A quick tutorial on generating a huffman tree

Lets say you have a set of numbers and their frequency of use and want to create a huffman encoding for them:

FREQUENCY	VALUE
5	1
7	2
10	3
15	4
20	5
45	6

Creating a huffman tree is simple. Sort this list by frequency and make the two-lowest elements into leaves, creating a parent node with a frequency that is the sum of the two lower element's frequencies:

```
   12:*
   /  \
 5:1   7:2
```

The two elements are removed from the list and the new parent node, with frequency 12, is inserted into the list by frequency. So now the list, sorted by frequency, is:

```
10:3
12:*
15:4
20:5
45:6
```

You then repeat the loop, combining the two lowest elements. This results in:

```
   22:*
   /   \
10:3   12:*
       /   \
     5:1   7:2
```

and the list is now:

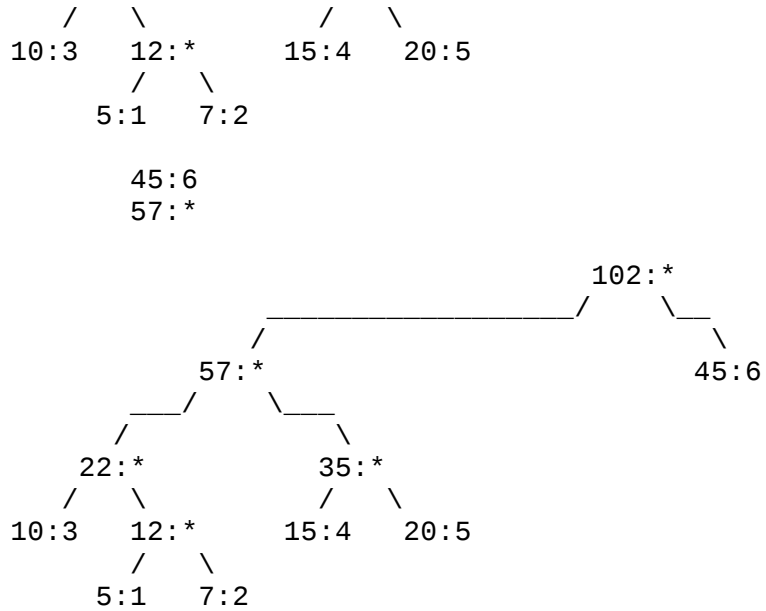
```
15:4
20:5
22:*
45:6
```

You repeat until there is only one element left in the list.

```
   35:*
   /   \
 15:4  20:5

   22:*
   35:*
   45:6

       57:*
   ___/    \___
  /            \
22:*          35:*
```



Now the list is just one element containing 102:*, you are done.

This element becomes the root of your binary huffman tree. To generate a huffman code you traverse the tree to the value you want, outputing a **0** every time you take a lefthand branch, and a **1** every time you take a righthand branch. (normally you traverse the tree backwards from the code you want and build the binary huffman encoding string backwards as well, since the *first* bit must start from the top).

Example: The encoding for the value **4** (15:4) is **010**. The encoding for the value **6** (45:6) is **1**

Decoding a huffman encoding is just as easy : as you read bits in from your input stream you traverse the tree beginning at the root, taking the left hand path if you read a **0** and the right hand path if you read a **1**. When you hit a leaf, you have found the code.

Generally, any huffman compression scheme also requires the huffman tree to be written out as part of the file, otherwise the reader cannot decode the data. For a static tree, you don't have to do this since the tree is known and fixed.

The easiest way to output the huffman tree itself is to, starting at the root, dump first the left hand side then the right hand side. For each node you output a **0**, for each leaf you output a **1** followed by N bits representing the value. For example, the partial tree in my last example above using 4 bits per value can be represented as follows:

```
000100 fixed 6 bit byte indicates how many bits the value
       for each leaf is stored in.  In this case, 4.
0      root is a node
       left hand side is
10011  a leaf with value 3
       right hand side is
0      another node
       recurse down, left hand side is
10001  a leaf with value 1
       right hand side is
10010  a leaf with value 2
       recursion return
```

So the partial tree can be represented with **000100010011010001100 10**, or 23 bits. Not bad!