Huffman Codes

An Application of Binary Trees and Priority Queues

This week, we consider an application of binary trees and priority queues. The algorithm we develop here is useful for file compression.

The normal ASCII character set consists of roughly 100 "printable" characters. In order to represent these characters in binary code, 7 bits are required. Seven bits allow the representation of 128 characters, so the ASCII character set adds some "nonprintable" characters. An eighth bit is added as a parity check.

Suppose we have a file that contains only the characters *a, b, c, d, e*. Suppose further, that the file has 12 *a*'s, 30 *b*'s, 14 *c*'s, 8 *d*'s, and 25 *e*'s. As Code 1 of the following table shows, this file can be represented in the binary format with 282 bits, since there are 94 characters and each character requires three bits in the representation.

Character	Frequency	Code1	Code2
a	12	000	000
b	35	001	11
c	14	010	01
d	8	011	001
e	25	100	10
Total bits		282	208

In practice, files can be very big. Moreover, many large data files have large occurrences of numbers, blanks, and newlines, but few characters such as a's and b's. Since disk space is precious, one might wonder if it would be possible to provide a better code and reduce the total number of bits required for storing the file. The answer is that this is possible, provided we use character codes that are different in length. The general idea is to allow the code length to vary from character to character and to make sure that the characters that occur more frequently have shorter codes.

However, when we use variable length character codes, we must ensure that **no code for a character is the prefix of the code for any other character**. This prefix property allows us to decode a string of 0s and 1s by repeatedly deleting prefixes of the string that are codes for characters.

In the above table, Code 1 has the prefix property. Code 2 also has the prefix property.

The problem:

Given a set of characters and their frequencies of occurrence in a file, find a code with the prefix property such that the number of bits in in encoded (compressed) file is a minimum.

The algorithm to find optional prefix codes was given by Huffman in 1952. Thus this coding system is commonly referred to as a Huffman code.

Huffman's algorithm is one technique for finding optional prefix codes. The algorithm works by selecting two characters *a* and *b* having the lowest frequencies and replacing them with a single (imaginary) character, say *x*, whose frequency of occurrence is the sum of frequencies for *a* and *b*. We then find an optimal prefix code for this smaller set of characters, using this procedure recursively. The code for the original character set is obtained by using the code for *x* with a 0 appended for "*a*" and a 1 appended for "*b*"

We can think of prefix codes as paths in binary trees. For example, see Figure 1.

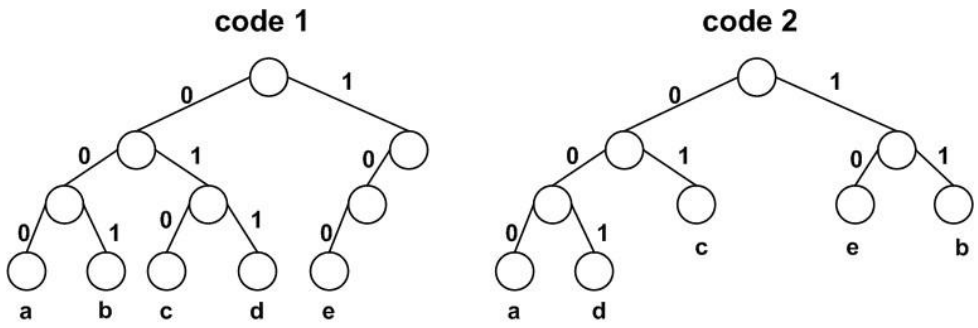

Figure 1: Code 1 and Code 2

The prefix property guarantees that no character can have a code that is an interior node, and conversely, labeling the leaves of any binary tree with characters gives us a code with the prefix property for these characters.

Huffman's algorithm is implemented using a *forest* (disjoint collection of trees), each of which has its leaves labeled by characters whose codes we desire to select and whose roots are labeled by the sum of the frequencies of all the leaf labels. We call this sum the weight of the tree.

Initially each character is in a one-node tree by itself and when the algorithm ends, there will be only one tree, with all the characters as its leaves. In this final tree, the path from the root to any leaf represents the code for the label of that leaf.

The essential step of the algorithm is to select the two trees in the forest that have the smallest weights (break ties arbitrarily). Combine these two trees into one, whose weight is the sum of the weights of the two trees. To combine the trees, we create a new node, which becomes the root and has the roots of the two given trees as left and right children (which is which doesn't matter). This process continues until only one tree remains. Figure 2 illustrates the Huffman algorithm for five characters a, b, c, d, e with frequencies of occurrence 3, 5, 4, 6 and 12

respectively.

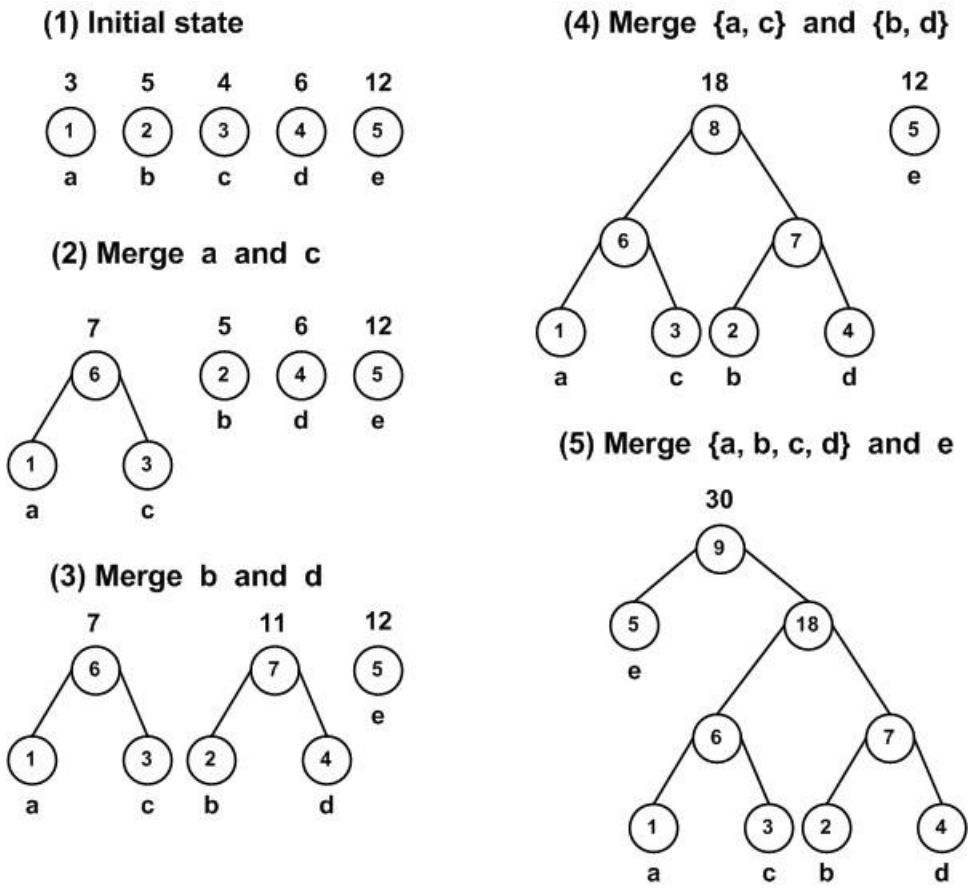

Figure 2: An example of Huffman algorithm

Implementation of Huffman algorithm

Let us now describe the needed data structures using language-agnostic pseudo-code.

First, we shall use a sequential container (list/array/vector) named *tree* of the type

```
tree_node {
      leftchild
      rightchild
      parent
}
```

to represent binary trees. Each field is an index. Parent index facilitate finding paths from leaves to the root, so we can discover the code for a character.

Second, we use another sequential container named *alphabet* of type

```
symbol_info {
```

```
        symbol
        frequency
        leaf
    }
```

to associate, with each symbol of the alphabet being encoded, its corresponding leaf. This array also records the frequency of each character. Third, we need something named *forest*, a priority queue of data structures that represent the trees themselves. The type inside the priority queue is

```
    forest_roots {
        weight
        root
    }
```

An example for these three data structures is shown below:

	Forest		**Alphabet**			**Tree**		
	weight	root	symbol	freq	leaf	leftchild	rightchild	parent
1	3	1	b	5	2	0	0	0
2	5	2	a	3	1	0	0	0
3	4	3	c	4	3	0	0	0
4	6	4	d	6	4	0	0	0
5	12	5	e	12	5	0	0	0

Here is a pseudo-code sketch of the program to build the Huffman tree:

```
while (there is more than one tree in the forest) {
    forest_roots least = forest.deletemin();
    forest_roots second = forest.deletemin();
    /* (X) */ create a new node with left child least.root and right
child second.root;
    forest_roots newroot =
        {least.weight + second.weight, new node};
    FOREST.insert(newroot);
}
```