Huffman Coding

(source: Introduction to Algorithms by Cormen, Leiserson, Rivest)

Huffman Coding is a popular technique for data compression. It the frequencies of occurrence of each character in a file to come up with an optimal way of representing each character as a binary string.

Suppose you are given a text file that contains one thousand characters and you wish to compress it (in other words store it in a more compact way). Let's assume that the only characters appearing in the file are a, b, c, d, e, and f. We want to come up with a binary code to represent each character uniquely. One method would be to use a fixed-length code. For example to encode six characters we would need 3 bits for each one (why 3? Because using 2 bits we can only come up with four codes, 00, 01, 10, 11. With 3 bits we can create a total of $3^2 = 9$ codes. We need 6 codes, so 3 bits are sufficient). A possible encoding would then be a=000, b=001, c=010, d=011, e=100, f=101. Since we have a total of 1,000 characters, our compressed file will contain 3,000 bits.

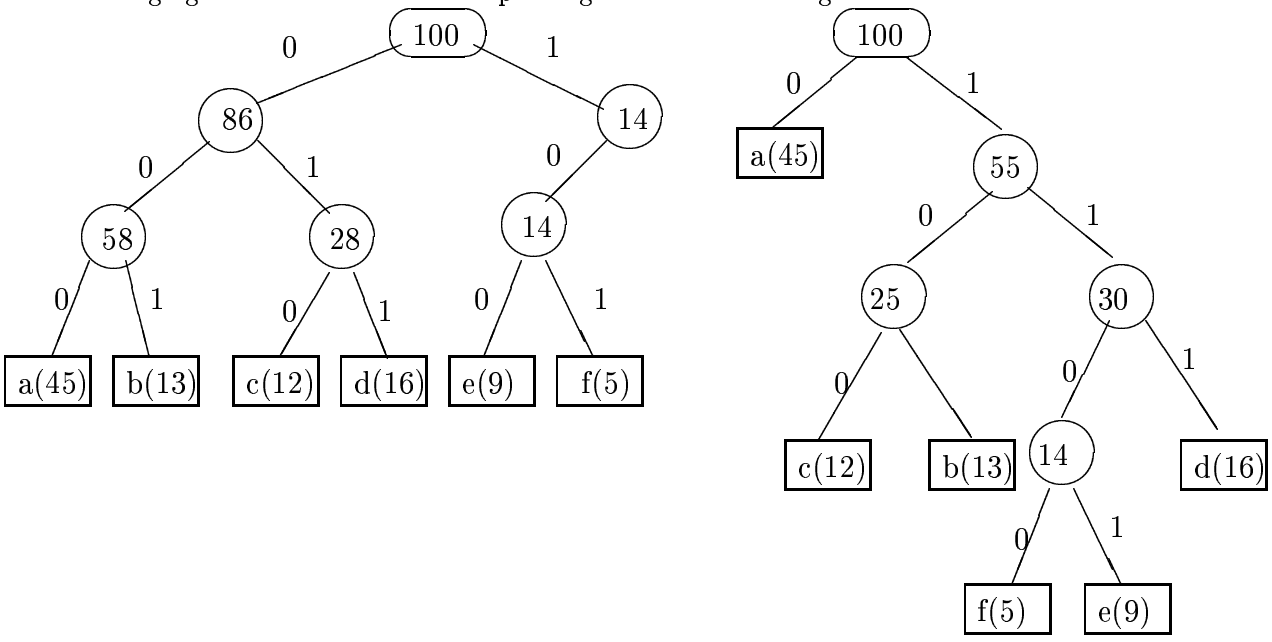
But we can do better than that if we take advantage of the frequencies of occurrence of the characters. Surely if we use shorter codes for characters that appear often and longer codes for characters that appears rarely, we will gain some space.

For example, if the frequencies are f(a)=450, f(b)=130, f(c)=120, f(d)=160, f(e)=90, f(f)=50 we could use the following encoding: a=0, b=101, c=100, d=111, e=1101, f=1100 and it would result in a compressed file of 2240 bits.

If we can make sure that no code is a prefix of another code, then we will be able to decode a file easily. For example, 1000101 can be nothing else but the word *cab*.

A code for a file can be represented as a binary tree that has as many leaves as there are (different) characters in the file. For example, a tree for the file described above would have six leaves. Each internal node has two children and it is labeled with the sum of the frequencies of the leaves in its subtree. Consider an imaginary label for each branch, which is zero if it's a left branch and 1 if it is a right branch. Concatenating these labels from the root to a leaf will give us the code for the character corresponding to that leaf.

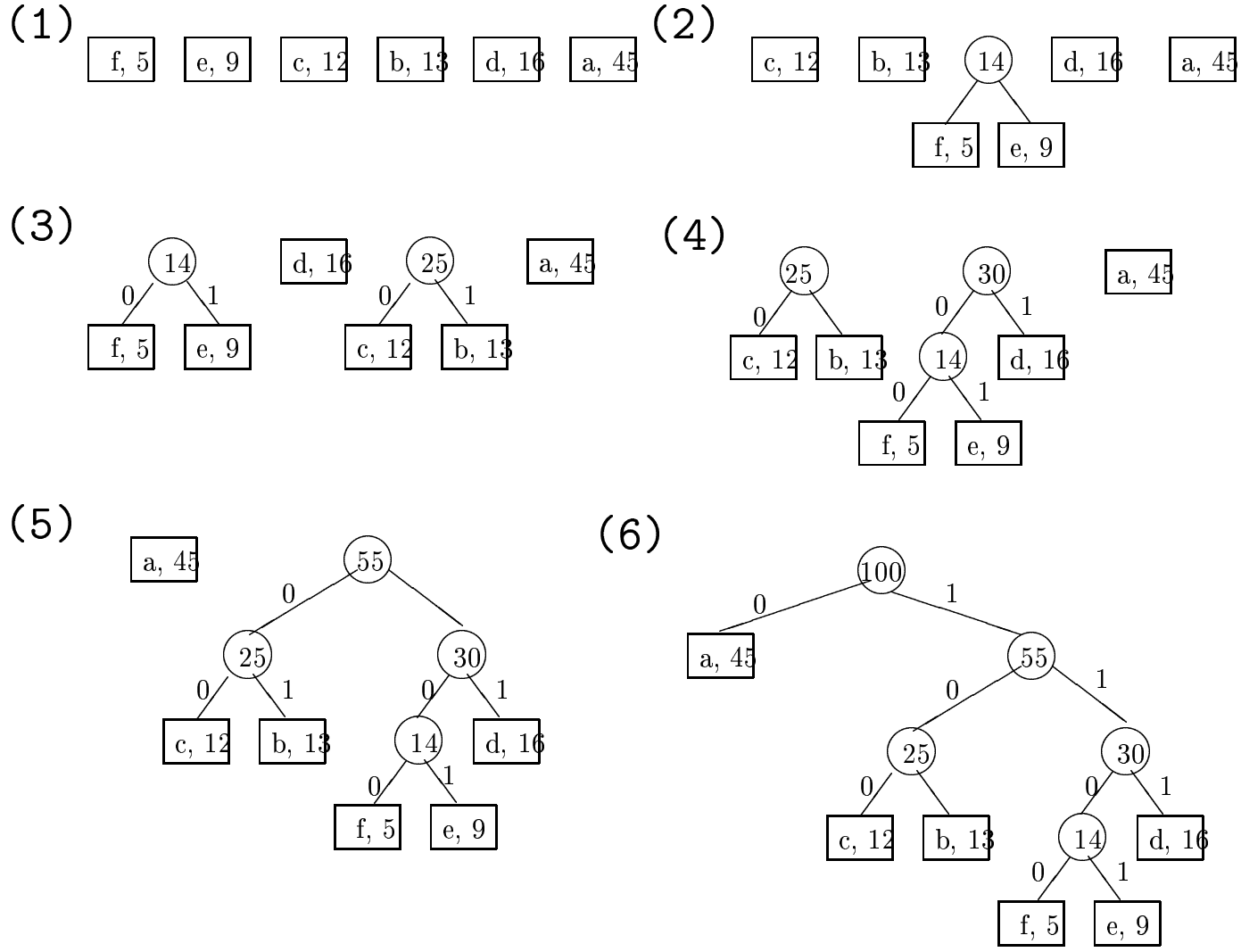
The following figure shows the trees corresponding to the two encodings described above.



The algorithm for creating a Huffman code creates a binary tree in the following way:

1. First find the number of occurrences (frequency) for each character.
2. For each (different) character create a node. The frequency of that character should be stored at the node.
3. Find the two least frequent nodes, x, y.
4. Create a new node z and make its children x and y. The frequency of z is the sum of the frequencies of its children.
5. Repeat (considering only nodes of zero depth; for example, x and y will not be examined any more) until there is only one node left. This is the root of the tree.

The following figure shows the steps of the Huffman algorithm for the file described earlier.



Since at each step we need the two smallest frequencies, we can use a priority queue (implemented with a min-heap) to hold the nodes.

The algorithm is as follows:

Input: a collection C of objects containing a character and its frequency.
Output: the root of a Huffman tree
Uses: priority queue Q
$n = |C|$
$Q = C$
for i=1 to n-1
    z = new treenode
    left(z) = ExtractMin(Q)
    right(z) = ExtractMin(Q)
    frequency(z) = frequency(left(z)) + frequency(right(z))
    Insert(Q, z)
return ExtractMin(Q)

Running time: $(O(n \lg n))$