9.4 Huffman Trees

Suppose we have to encode a text that comprises n characters from some alphabet by assigning to each of the text's characters some sequence of bits called the ***codeword***. For example, we can use a ***fixed-length encoding*** that assigns to each character a bit string of the same length m ($m \geq \log_2 n$). This is exactly what the standard seven-bit ASCII code does. One way of getting a coding scheme that yields a shorter bit string on the average is based on the old idea of assigning shorter codewords to more frequent characters and longer codewords to less frequent characters. (This idea was used, in particular, in the telegraph code invented in the mid-19th century by Samuel Morse. In that code, frequent letters such as e ($\cdot$) and a ($\cdot-$) are assigned short sequences of dots and dashes while infrequent letters such as q ($--\cdot-$) and z ($--\cdot\cdot$) have longer ones.)

Using a ***variable-length encoding***, which assigns codewords of different lengths to different characters, introduces a problem that fixed-length encoding does not have. Namely, how can we tell how many bits of an encoded text represent the first (or, more generally, the ith) character? To avoid this complication, we can limit ourselves to the so-called ***prefix-free*** (or simply ***prefix***) ***codes***. In a prefix code, no codeword is a prefix of a codeword of another character. Hence, with such an encoding, we can simply scan a bit string until we get the first group

of bits that is a codeword for some character, replace these bits by this character, and repeat this operation until the bit string's end is reached.

If we want to create a binary prefix code for some alphabet, it is natural to associate the alphabet's characters with leaves of a binary tree in which all the left edges are labeled by 0 and all the right edges are labeled by 1 (or vice versa). The codeword of a character can then be obtained by recording the labels on the simple path from the root to the character's leaf. Since there is no simple path to a leaf that continues to another leaf, no codeword can be a prefix of another codeword; that is, any such tree yields a prefix encoding.

Among the many trees that can be constructed in this manner for a given alphabet with known frequencies of the character occurrences, how can we construct a tree that would assign shorter bit strings to high-frequency characters and longer ones to low-frequency characters? It can be done by the following greedy algorithm, invented by David Huffman as part of a class assignment in his student days at MIT [Huf52].

Huffman's Algorithm

Step 1 Initialize n one-node trees and label them with the characters of the alphabet. Record the frequency of each character in its tree's root to indicate the tree's ***weight***. (More generally, the weight of a tree will be equal to the sum of the frequencies in the tree's leaves.)

Step 2 Repeat the following operation until a single tree is obtained. Find two trees with the smallest weight (ties can be broken arbitrarily, but see Problem 2 in the exercises). Make them the left and right subtree of a new tree and record the sum of their weights in the root of the new tree as its weight.

A tree constructed by the above algorithm is called a ***Huffman tree***. It defines—in the manner described—a ***Huffman code***.

EXAMPLE Consider the five-character alphabet {A, B, C, D, _} with the following occurrence probabilities:

character	A	B	C	D	_
probability	0.35	0.1	0.2	0.2	0.15

The Huffman tree construction for this input is shown in Figure 9.11. The resulting codewords are as follows:

character	A	B	C	D	_
probability	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101

Hence, DAD is encoded as 011101, and 10011011011101 is decoded as BAD_AD.

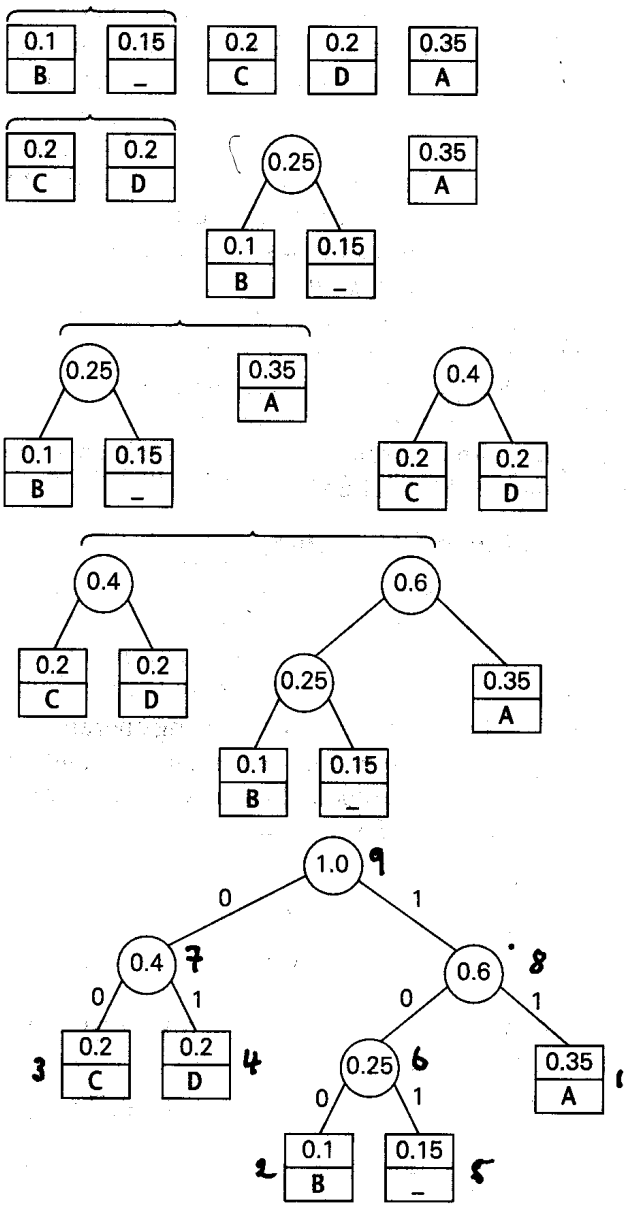

FIGURE 9.11 Example of constructing a Huffman coding tree

With the occurrence probabilities given and the codeword lengths obtained the expected number of bits per character in this code is

$$2 \cdot 0.35 + 3 \cdot 0.1 + 2 \cdot 0.2 + 2 \cdot 0.2 + 3 \cdot 0.15 = 2.25.$$

Had we used a fixed-length encoding for the same alphabet, we would have to use at least three bits per each character. Thus, for this toy example, Huffman's code achieves the ***compression ratio***, a standard measure of a compression algorithm's effectiveness, of $(3 - 2.25)/3 \cdot 100\% = 25\%$. In other words, we should expect that Huffman's encoding of a text will use 25% less memory than its fixed-length encoding. (Extensive experiments with Huffman codes have shown that the compression ratio for this scheme typically falls between 20% and 80%, depending on the characteristics of the file being compressed.) ■

Huffman's encoding is one of the most important file compression methods. In addition to its simplicity and versatility, it yields an optimal, i.e., minimal-length encoding (provided the probabilities of character occurrences are independent and known in advance). The simplest version of Huffman compression calls, in fact, for a preliminary scanning of a given text to count the frequencies of character occurrences in it. Then these frequencies are used to construct a Huffman coding tree and encode the text as described. This scheme makes it necessary, however, to include the information about the coding tree into the encoded text to make its decoding possible. This drawback can be overcome by the so-called ***dynamic Huffman encoding***, in which the coding tree is updated each time a new character is read from the source text (see, e.g., [Say00]).

It is important to note that applications of Huffman's algorithm are not limited to data compression. Suppose we have n positive numbers $w_1, w_2, \ldots, w_n$, which have to be assigned to n leaves of a binary tree, one per node. If we define the ***weighted path length*** as the sum $\sum_{i=1}^{n} l_i w_i$, where l_i is the length of the simple path from the root to the ith leaf, how can we construct a binary tree with minimum weighted path length? It is this more general problem that Huffman's algorithm actually solves. (For the coding application considered, l_i and w_i are the length of the codeword and the frequency of the ith character, respectively.) This problem arises in many situations involving decision making. Consider, for example, the game of guessing a chosen object from n possibilities (say, an integer between 1 and n) by asking questions answerable by yes or no. Different strategies for playing this game can be modeled by ***decision trees***[4] such as those depicted in Figure 9.12 for $n = 4$.

The length of the simple path from the root to a leaf in such a tree is equal to the number of questions needed to get to the chosen number represented by the leaf. If number i is chosen with probability p_i, the sum $\sum_{i=1}^{n} l_i p_i$ (where l_i is the length of the path from the root to the ith leaf) indicates the average number of questions needed to "guess" the chosen number with a game strategy represented by its decision tree. If each of the numbers is chosen with the same probability of $1/n$, the best strategy is to successively eliminate half (or almost half) the candidates as binary search does. This may not be the case for arbitrary

4. Decision trees are discussed in more detail in Section 10.2.

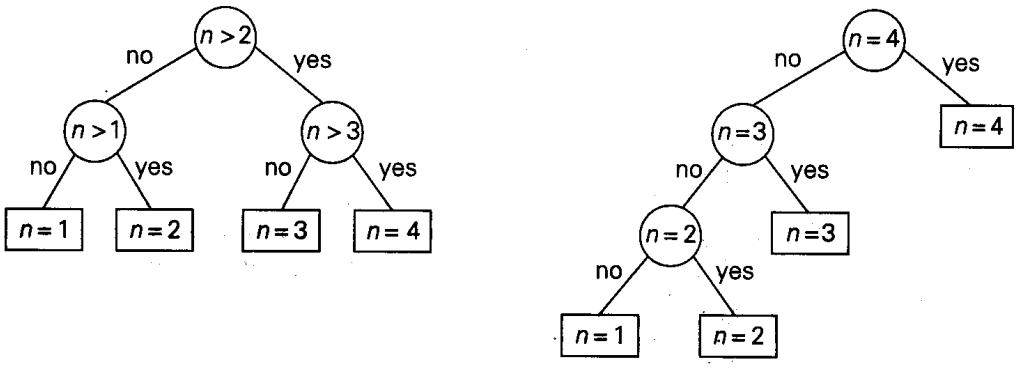

FIGURE 9.12 Two decision trees for guessing an integer between 1 and 4

p_i's, however. (For example, if $n = 4$ and $p_1 = 0.1$, $p_2 = 0.2$, $p_3 = 0.3$, and $p_4 = 0.4$, the minimum weighted path tree is the rightmost one in Figure 9.12.) Thus, we need Huffman's algorithm to solve this problem in its general case.

In conclusion, it is worthwhile to remember that it is the second time we are encountering the problem of constructing an optimal binary tree. In Section 8.3, we discussed the problem of constructing an optimal binary search tree with positive numbers (the search probabilities) assigned to every node of the tree. In this section, given numbers are assigned just to leaves. The latter problem turns out to be easier: it can be solved by the greedy algorithm whereas the former is solved by the more complicated dynamic programming algorithm.

Exercises 9.4

1. **a.** Construct a Huffman code for the following data:

character	A	B	C	D	_
probability	0.4	0.1	0.2	0.15	0.15

   b. Encode the text ABACABAD using the code of question a.

   c. Decode the text whose encoding is 100010111001010 in the code of question a.

2. For data transmission purposes, it is often desirable to have a code with a minimum variance of the codeword lengths (among codes of the same average length). Compute the average and variance of the codeword length in two Huffman codes that result from a different tie breaking during a Huffman code construction for the following data:

character	A	B	C	D	E
probability	0.1	0.1	0.2	0.2	0.4

3. Indicate whether each of the following properties are true for every Huffman code.
   a. The codewords of the two least frequent characters have the same length.
   b. The codeword's length of a more frequent character is always smaller than or equal to the codeword's length of a less frequent one.
4. What is the maximal length of a codeword possible in a Huffman encoding of an alphabet of n characters?
5. a. Write a pseudocode for the Huffman tree construction algorithm.
   b. What is the time efficiency class of the algorithm for constructing a Huffman tree as a function of the alphabet's size?
6. Show that a Huffman tree can be constructed in linear time if the alphabet's characters are given in a sorted order of their frequencies.
7. Given a Huffman coding tree, which algorithm would you use to get the codewords for all the characters? What is its time-efficiency class as a function of the alphabet's size?
8. Explain how one can generate a Huffman code without an explicit generation of a Huffman coding tree.
9. a. Write a program that constructs a Huffman code for a given English text and encode it.
   b. Write a program for decoding an English text that has been encoded with a Huffman code.
   c. Experiment with your encoding program to find a range of typical compression ratios for Huffman's encoding of English texts of, say, 1000 words.
   d. Experiment with your encoding program to find out how sensitive the compression ratios are to using standard estimates of frequencies instead of actual frequencies of character occurrences in English texts.
10. Design a strategy that minimizes the expected number of questions asked in the following game [Gar94], #52. You have a deck of cards that consists of one ace of spades, two deuces of spades, three threes, and on up to nine nines, making 45 cards in all. Someone draws a card from the shuffled deck, which you have to identify by asking questions answerable with yes or no.

SUMMARY

- The *greedy technique* suggests constructing a solution to an optimization problem through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. On each step, the choice made must be *feasible*, *locally optimal*, and *irrevocable*.

```
#include <iostream.h>
#include <stdlib.h>
#include "minHeap2.cpp"  // modified version of minHeap.cpp


class HuffmanTree {

  private:
      int * dad;         // store the Huffman tree
      int * count;       // stores freq or prob of each char
      int * code;        // store codeword for each char
      int * len;         // stores number of bits in each codeword
      int m;             // m = alphabet size
      int n;          // n >= total number of nodes in Huffman tree

  public:
    HuffmanTree( int count_[], int asize, int size)
    {
        count = count_;
        m = asize;
        n = size;

        // h is a priority queue initially empty
        MinHeap h(count);
        dad = new int[n+1];

        int i, t1, t2;

        //insert alphabet into heap
        // heap will use count[] for prioritising
        for(i = 1; i <= m; ++i)
            if( count[i] != 0) h.insert( i);

        while( h.size() > 1) {
            t1 = h.remove();
            t2 = h.remove();
            count[i] = count[t1] + count[t2];
            dad[t1] = i;  dad[t2] = -i;
            h.insert( i); ++i;
        }
        dad[--i] = 0;
    }
```

```
    // Store codewords as a decimal number in an array code[]
    // When number converted to binary, you get the codeword
    // Also store the number of bits for each codeword in len[]
    void buildCodewords()
    {
        int x, j, i, t;
        code = new int[m+1];
        len = new int[m+1];

        for(int k=1; k<=m; ++k)
        {
            x = 0; j = 1; i = 0;
            t = dad[k];
            while( t != 0) {
                if(t < 0) { t = -t; x = x + j;}
                j = 2 * j;  ++i;
                t = dad[t];
            }
            len[k] = i;
            code[k] = x;
        }
    }



    void displayCodewords()
    {
        cout << "\nCode words for alphabet are:\n";
        for(int k=1; k<=m; ++k) {
            cout << "code[" << char (k+64) << "] is "
                 << codewordToStr(k) << "\n";
        }
        cout << endl;
    }


    char * codewordToStr( int k)
    {
        int w, l;
        char * str;

        w = code[k]; l = len[k]; str = new char[l+1];
        for(int i = l-1; i >= 0; --i) {
            str[i]  =  w % 2 == 0? '0' : '1';
            w = w / 2;
        }
        str[l] = '\0';
        return str;
    }
}; // end of class
```

```
void main()
{
    int count[20];
    for(int i=0; i< 20; ++i) count[i] = 0;

    count[1] = 35; count[2] = 10; count[3] = 20;
    count[4] = 20; count[5] = 15;

    HuffmanTree ht( count, 5, 15);

    ht.buildCodewords();

    ht.displayCodewords();
}
```