Huffman coding

Definition:

Huffman coding assigns codes to characters such that the length of the code depends on the relative frequency or weight of the corresponding character. Huffman codes are of variable-length, and prefix-free (no code is prefix of any other). Any prefix-free binary code can be visualized as a binary tree with the encoded characters stored at the leaves.

Huffman coding tree or **Huffman tree** is a full binary tree in which each leaf of the tree corresponds to a letter in the given alphabet.

Define the weighted path length of a leaf to be its weight times its depth. The Huffman tree is the binary tree with minimum external path weight, i.e., the one with the minimum sum of weighted path lengths for the given set of leaves. So the goal is to build a tree with the minimum external path weight.

See an example below:

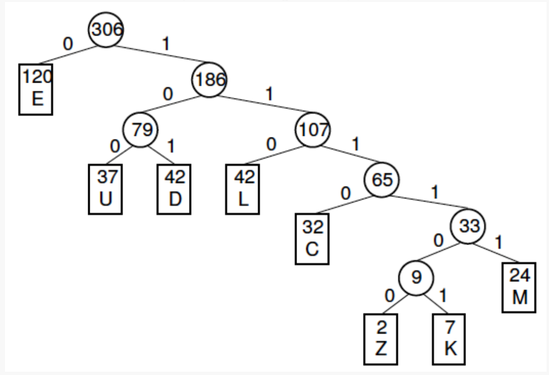
Letter freqency table

Letter	Z	K	M	C	U	D	L	E
Frequency	2	7	24	32	37	42	42	120

Huffman code

Letter	Freq	Code	Bits
E	120	0	1
D	42	101	3
L	42	110	3
U	37	100	3
C	32	1110	4
M	24	11111	5
K	7	111101	6
Z	2	111100	6

The Huffman tree (Shaffer Fig. 5.24)



Three problems:

- Problem 1: Huffman tree building
- Problem 2: Encoding
- Problem 3: Decoding

Problem 2: Encoding

Encoding a string can be done by replacing each letter in the string with its binary code (the Huffman code).

Examples:
DEED 10100101 (8 bits)
MUCK 11111100111011101 (18 bits)

Problem 3: Decoding

Decoding an encoded string can be done by looking at the bits in the coded string from left to right until a letter decoded.
10100101 -> DEED

Problem 1: Huffman tree building

A simple algorithm:

1. Prepare a collection of *n* initial Huffman trees, each of which is a single leaf node. Put the *n* trees onto a **priority queue** organized by weight (frequency).
2. Remove the first two trees (the ones with lowest weight). Join these two trees to create a new tree whose root has the two trees as children, and whose weight is the sum of the weights of the two children trees.
3. Put this new tree into the priority queue.
4. Repeat steps 2-3 until all of the partial Huffman trees have been combined into one.

It's a greedy algorithm: at each iteration, the algorithm makes a "greedy" decision to merge the two subtrees with least weight. Does it give the desired result?

- Lemma: Let x and y be the two least frequent characters. There is an optimal code tree in which x and y are siblings whose depth is at least as any other leaf nodes in the tree.
- Theorem: Huffman codes are optimal prefix-free binary codes (The greedy algorithm builds the Huffman tree with the minimum external path weight for a given set of letters).

See **an implementation at github** (**HuffTree.java MinHeap.java**)