12.2 Application: Huffman Codes

In the preceding section we indicated how a binary tree can be used in various encoding and decoding problems. In particular we showed part of a binary tree for Morse code which represents each character by a sequence of dots and dashes. Unlike ASCII and EBCDIC coding schemes, in which the length of the code is the same for all characters (eight bits), Morse code uses variable-length sequences. In this section we consider another coding scheme that uses variable-length codes.

The basic idea in these variable-length coding schemes is to use shorter codes for those characters that occur more frequently than for those used less frequently. For example, 'E' is encoded in Morse code as a single dot, whereas 'Z' is represented as – – · · . The objective is to minimize the expected length of the code for a character.

To state the problem more precisely, suppose that some character set $\{C_1, C_2, \ldots, C_n\}$ is given and certain *weights* $w_1, w_2, \ldots, w_n$ are associated with these characters; w_i is the weight attached to character C_i and is some measure (e.g., probability or relative frequency) of how frequently this character occurs in messages to be encoded. If $l_1, l_2, \ldots, l_n$ are the lengths of the codes for characters $C_1, C_2, \ldots, C_n$, respectively, then the ***expected length*** of the code for any one of these characters is given by

$$\text{expected length} = w_1 l_1 + w_2 l_2 + \cdots + w_n l_n = \sum_{i=1}^{n} w_i l_i$$

As a simple example, consider the five characters A, B, C, D, and E, and suppose they occur with the following weights (probabilities):

character	A	B	C	D	E
weight	0.2	0.1	0.1	0.15	0.45

In Morse code with a dot replaced by 0 and a dash by 1, these characters are encoded as follows:

Character	Code
A	01
B	1000
C	1010
D	100
E	0

Thus the expected length of the code for each of these five letters in this scheme is

$$0.2 \times 2 + 0.1 \times 4 + 0.1 \times 4 + 0.15 \times 3 + 0.45 \times 1 = 2.1$$

Another useful property that some coding schemes have is that they are ***immediately decodable***. This means that no sequence of bits that represents a character is a prefix of a longer sequence for some other character.

Consequently, when a sequence of bits is received that is the code for a character, it can be decoded as that character immediately, without our waiting to see if subsequent bits change it into a longer code for some other character. Note that the preceding Morse code scheme is not immediately decodable because the code for E (0) is a prefix of the code for A (01), and the code for D (100) is a prefix of the code for B (1000). (For decoding, Morse code uses a third "bit," a pause, to separate letters.) A coding scheme for which the code lengths are the same as in the preceding scheme and which is immediately decodable is as follows:

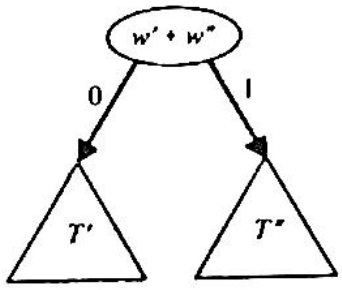
Character	Code
A	01
B	0000
C	0001
D	001
E	1

The following algorithm, given by D. A. Huffman in 1952, can be used to construct coding schemes that are immediately decodable and for which each character has minimal expected code length:

HUFFMAN'S ALGORITHM

(* Algorithm to construct a binary code for a given set of characters for which the expected length of the bit string for a given character is minimal. *)

1. Initialize a list of one-node binary trees containing the weights w_1, w_2, . . . , w_n, one for each of the characters C_1, C_2, . . . , C_n.
2. Do the following $n - 1$ times:
   a. Find two trees T' and T'' in this list with roots of minimal weights w' and w''.
   b. Replace these two trees with a binary tree whose root is $w' + w''$, whose subtrees are T' and T'', and label the pointers to these subtrees 0 and 1, respectively:

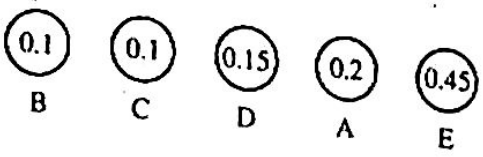


3. The code for character C_i is the bit string labeling a path from the root in the final binary tree to the leaf for C_i.

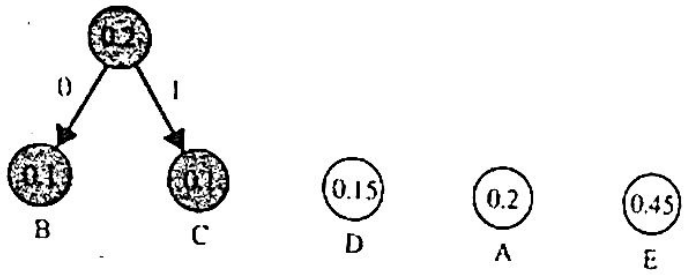
As an illustration of Huffman's algorithm, consider again the characters A, B, C, D, E with the weights given earlier.

representing E to give the final Huffman tree;

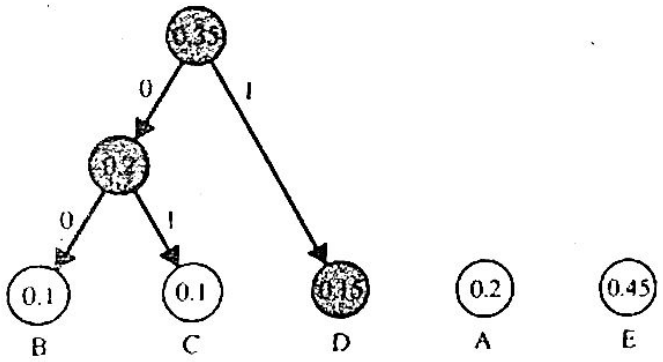
a list of one-node binary trees, one for each character:



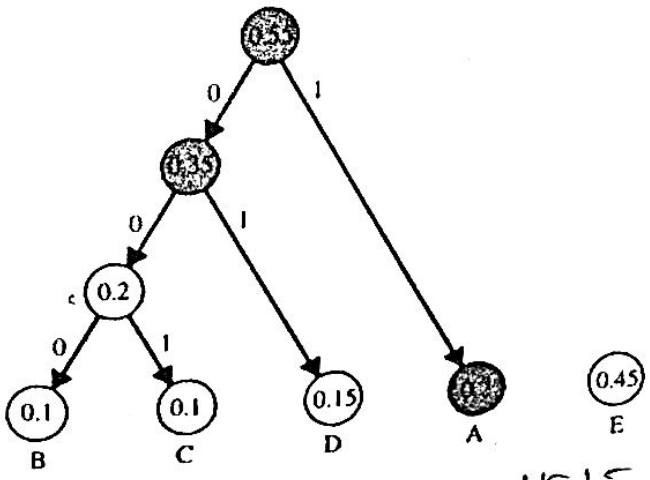
The first two trees to be selected are those corresponding to letters B and C, since they have the smallest weights, and these are combined to produce a tree having weight $0.1 + 0.1 = 0.2$ and having these two trees as subtrees:



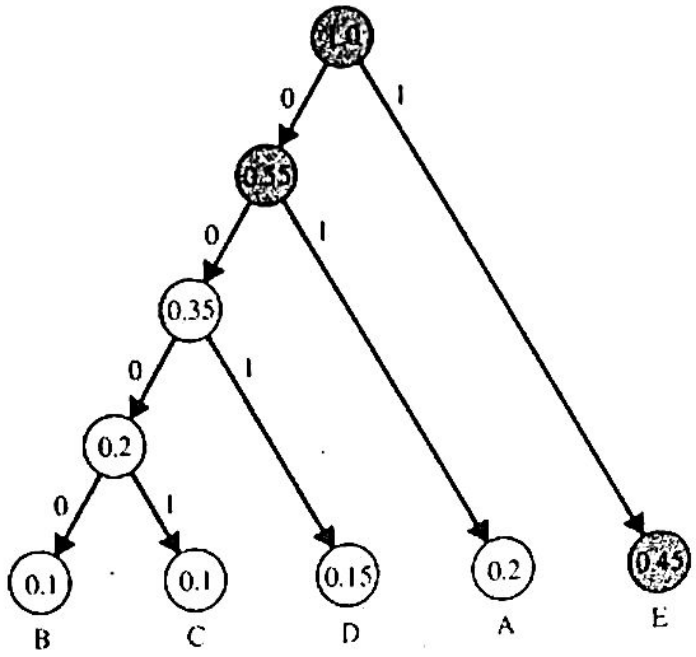
From this list of four binary trees, we again select two of minimal weights, the first and the second (or the second and the third), and replace them with another tree with weight $0.2 + 0.15 = 0.35$ and having these two trees as subtrees:



From this list of three binary trees, the first two have minimal weights and are combined to produce a binary tree with weight $0.35 + 0.2 = 0.55$ and having these trees as subtrees:



The resulting B.T is then combined with the one-node tree

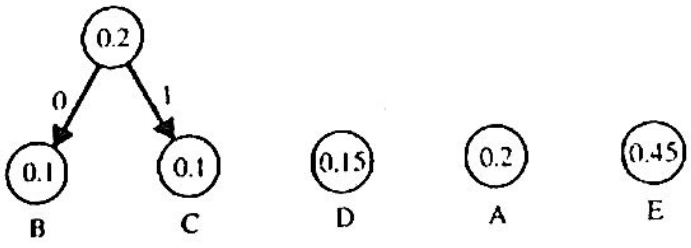


The Huffman codes obtained from this tree are as follows:

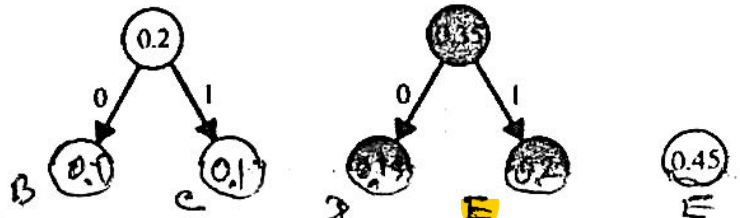
Character	Huffman Code
A	01
B	0000
C	0001
D	001
E	1

As we calculated earlier, the expected length of the code for each of these characters is 2.1.

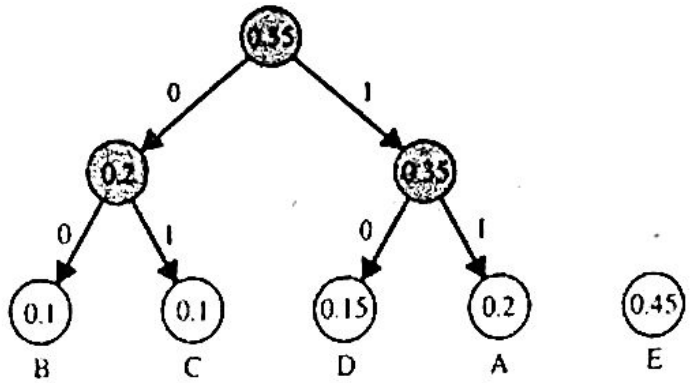
A different assignment of codes to these characters for which the expected length is also 2.1 is possible because at the second stage we had two choices for trees of minimal weight:



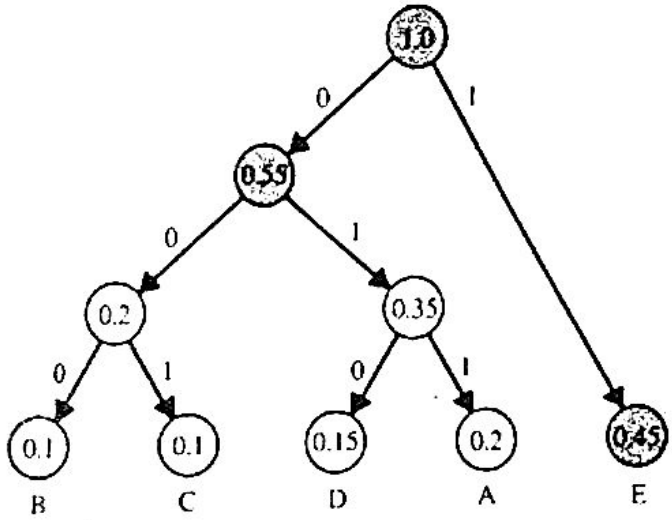
We selected the first and second trees from this list, but we could have used the second and third:



At the next stage, the resulting list of two binary trees would have been



and the final Huffman tree



The assignment of codes corresponding to this tree is

Character	Huffman Code
A	011
B	000
C	001
D	010
E	1

The immediate decodability property of Huffman codes is clear. Each character is associated with a leaf node in the Huffman tree, and there is a unique path from the root of a tree to each leaf. Consequently, no sequence of bits comprising the code for some character can be a prefix of a longer sequence of bits for some other character.

Because of this property of immediate decodability, a decoding algorithm is easy:

HUFFMAN DECODING ALGORITHM

(* Algorithm to decode messages that were encoded using a Huffman

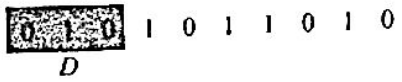
1. Initialize pointer p to the root of the Huffman tree.
2. While the end of the message string has not been reached, do the following:

   a. Let x be the next bit in the string.
   b. If $x = 0$ then
      Set p equal to its left child pointer.
      Else
      Set p equal to its right child pointer.
   c. If p points to a leaf then:
      i. Display the character associated with that leaf.
      ii. Reset p to the root of the Huffman tree.

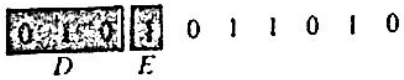
As an illustration, suppose the message string

0 1 0 1 0 1 1 0 1 0

is received, and that this message was encoded using the second Huffman tree constructed earlier. The pointer follows the path labeled 010 from the root of this tree to the letter D and is then reset to the root:



The next bit, 1, leads immediately to the letter E:



The pointer p next follows the path 011 to the letter A,

and finally the path 010 to the letter D again:

The program in Figure 12.1 implements this decoding algorithm. The procedure *BuildDecodingTree* initializes a tree consisting of a single node and then reads letters and their codes from *CodeFile* and constructs the decoding tree. For each letter in the file, it calls procedure *AddToTree* to follow a path determined by the code of the character, creating nodes as necessary. When the end of the code string is reached, the character in inserted in the last (leaf) node created on this path. The procedure *Decode* is then called to read a message string of bits from *MessageFile* and to decode it using the decoding tree. Procedure *PrintTree* is included in this program simply to give an idea of what the tree looks like. It is basically nothing more than an *RNL* traversal of