Chapter 9

Trees

Any data-type which exhibits a non-linear (or 'branching') structure is generically called a *tree*. Trees serve as natural representations for any form of hierarchically organised data. One example was given in the last chapter, where the syntactic structure of arithmetic expressions was modelled as a tree. As we shall see, trees are also useful for the efficient implementation of functions concerned with search and retrieval.

There are numerous species and subspecies of tree. They can be classified according to the precise form of the branching structure, the location of information within the tree, and the nature of the relationships between the information in different parts of the tree. In the present chapter we shall study two or three of the most common species, describe a little of the basic terminology associated with trees, and outline some of the more important applications.

9.1 Binary trees

As its name implies, a binary tree is a tree with a simple two-way branching structure. This structure is captured by the following type declaration:

$$\mathit{btree}\ \alpha ::= \mathit{Tip}\ \alpha \mid \mathit{Bin}\ (\mathit{btree}\ \alpha)\ (\mathit{btree}\ \alpha)$$

A value of $(\mathit{btree}\ \alpha)$ is therefore either a 'tip' (indicated by the constructor *Tip*), which contains a value of type α, or a binary 'node' (indicated by the constructor *Bin*), which consists of two further trees called the *left* and *right* subtrees of the node. For example, the tree:

```
Bin (Tip 1)
    (Bin (Tip 2) (Tip 3))
```

consists of a binary node with a left subtree $(\mathit{Tip}\ 1)$, and a right subtree $(\mathit{Bin}\ (\mathit{Tip}\ 2)\ (\mathit{Tip}\ 3))$ which, in turn, has a left subtree $(\mathit{Tip}\ 2)$ and a right

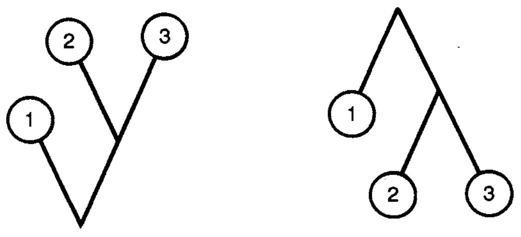

Figure 9.1 Upward and downward trees.

subtree $(Tip\ 3)$. Compare this element of $(btree\ num)$ with the tree:

$$\begin{array}{l} Bin\ (Bin\ (Tip\ 1)\ (Tip\ 2)) \\ \quad (Tip\ 3) \end{array}$$

Although the second tree contains the same sequence of numbers in its tips as the first, the way the information is organised is quite different and the two expressions denote distinct values.

A tree can be pictured in one of two basic ways, growing upwards or growing downwards. Both orientations are illustrated in Figure 9.1. The downward pointing orientation is the one normally preferred in computing, and this is reflected in some of the basic terminology of trees. For instance, we talk about the 'depth' of a tree rather than its 'height'. The depth of a tree is defined below.

9.1.1 Measures on trees

There are two important measures on binary trees, size and depth. The *size* of a tree is the number of tips it contains. Hence:

$$\begin{array}{lcl} size\ (Tip\ x) & = & 1 \\ size\ (Bin\ t1\ t2) & = & size\ t1 + size\ t2 \end{array}$$

The function *size* plays the same role for trees as $(\#)$ does for lists. In particular, a tree is finite if and only if it has a well-defined size.

There is a simple, but important, relationship between the number of tips and the number of nodes in a finite tree: the former is always one more than the latter. If we count the number of nodes by the function *nsize*, where:

$$\begin{array}{lcl} nsize\ (Tip\ x) & = & 0 \\ nsize\ (Bin\ t1\ t2) & = & 1 + nsize\ t1 + nsize\ t2 \end{array}$$

then we have:

$$size\ t = 1 + nsize\ t$$

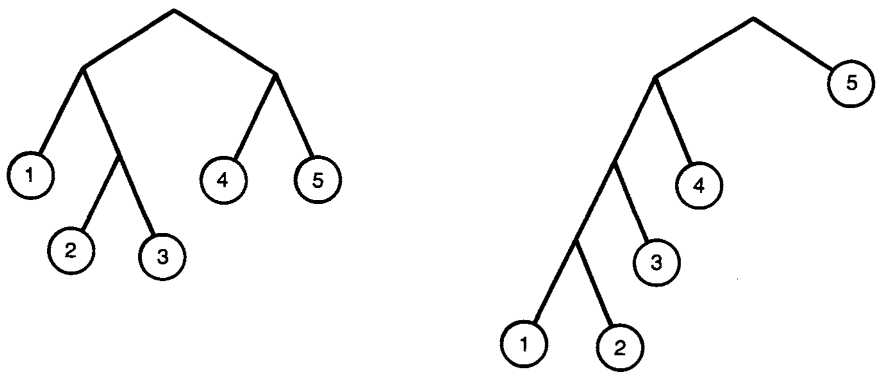

Figure 9.2 Trees of depth 3 and 4.

for all binary trees t. The result can be proved by structural induction and is left as an exercise. It also holds, by default, in the case of infinite trees, since both sides of the above equation reduce to $\perp$.

The second useful measure on trees is the notion of depth. The depth of a finite tree is defined as follows:

$$\begin{array}{lcl} \mathit{depth}\,(\mathit{Tip}\ x) & = & 0 \\ \mathit{depth}\,(\mathit{Bin}\ t1\ t2) & = & 1 + (\mathit{depth}\ t1)\ \mathbf{max}\ (\mathit{depth}\ t2) \end{array}$$

In words, the depth of a tree consisting of a single tip is 0; otherwise it is one more than the greater of the heights of its two subtrees. For example, the tree on the left in Figure 9.2 has depth 3, while the one on the right has depth 4. Notice these trees have the same size and, indeed, exactly the same sequence of tip values. The notion of depth is important because it is a measure of the time required to retrieve a tip value from a tree.

Although two trees of the same size need not have the same depth, the two measures are not independent. The following result is one of the most important facts about binary trees. It says that:

$$\mathit{size}\ t \leq 2\,\hat{}\ \mathit{depth}\ t$$

for all (finite) trees t. We shall prove this inequality by structural induction on t.

Case $(\mathit{Tip}\ x)$. We have:

$$\begin{array}{lcll} \mathit{size}\,(\mathit{Tip}\ x) & = & 1 & (\mathit{size}.1) \\ & = & 2\,\hat{}\,0 & (\hat{}.1) \\ & = & 2\,\hat{}\ \mathit{depth}\,(\mathit{Tip}\ x) & (\mathit{depth}.1) \end{array}$$

as required.

Case $(Bin\ t1\ t2)$. Assume, by way of induction, that:

$$\begin{aligned} size\ t1 &\leq 2\,\hat{}\ depth\ t1 \\ size\ t2 &\leq 2\,\hat{}\ depth\ t2 \end{aligned}$$

and let:

$$d = (depth\ t1)\ \mathbf{max}\ (depth\ t2)$$

We now have:

$$\begin{aligned} size\,(Bin\ t1\ t2) &= size\ t1 + size\ t2 && (size.2) \\ &\leq 2\,\hat{}\,(depth\ t1) + 2\,\hat{}\,(depth\ t2) && \text{(hypothesis)} \\ &\leq 2\,\hat{}\ d + 2\,\hat{}\ d && \text{(monotonicity of } \hat{}\,) \\ &= 2\,\hat{}\,(1+d) && \text{(arithmetic)} \\ &= 2\,\hat{}\,(depth\,(Bin\ t1\ t2)) && (depth.2) \end{aligned}$$

as required. □

By taking logarithms (to base 2), we can restate the result in the following equivalent form:

$$depth\ t \geq \log_2 (size\ t)$$

for all finite trees t.

Given any positive integer n, it is always possible to construct a tree of size n with depth d satisfying:

$$d = \lceil \log_2\ n \rceil$$

where $\lceil x \rceil$ denotes the smallest integer $k \geq x$. Such trees are said to be *minimal*. In general, there will be more than one minimal tree of a given size. Minimal trees are useful because, by making a tree minimal, we can ensure that the cost of retrieving tip values is as small as possible.

9.1.2 Map and fold over trees

The generic functions *map* and *fold* for lists have analogues *mapbtree* and *foldbtree* for binary trees. They are defined as follows:

$$\begin{aligned} &mapbtree && :: && (\alpha \to \beta) \to btree\ \alpha \to btree\ \beta \\ &mapbtree\ f\ (Tip\ x) && = && Tip\ (f\ x) \\ &mapbtree\ f\ (Bin\ t1\ t2) && = && Bin\ (mapbtree\ f\ t1)\ (mapbtree\ f\ t2) \end{aligned}$$

$$\begin{aligned} &foldbtree && :: && (\alpha \to \alpha \to \alpha) \to btree\ \alpha \to \alpha \\ &foldbtree\ (\oplus)\ (Tip\ x) && = && x \\ &foldbtree\ (\oplus)\ (Bin\ t1\ t2) && = && (foldbtree\ (\oplus)\ t1) \oplus (foldbtree\ (\oplus)\ t2) \end{aligned}$$

Many operations on trees can be defined in terms of these functions. For example, the sum of the tips in a tree of numbers can be defined by:

$$sumtips = foldbtree\ (+)$$

The size of a tree is defined by:

$$size = foldbtree\ (+) \cdot mapbtree\ (const\ 1)$$

and the depth of a tree is defined by:

$$\begin{aligned} depth &= foldbtree\ (\oplus) \cdot mapbtree\ (const\ 0) \\ &\quad \textbf{where}\ d1 \oplus d2 = 1 + (d1\ \mathbf{max}\ d2) \end{aligned}$$

Finally, the function *tips* for listing the tip values of a tree in left to right order can be defined by:

$$\begin{aligned} tips &= foldbtree\ (+\!\!+) \cdot mapbtree\ unit \\ &\quad \textbf{where}\ unit\ x = [x] \end{aligned}$$

The function *mapbtree* satisfies laws similar to the function *map*. In particular, we have:

$$mapbtree\ f \cdot mapbtree\ g = mapbtree\ (f \cdot g)$$

for any functions f and g. We also have the identity:

$$map\ f \cdot tips = tips \cdot mapbtree\ f$$

which relates *map*, *tips* and *mapbtree*.

There is also a law relating *foldbtree* and *foldr1* (or, equally, *foldl1*). It says that if $\oplus$ is associative, then:

$$\begin{aligned} foldbtree\ (\oplus) &= foldr1\ (\oplus) \cdot tips \\ &= foldl1\ (\oplus) \cdot tips \end{aligned}$$

(Note that *tips* always returns a non-empty list.) All of the above identities can be proved by structural induction.

Like *map* and *fold* with lists, we can use *foldbtree* and *mapbtree* to define many functions over trees without using recursion explicitly. Since the resulting definitions are shorter, this is certainly a good idea in any application where a number of tree processing functions are required. On the other hand, a direct recursive definition is just as good in simple situations. Unlike lists, there is a natural recursive decomposition of trees in terms of their subtrees, so a definition which exhibits the same kind of recursive decomposition is often simplest.

9.1.3 Labelled binary trees

Finally, we introduce a slight variation on the basic structure of binary trees. By definition, a *labelled* binary tree is a value of the following type:

$$lbtree\ \alpha\ \beta ::= Tip\ \alpha \mid Bin\ \beta\ (lbtree\ \alpha\ \beta)\ (lbtree\ \alpha\ \beta)$$

Here, binary nodes are 'labelled' with values of a second type β. Apart from these additional values, a labelled binary tree has exactly the same structure as the earlier kind. Because extra information is available, many operations on binary trees can be implemented efficiently in terms of labelled binary trees. We shall see examples of this idea in subsequent sections.

Exercises

9.1.1 Prove that the number of tips in a binary tree is always one more than the number of internal nodes.

9.1.2 The subtrees of a binary tree t can be defined by:

$$\begin{aligned} \mathit{subtrees}\,(\mathit{Tip}\ x) &= [\mathit{Tip}\ x] \\ \mathit{subtrees}\,(\mathit{Bin}\ t1\ t2) &= \mathit{subtrees}\ t1 \mathbin{+\!\!+} \mathit{subtrees}\ t2 \mathbin{+\!\!+} [\mathit{Bin}\ t1\ t2] \end{aligned}$$

State and prove a relationship between $\#(\mathit{subtrees}\ t)$ and $\mathit{size}\ t$.

9.1.3 Show that:

$$\mathit{depth}\ t \leq \mathit{size}\ t - 1$$

for all finite binary trees.

9.1.4 Prove that if xs is a list of 2^n values, then there is a unique minimal tree t such that $\mathit{tips}\ t = xs$.

9.1.5 Prove that a minimal tree of size n has depth $\lceil \log_2 n \rceil$.

9.1.6 Design a function that takes a non-empty list xs into a minimal tree t such that $\mathit{tips}\ t = xs$.

9.1.7 Prove the laws:

$$\begin{aligned} \mathit{mapbtree}\ f \cdot \mathit{mapbtree}\ g &= \mathit{mapbtree}\ (f \cdot g) \\ \mathit{map}\ f \cdot \mathit{tips} &= \mathit{tips} \cdot \mathit{mapbtree}\ f \\ \mathit{foldbtree}\ (\oplus) &= \mathit{foldr1}\ (\oplus) \cdot \mathit{tips} \end{aligned}$$

where, in the last law, $\oplus$ is associative.

9.1.8 Suppose $f = \mathit{foldbtree}(\oplus)$, where $\oplus$ is associative with identity element e. Prove that $f\ t = \mathit{fm}\ t\ e$, where:

$$\mathit{fm} = \mathit{foldbtree}\ (\cdot) \cdot \mathit{mapbtree}\ (\oplus)$$

Using this result and the fact that:

$$\begin{aligned} \mathit{tips} &= \mathit{foldbtree}\ (\mathbin{+\!\!+}) \cdot \mathit{mapbtree}\ \mathit{unit} \\ &\quad \textbf{where}\ \ \mathit{unit}\ x = [x] \end{aligned}$$

derive the equation:

$$tips\ t = mtips\ t\ [\,]$$

where

$$mtips = foldbtree\ (\cdot) \cdot mapbtree\ (:)$$

Compare the costs of computing $(tips\ t)$ by the two definitions. (*Hint:* Recall that the cost of computing $xs + \!\!+\ ys$ is proportional to $\#xs$.)

9.2 Huffman coding trees

As a first example of the use of binary trees, we shall consider the problem of coding data efficiently. As many computer users know only too well, it is often necessary to store files of information as compactly as possible in order to free precious space for other, more urgent, purposes. Suppose the information to be stored is a text consisting of a sequence of characters. The ASCII standard code uses seven bits to represent each of $2^7 = 128$ possible different characters, so a text of n characters contains $7n$ bits of information. For example, the letters 't', 'e', and 'x' are represented in ASCII by the codes:

$$\begin{array}{lcl} t & \longrightarrow & 1110100 \\ e & \longrightarrow & 1100101 \\ x & \longrightarrow & 1111000 \end{array}$$

In particular, the text "text" is coded in ASCII as the sequence:

1110100110010111110001110100

of 28 bits. As ASCII is a fixed-length code, the original text can be recovered by decoding each successive group of seven bits.

One idea for reducing the total number of bits required to code a text is to abandon the notion of fixed-length codes, and seek instead a coding scheme based on the relative frequency of occurrence of the characters in the text. The basic idea is to take a sample piece of text, estimate the number of times each character appears, and choose short codes for the more frequent characters and longer codes for the rarer ones. For example, if we take the codes:

$$\begin{array}{lcl} t & \longrightarrow & 0 \\ e & \longrightarrow & 10 \\ x & \longrightarrow & 11 \end{array}$$

then "text" can be coded as the bit sequence 010110 of length 6.

It is important to realise that codes must be chosen in such a way as to ensure that the coded text can be deciphered uniquely. To illustrate, suppose the codes had been:

$$\begin{array}{lcl} t & \longrightarrow & 0 \\ e & \longrightarrow & 10 \\ x & \longrightarrow & 1 \end{array}$$

Under this scheme, "text" would be coded as the sequence 01010 of length 5. However, the string "tee" would be coded by exactly the same sequence, and this is obviously not what is wanted. The simplest way to prevent this happening is to choose codes so that no code is a proper initial segment (or *prefix*) of any other.

As well as requiring unique decipherability, we also want the coding scheme to be optimal. An optimal coding scheme is one which minimises the expected length of the coded text. More precisely, if characters c_j, for $1 \leq j \leq n$, have probabilities of occurrence p_j, then we want to choose codes with lengths l_j such that

$$\sum_{j=1}^{n} p_j l_j$$

is as small as possible.

One method for constructing an optimal code satisfying the prefix property is called Huffman coding (after its inventor, David Huffman). Each character is stored as a tip of a binary tree, the structure of which is determined by the computed frequencies. The code for a character c is a sequence of binary values describing the path in the tree to the tip containing c. Such a scheme guarantees that no code is a prefix of any other. We can define a path formally by:

$$\begin{array}{lcl} \textit{step} & ::= & \textit{Left} \mid \textit{Right} \\ \textit{path} & == & [\textit{step}] \end{array}$$

A path is therefore a sequence of steps, each of which is one of the two values *Left* or *Right*. A path can be traced by the function *trace*, defined by:

$$\begin{array}{lcl} \textit{trace}\,(\textit{Tip}\;x)\,[\,] & = & x \\ \textit{trace}\,(\textit{Bin}\;t1\;t2)\,(\textit{Left} : ps) & = & \textit{trace}\;t1\;ps \\ \textit{trace}\,(\textit{Bin}\;t1\;t2)\,(\textit{Right} : ps) & = & \textit{trace}\;t2\;ps \end{array}$$

If *ps* is a path in *t* leading to a tip, then (*trace t ps*) is the value associated with the tip; otherwise $\textit{trace}\;t\;ps = \bot$.

To illustrate the idea, consider the tree:

$$\begin{array}{l} \textit{Bin}\,(\textit{Bin}\,(\textit{Tip}\;\text{'x'})\,(\textit{Tip}\;\text{'e'})) \\ \quad\;(\textit{Tip}\;\text{'t'}) \end{array}$$

In this tree the character 'x' is coded by the path [*Left*, *Left*], character 'e' by [*Left*, *Right*], and character 't' by [*Right*]. So 't' is coded by one bit of information, while the others require two bits. For example, the string "text" is encoded by the sequence:

[*Right*, *Left*, *Right*, *Left*, *Left*, *Right*]

which is the same as 010110 when *Right* is replaced by 0 and *Left* by 1.

There are three aspects to the problem of implementing Huffman coding: (i) building a binary tree; (ii) coding a sequence of characters; and (iii) decoding a coded sequence. We shall deal with these stages in reverse order.

Decoding. Suppose *ps* is a sequence of steps representing a sequence *xs* of characters with respect to a given tree *t*. The function *decodexs*, where *decodexs t ps* = *xs*, can be defined in the following way:

$$\begin{array}{lcl}
\mathit{decodexs}\ t\ ps & = & \mathit{traces}\ t\ t\ ps \\
\mathit{traces}\ t\ (\mathit{Tip}\ x)\ [\,] & = & [x] \\
\mathit{traces}\ t\ (\mathit{Tip}\ x)\ (p : ps) & = & [x] +\!\!\!+\ \mathit{traces}\ t\ t\ (p : ps) \\
\mathit{traces}\ t\ (\mathit{Bin}\ t1\ t2)\ (\mathit{Left} : ps) & = & \mathit{traces}\ t\ t1\ ps \\
\mathit{traces}\ t\ (\mathit{Bin}\ t1\ t2)\ (\mathit{Right} : ps) & = & \mathit{traces}\ t\ t2\ ps
\end{array}$$

The first argument of *traces* is the given tree, while the second argument is the subtree currently being traversed. Each time a tip is reached, the associated character is produced and *t* is regenerated in order to process the remaining paths, if any. The time for decoding is clearly linear in the length of *ps*. Notice that if *ps* does not correspond to a legal sequence of paths, then (*decodexs t ps*) will be a partial list.

Coding. Next, let us deal with step (ii), the coding phase. Here, the input is a sequence of characters and the output is a sequence of steps. We can define:

$$\mathit{codexs}\ t = \mathit{concat} \cdot \mathit{map}\ (\mathit{codex}\ t)$$

so the problem reduces to how to code a single character. The following definition of *codex* is straightforward, but leads to an inefficient algorithm:

$$\begin{array}{lcll}
\mathit{codex}\ (\mathit{Tip}\ y)\ x & = & [\,], & \textbf{if}\ x = y \\
\mathit{codex}\ (\mathit{Bin}\ t1\ t2)\ x & = & \mathit{Left} : \mathit{codex}\ t1\ x, & \textbf{if}\ \mathit{member}\ t1\ x \\
 & = & \mathit{Right} : \mathit{codex}\ t2\ x, & \textbf{if}\ \mathit{member}\ t2\ x
\end{array}$$

The formal definition of *member* is:

$$\begin{array}{lcl}
\mathit{member}\ (\mathit{Tip}\ y)\ x & = & (x = y) \\
\mathit{member}\ (\mathit{Bin}\ t1\ t2)\ x & = & \mathit{member}\ t1\ x \vee \mathit{member}\ t2\ x
\end{array}$$

Note that if x is not a tip value in t, then *codex* $t\ x = \bot$.

The trouble with the definition of *codex* lies in the many costly calculations of *member*. Since the time to calculate (*member* $t\ x$) is $O(n)$ steps, where n = *size* t, the time $T(n)$ required to compute (*codex* $t\ x$) in the worst possible case satisfies:

$$T(n) = T(n-1) + O(n)$$

and so $T(n) = O(n^2)$. The worst possible case arises, for instance, when every left subtree has size 1, and x appears as the rightmost tip value. It also arises when every right subtree has size 1, and x appears as the leftmost tip value.

One way to improve this unacceptable quadratic behaviour is to define:

$$\begin{array}{lcll}
\mathit{codex}\ t\ x & = & \mathit{hd}\,(\mathit{codesx}\ t\ x) & \\
\mathit{codesx}\,(\mathit{Tip}\ y)\ x & = & [[\,]], & \textbf{if } x = y \\
 & = & [\,], & \textbf{otherwise} \\
\mathit{codesx}\,(\mathit{Bin}\ t1\ t2)\ x & = & \mathit{map}\,(\mathit{Left}\,:)\,(\mathit{codesx}\ t1\ x) +\!\!\!+ & \\
 & & \mathit{map}\,(\mathit{Right}\,:)\,(\mathit{codesx}\ t2\ x) &
\end{array}$$

This is an example of the list of successes technique described in Chapter 6. The value $(\mathit{codesx}\ t\ x)$ is a list of *all* paths in t which lead to a tip containing x. If t is a tip, then the list is either empty (if the characters do not match), or is a singleton list containing the empty path. If t is not a tip, then the final list is the concatenation of the list of paths through the left subtree with the list of paths through the right subtree. If t contains exactly one occurrence of x, then $(\mathit{codesx}\ t\ x)$ will be a singleton list containing the desired path. We shall leave as an exercise the proof that this version of *codex* requires only $O(n)$ steps.

Constructing a Huffman tree. Now we must deal with the most interesting part, building a coding tree. Let us suppose that the relative frequencies of the characters have been computed from the sample, so we are given a list of pairs:

$$[(c_1, w_1), (c_2, w_2), \ldots, (c_n, w_n)]$$

where $c_1, c_2, \ldots, c_n$ are the characters and $w_1, w_2, \ldots, w_n$ are numbers, called *weights*, indicating the frequencies. The probability of character c_j occurring is therefore w_j/W, where $W = \sum w_j$. Without loss of generality, we shall also suppose that the weights satisfy $w_1 \leq w_2 \leq \cdots \leq w_n$, so the characters are listed in increasing order of likelihood.

The procedure for building a Huffman tree is as follows. The first step is to convert the list of character-weight pairs into a list of trees by applying $(\mathit{map}\ \mathit{Tip})$. Each tip will contain a pair (x, w), where x is a character and w its associated weight. This list of trees is then reduced to a single tree by repeatedly applying a function which combines two trees into one, until just a single tree is left. Thus:

$$\mathit{build} \;=\; \mathit{until}\ \mathit{single}\,(\mathit{combine} \cdot \mathit{map}\ \mathit{Tip})$$

Here, $(\mathit{until}\ p\ f)$ is the function that repeatedly applies f until p becomes true, and *single* is the test for a singleton list.

The effect of *combine* on a list *ts* of trees is to combine two trees in *ts* with the lightest weights, where:

$$\begin{array}{lcl}
\mathit{weight}\,(\mathit{Tip}\,(x, w)) & = & w \\
\mathit{weight}\,(\mathit{Bin}\ t1\ t2) & = & \mathit{weight}\ t1 + \mathit{weight}\ t2
\end{array}$$

Thus, the weight of a tree is the sum of the weights in its tip nodes.

In order to determine at each stage which are the lightest trees, we simply keep the trees in increasing order of weight. Thus:

$$\mathit{combine}\,(t1 : t2 : ts) \;=\; \mathit{insert}\,(\mathit{Bin}\ t1\ t2)\ ts$$

where:

$$\begin{array}{lcll} \mathit{insert}\ u\ [\,] & = & [u] & \\ \mathit{insert}\ u\ (t : ts) & = & u : t : ts, & \textbf{if}\ \mathit{weight}\ u \le \mathit{weight}\ t \\ & = & t : \mathit{insert}\ u\ ts, & \textbf{otherwise} \end{array}$$

Although this definition of *combine* is adequate, it leads to an inefficient algorithm as tree weights are constantly recomputed. A better solution is to store the weights in the tree as labels, and this is where the idea of using a labelled binary tree comes in. Consider the type:

$$\mathit{htree} ::= \mathit{Leaf}\ \mathit{num}\ \mathit{char} \mid \mathit{Node}\ \mathit{num}\ \mathit{htree}\ \mathit{htree}$$

Here, a tip node is indicated by a new constructor *Leaf* that takes two arguments: a number and a character. A binary node is indicated by the new constructor *Node*. The numeric label is a value w satisfying:

$$w = \mathit{weight}\,(\mathit{Node}\ w\ t1\ t2)$$

where:

$$\begin{array}{lcl} \mathit{weight}\,(\mathit{Leaf}\ w\ x) & = & w \\ \mathit{weight}\,(\mathit{Node}\ w\ t1\ t2) & = & \mathit{weight}\ t1 + \mathit{weight}\ t2 \end{array}$$

By maintaining this relationship we can avoid recomputing weights.

To implement the change, we redefine:

$$\begin{array}{lcl} \mathit{build} & = & \mathit{unlabel} \cdot (\mathit{until}\ \mathit{single}\ \mathit{combine}) \cdot \mathit{map}\ \mathit{leaf} \\ \mathit{leaf}\,(x, w) & = & \mathit{Leaf}\ w\ x \end{array}$$

where *unlabel* throws away the labels after they have served their purpose:

$$\begin{array}{lcl} \mathit{unlabel}\,(\mathit{Leaf}\ w\ x) & = & \mathit{Tip}\ x \\ \mathit{unlabel}\,(\mathit{Node}\ w\ t1\ t2) & = & \mathit{Bin}\,(\mathit{unlabel}\ t1)\,(\mathit{unlabel}\ t2) \end{array}$$

We must also modify the definition of *combine*:

$$\begin{array}{lcl} \mathit{combine}\,(t1 : t2 : ts) & = & \mathit{insert}\,(\mathit{Node}\ w\ t1\ t2)\ ts \\ & & \quad \textbf{where}\ \ w = \mathit{wt}\ t1 + \mathit{wt}\ t2 \\ & & \\ \mathit{wt}\,(\mathit{Leaf}\ w\ x) & = & w \\ \mathit{wt}\,(\mathit{Node}\ w\ t1\ t2) & = & w \end{array}$$

The modification of *insert* is left as an exercise.

Let us see this algorithm at work on an example. Consider the sequence:

$$[(\text{'G'},8),(\text{'R'},9),(\text{'A'},11),(\text{'T'},13),(\text{'E'},17)]$$

of characters and their weights. The first step is to convert this into the list:

[*Leaf* 8 'G', *Leaf* 9 'R', *Leaf* 11 'A', *Leaf* 13 'T', *Leaf* 17 'E']

of leaf trees. The next step is to combine the first two trees and rearrange the resulting trees in increasing order of weight. Thus we obtain:

```
[Leaf 11 'A',
 Leaf 13 'T',
 Node 17 (Leaf 8 'G') (Leaf 9 'R'),
 Leaf 17 'E']
```

The result of the second step is the list:

```
[Node 17 (Leaf 8 G) (Leaf 9 'R'),
 Leaf 17 'E',
 Node 24 (Leaf 11 'A') (Leaf 13 'T')]
```

The third step gives:

```
[Node 24 (Leaf 11 'A') (Leaf 13 'T'),
 Node 34 (Node 17 (Leaf 8 'G') (Leaf 9 'R'))
         (Leaf 17 'E')]
```

so the final tree is:

```
Node 58 (Node 24 (Leaf 11 'A') (Leaf 13 'T'))
        (Node 34 (Node 17 (Leaf 8 'G') (Leaf 9 'R'))
                 (Leaf 17 'E'))
```

In this tree, which is pictured in Figure 9.3, the characters 'A', 'T' and 'E' are coded by two-bit sequences and 'G' and 'R' by three-bit sequences.

The average, or expected, length of a character code is:

$$\sum w_j l_j / \sum w_j$$

where l_j is the number of bits assigned to character c_j. In the above example this value is:

$$((11+13+17)\times 2+(8+9)\times 3)/(8+9+11+13+17)$$

or approximately 2.29. The crucial property of a Huffman code is that it minimises expected length. Putting it another way, a Huffman tree has the property that it is a binary tree, over tip values $w_1, w_2, \ldots, w_n$, which minimises the sum of the "weighted" path lengths $w_j l_j$ for $1 \le j \le n$. For a proof of this fact, the reader should consult Knuth [3] or Standish [11].