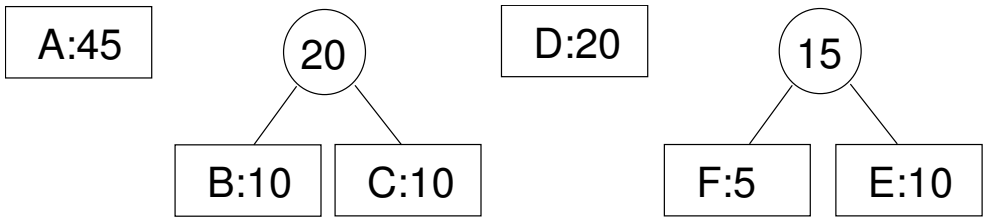
A: 45% B:10% C:10% D:20% E:10% F:5%



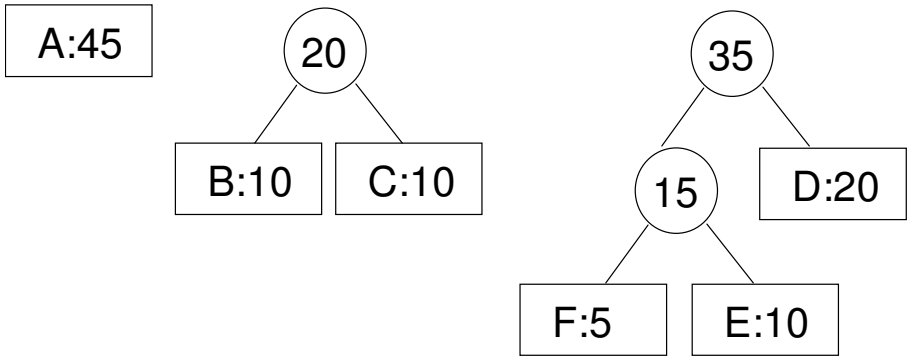
Extract the minimum and join together. These will end up as leaves farther down the tree.
Mins=F and E. Put sum as the new frequency. Put minimum to the left.



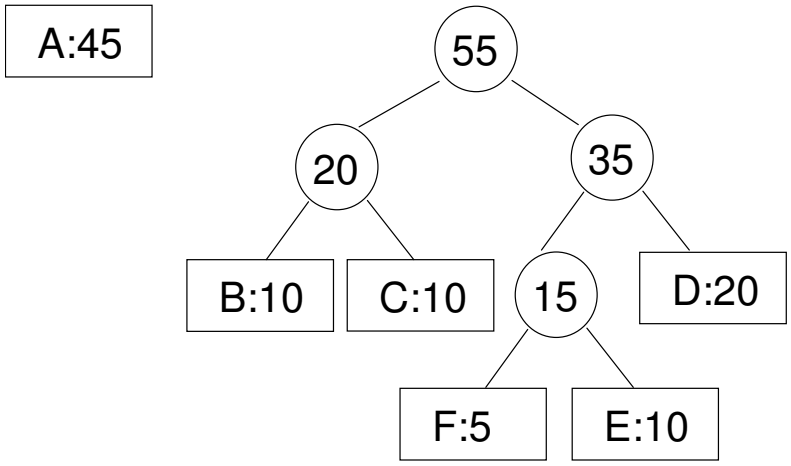
Repeat process: Extract min, B and C, and put sum as new frequency:



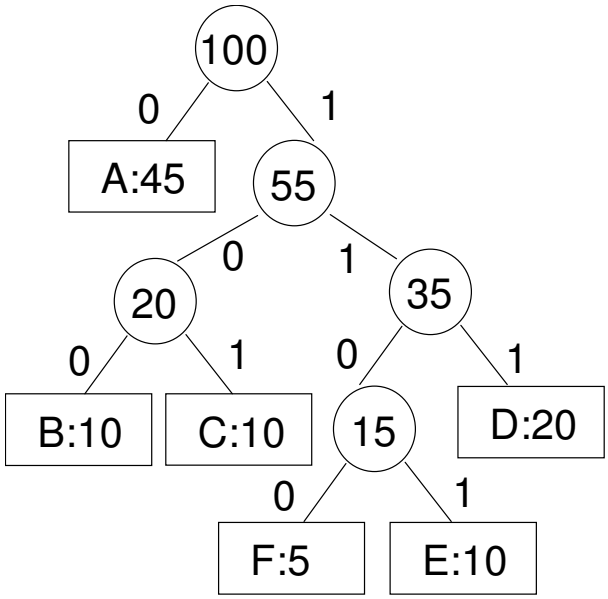
Repeat process: Extract D and 15 as min, put sum as new frequency:



Repeat process: Extract 20 and 35 as min, put sum as new frequency:



Finally, combine with A and assign 0,1 to edges:



Travel down tree to get the code:

A	0
B	100
C	101
D	111
E	1100
F	1101

We're done!

Correctness: This we will not prove, but the idea is to put the rarest characters at the bottom of the tree, and build the tree so we will not have any prefixes that are identical (guaranteed in tree merging step). By putting the rarest characters at the bottom, they will have the longest codes and the most frequent codes will be at the top. This algorithm produces an optimal prefix code, but not necessarily the most optimal compression possible.

Runtime of Huffman's algorithm: If Q implemented as a binary heap then the extract operation takes lgn time. This is inside a for-loop that loops n times, resulting in O(nlgn) runtime.