# Huffman Code: Example 1

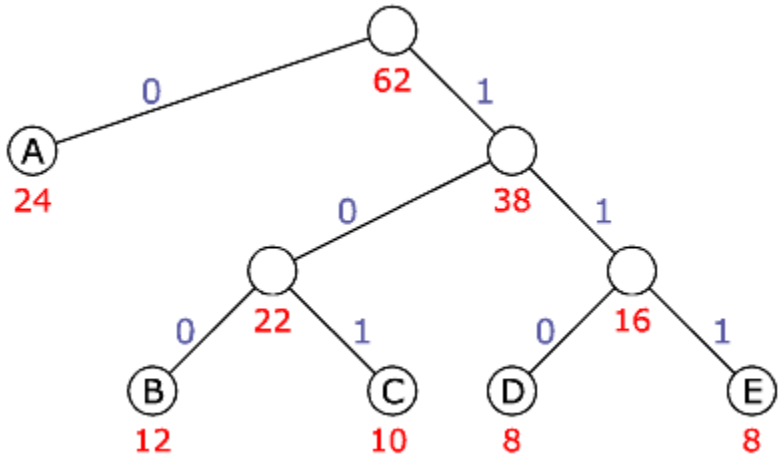
The following example bases on a data source using a set of five different symbols. The symbol's frequencies are:

| Symbol | Frequency |
|---|---|
| A | 24 |
| B | 12 |
| C | 10 |
| D | 8 |
| E | 8 |

----> total 186 bit
(with 3 bit per code word)

The two rarest symbols 'E' and 'D' are connected first, followed by 'C' and 'D'. The new parent nodes have the frequency 16 and 22 respectively and are brought together in the next step. The resulting node and the remaining symbol 'A' are subordinated to the root node that is created in a final step.

## Code Tree according to Huffman



```
Symbol Frequency  Code  Code   total
                        Length  Length
   A      24      0      1      24
   B      12      100    3      36
   C      10      101    3      30
   D       8      110    3      24
   E       8      111    3      24
-------------------------------------
    ges. 186 bit          tot. 138 bit
    (3 bit code)
```

# Huffman Code: Example 2

| 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
```

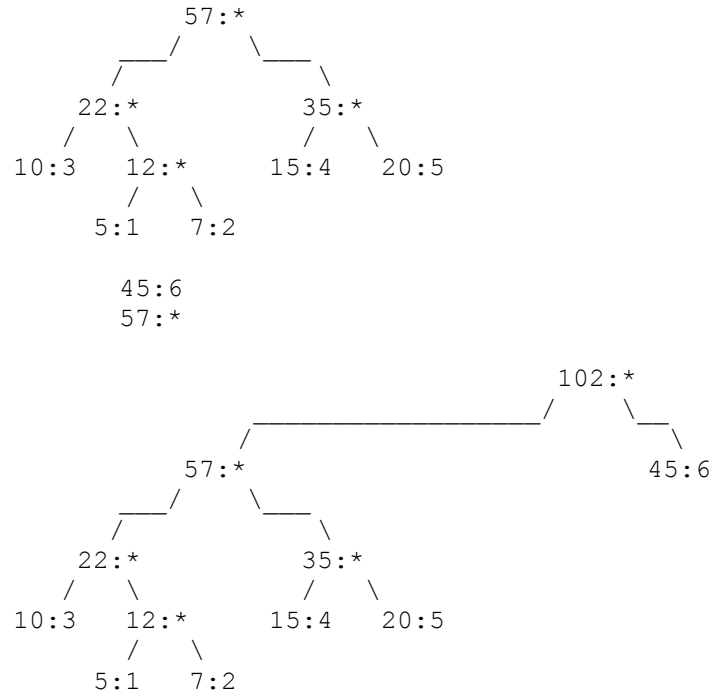
and the list is now:

```
        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
```




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 the 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.

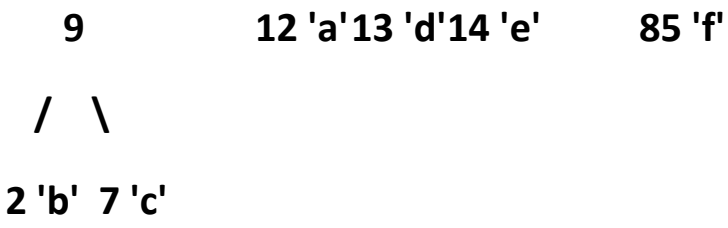
# Huffman Code: Example 3

Let's assume that after scanning a file we find the following character frequencies:

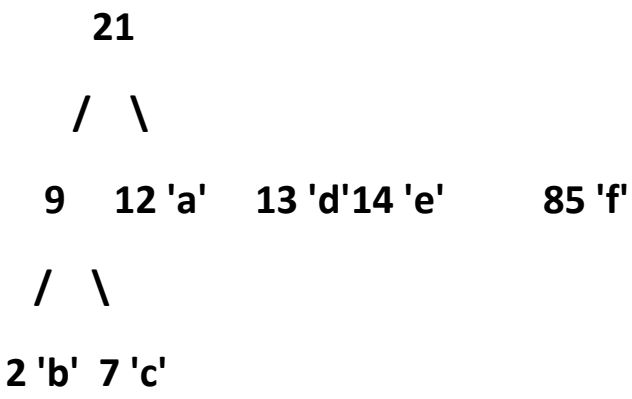
| Character | Frequency |
|---|---|
| 'a' | 12 |
| 'b' | 2 |
| 'c' | 7 |
| 'd' | 13 |
| 'e' | 14 |
| 'f' | 85 |

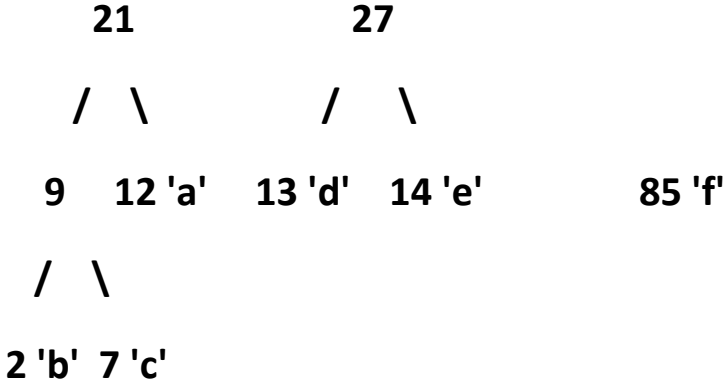
Now, create a binary tree for each character that also stores the frequency with which it occurs.

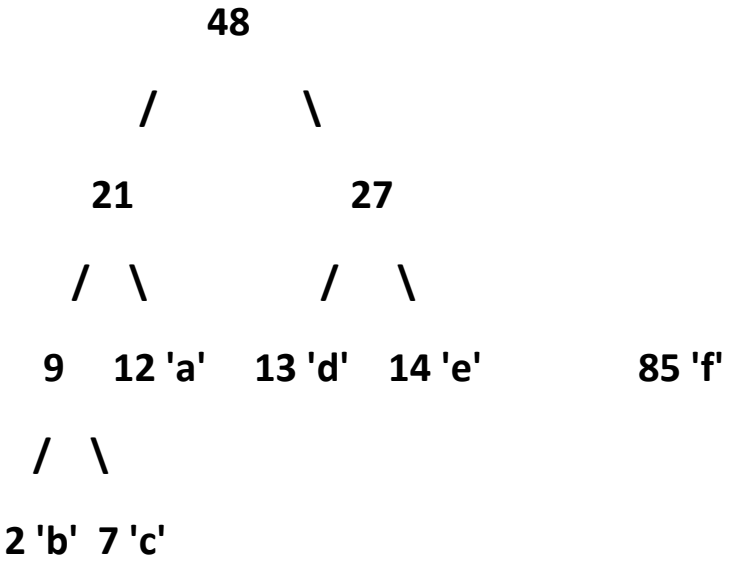
The algorithm is as follows: Find the two binary trees in the list that store minimum frequencies at their nodes. Connect these two nodes at a newly created common node that will store NO character but will store the sum of the frequencies of all the nodes connected below it. So our picture looks like follows:

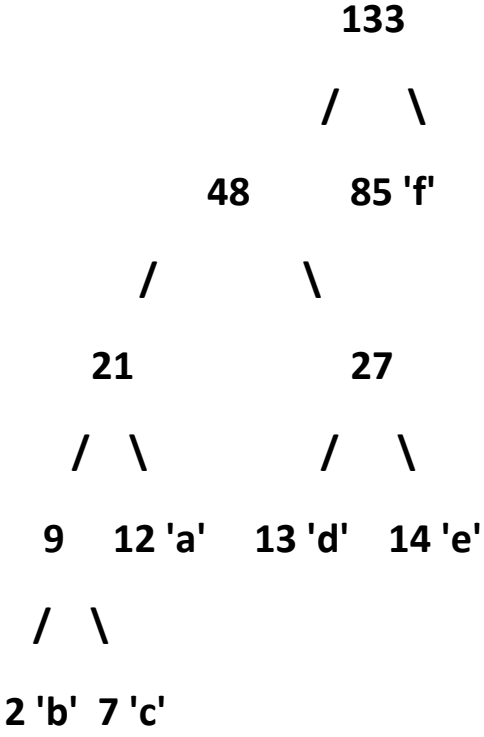



Now, repeat this process until only one tree is left:






Once the tree is built, each leaf node corresponds to a letter with a code. To determine the code for a particular node, walk a standard search path from the root to the leaf node in question. For each step to the left, append a 0 to the code and for each step right append a 1. Thus for the tree above we get the following codes:

| Letter | Code |
|---|---|
| 'a' | 001 |
| 'b' | 0000 |
| 'c' | 0001 |
| 'd' | 010 |
| 'e' | 011 |
| 'f' | 1 |

## Calculating Bits Saved

All we need to do for this calculation is figure out how many bits are originally used to store the data and subtract from that how many bits are used to store the data using the Huffman code.

In the first example given, since we have six characters, let's assume each is stored with a three bit code. Since there are 133 such characters, the total number of bits used is 3*133 = 399.

Now, using the Huffman coding frequencies we can calculate the new total number of bits used:

| Letter | Code | Frequency | Total Bits |
|---|---|---|---|
| 'a' | 001 | 12 | 36 |
| 'b' | 0000 | 2 | 8 |
| 'c' | 0001 | 7 | 28 |
| 'd' | 010 | 13 | 39 |
| 'e' | 011 | 14 | 42 |
| 'f' | 1 | 85 | 85 |
| Total | | | 238 |

Thus, we saved 399 - 238 = 161 bits, or nearly 40% storage space.