TABLE 8.8
Frequency of Letters in English Text

Symbol	Frequency	Symbol	Frequency	Symbol	Frequency
␣	186	h	47	g	15
e	103	d	32	p	15
t	80	l	32	b	13
a	64	u	23	v	8
o	63	c	22	k	5
i	57	f	21	j	1
n	57	m	20	q	1
s	51	w	18	x	1
r	48	y	16	z	1

FIGURE 8.35
Huffman Tree Based on Frequency of Letters in English Text

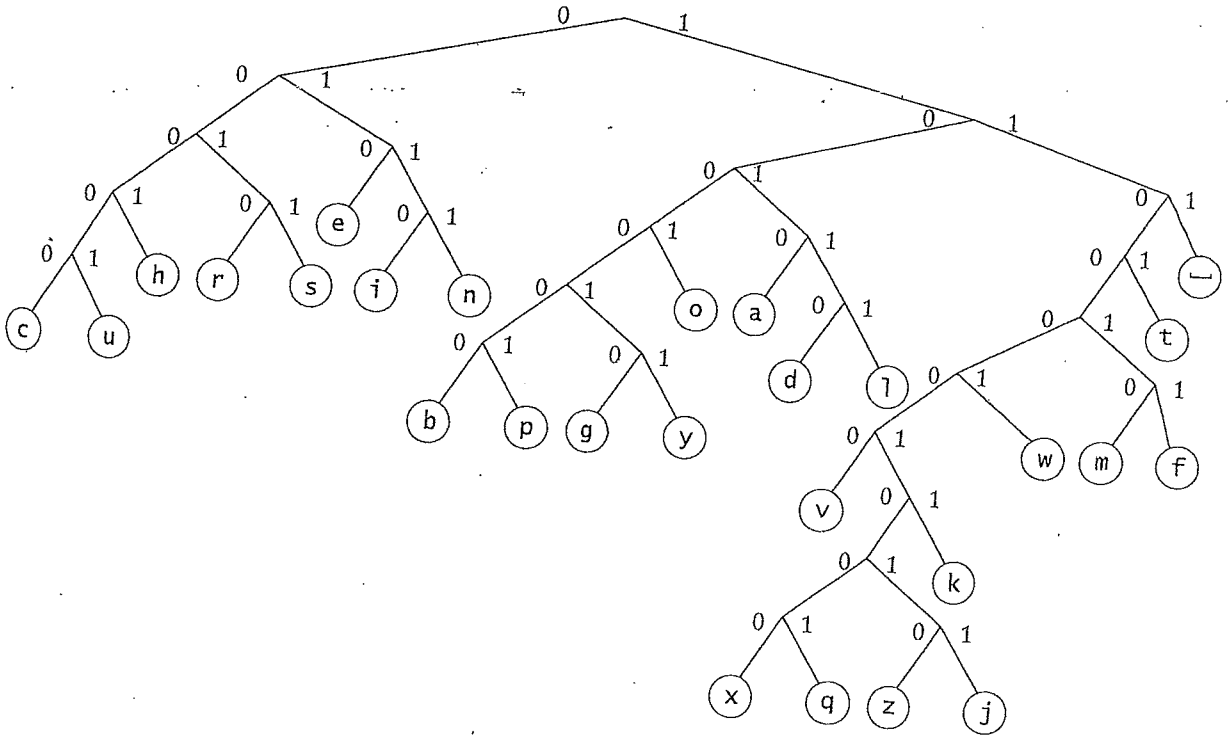


TABLE 6.5 Variable-Length Codes

A	B	C	D	E	F	G
01	001	11	1001	1000	011	111

TABLE 6.6 Example Symbols and Frequencies

Symbol (S_i)	Relative Frequency (P_i)
A	6
B	5
C	4
D	1
E	2
F	2
G	3

for those that are rarer. Some compression programs, such as the Unix utilities *pack* and *compact*, work in this manner.

The set of codes that we use must result in uniquely decodable files; that is, there must be an unambiguous decoding of the file. If the codes are all the same length this is not a problem. The ASCII for BEAD is the following 32-bit string made up o four 8-bit character codes.

```
01000010010001010100000101000100
```

Consider the characters and binary codes shown in Table 6.5.

Using the codes in Table 6.5, BEAD is represented as 0011000011001. However, this binary string is ambiguous because it can be decoded both as BEAD and as BEFB. The problem arises because one code (01, the code for *A*) is the prefix of another (011, the code for *F*). We can eliminate the possibility of ambiguity and also arrive at an encoding that is in some sense optimal by employing a technique devised by Huffman (1952) that uses a binary trie.

Assume we have symbols $S_1 \ldots S_n$ with associated relative frequencies of $P_1 \ldots P_n$. For illustrative purposes, we use the symbols and relative frequencies shown in Table 6.6.

Huffman's technique for deriving codes works as follows.

```
Make each symbol Si a single node with associated weight Pi.
While (more than one parentless node)
    {
      Identify the two parentless nodes with the smallest
          weights (breaking ties arbitrarily)
      Create a new node, make it the parent of the two selected nodes
          and associate with it the sum of the weights of its children
    }
Label the branches of the tree: for each nonleaf node label one of its
      Branches with '0' and the other with '1'.
The code for a symbol is constructed by concatenating the characters
      ('0' or '1') on the path from the root to the symbol.
```

For example, from the data in Table 6.6, we first combine node D (weight 1) with either E or F (weight 2). Choosing E yields the structure shown in Figure 6.46.

We now combine node F (weight 2) with one of the nodes with weight 3. Choosing G gives us the tree shown in Figure 6.47.

The two nodes with the smallest weights now are node C and the node that resulted from our first construction. Joining these yields the structure shown in Figure 6.48.

We continue in this manner, finally arriving at the tree shown in Figure 6.49 (other trees are possible depending on how we break ties).

We now label the links. One of the links from a parent to its child will be labeled "1" and the other will be labeled "0," but it does not matter which is which. Figure 6.50 shows a possible tree after this labeling has been done.

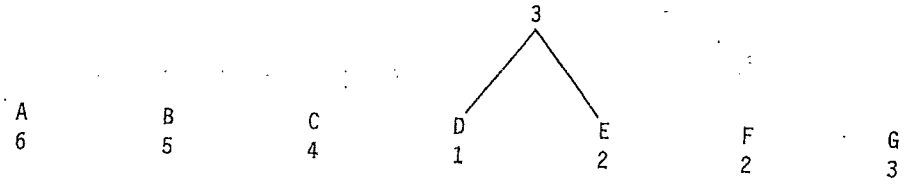


Figure 6.46 • Tree construction: Step 1

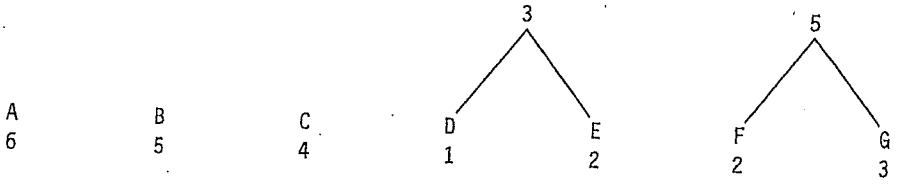


Figure 6.47 • Tree construction: Step 2

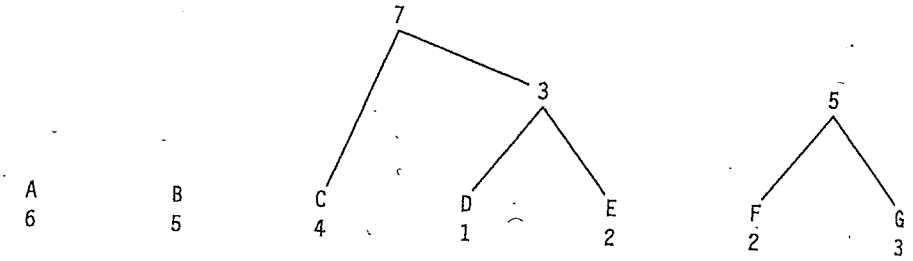


Figure 6.48 • Tree construction: Step 3

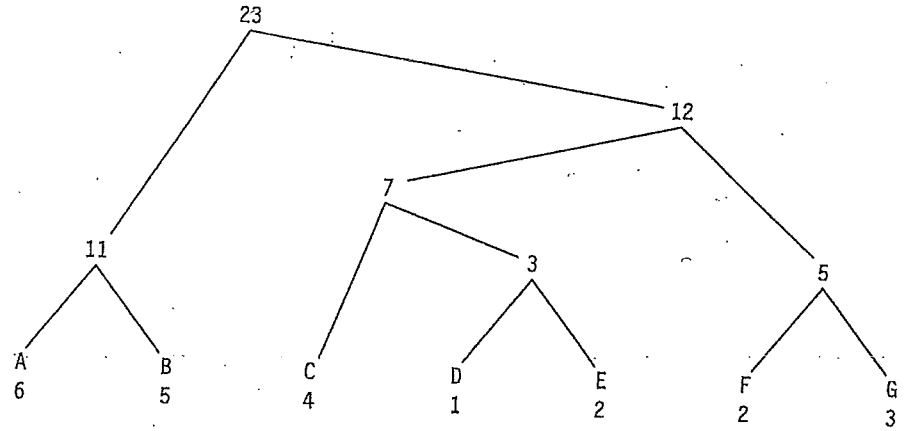


Figure 6.49 • Huffman tree before labeling

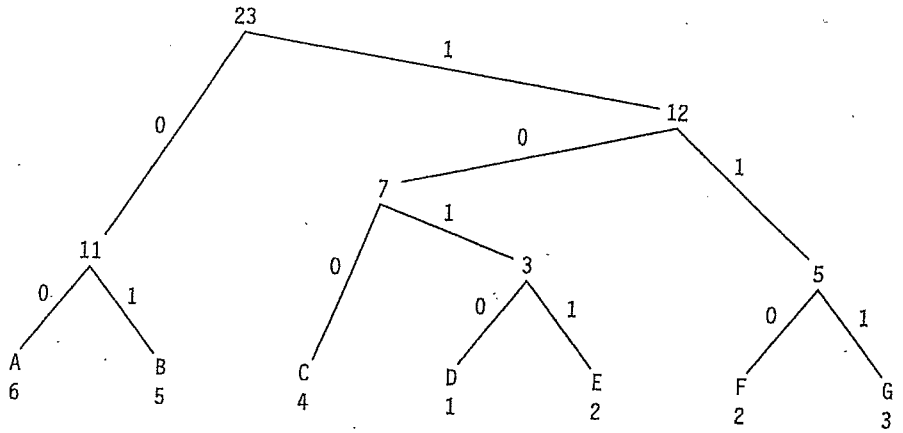


Figure 6.50 • Labeled Huffman tree

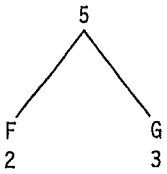
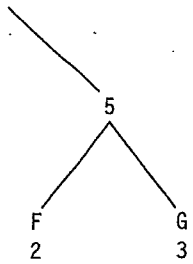


TABLE 6.7 Example Symbols and Codes

Symbol	Code
A	00
B	01
C	100
D	1010
E	1011
F	110
G	111



The codes for each symbol are now derived by following paths from the root to the leaves. In essence, we have a trie because the code is the complete path and each edge represents only part of the symbol. The codes in this example are shown in Table 6.7.

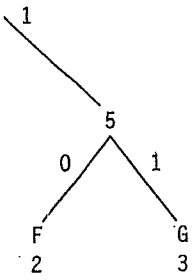
To encode a character, we just replace it with the corresponding code, thus BEADED becomes 01101100101010111010.

To decode a string of bits, we start at the beginning of the string and with a pointer at the top of the code tree. When reading bits, we follow the appropriate paths down the tree. Whenever we reach a leaf, we output the corresponding symbol and reset the pointer to the root of the tree. Try it for yourself using the tree in Figure 6.50 to confirm that 100001101011 decodes to CAFE.

Optimality

The average weighted code length is determined using

$$\frac{\sum\left(P_i * codelength(S_i)\right)}{\sum P_i}$$



For our example, the average weighted code length is (6 * 2 + 5 * 2 + 4 * 3 + 1 * 1 + 2 * 2 + 2 * 3 + 3 * 3) / 25 = 61 / 25 = 2.44 bits.[11] Huffman coding is optimal in that there is no other assignment of codes to symbols that will have a shorter average weighted code length.

Static Versus Dynamic Huffman Encoding

What is outlined in the previous section is *static* Huffman encoding, in which a particular symbol has the same encoding throughout a file. We use some frequency data to determine the codes. For example, in a compression program we might read the file to be compressed twice: once to get the frequency distribution and a second