9 Huffman minimum redundancy coding

It has become usual to store data and transmit messages using fixed length codes such as ASCII. The character set is represented by some number of codewords, each of the same length, which in the case of ASCII is seven binary digits. The result is that it takes the same number of bits to store, or the same bandwith to transmit, all messages with the same number of characters. Of course, if you know in advance that your message is in, say, English, then you know that it is much less likely to contain letter 'z's than letter 'e's. This means that if you use a shorter codeword to represent 'e' than that used for 'z', you can expect to use less store, or bandwidth, for the average message.

In ASCII, the message 'easily' is encoded

1100101	1100001	1110011	1101001	1101100	1111001
e	a	s	i	l	y

requiring forty-two bits, whereas by using a code which included the following representations

$$
\begin{array}{lll}
a & \mapsto & \mathtt{1\ 0\ 0\ 1} \\
e & \mapsto & \mathtt{0} \\
i & \mapsto & \mathtt{1\ 0\ 1\ 0} \\
l & \mapsto & \mathtt{1\ 1\ 0\ 0\ 1} \\
s & \mapsto & \mathtt{1\ 1\ 0\ 1\ 0} \\
y & \mapsto & \mathtt{1\ 0\ 1\ 1}
\end{array}
$$

the same message can be encoded

```
0 1 0 0 1 1 1 0 1 0 1 0 1 0 1 1 0 0 1 1 0 1 1
e a       s         i       l         y
```

in only twenty-one bits. The codewords must be chosen in such a way that none is a prefix of any of the others, to ensure that there can be only one way of decoding a particular coded text.

In a classic paper, published in 1952, David Huffman described an algorithm to find the set of codes that would minimize the expected length of a message, given that the probability of each character's occurring were known. Essentially, his method decides the lengths of codewords, giving the longest to the least likely characters. It then remains to create an arbitrary unambiguous code with codewords of the right lengths.

The terminology of Huffman's paper is a little different from that in use today, as indicated in the brackets. He uses the term 'message' to mean an individual character. First of all, the ensemble [= character set] is sorted in decreasing order of probability:

> [It is] necessary that the two least probable messages [= characters] have codes of equal length ... [and that] there be only two of the messages with coded length $L(N)$ which are identical except for their last digits. The final digits of these two codes will be one of the two binary digits, 0 and 1. It will be necessary to assign these two message codes to the Nth and $(N-1)$st messages [= two least probable characters] since at this point it is not known whether or not other codes of length $L(N)$ exist. Once this has been done, these two messages are equivalent to a single composite message. Its code (as yet undetermined) will be the common prefixes of order $L(N)-1$ of these two messages. Its probability will be the sum of the probabilities of the two messages from which it was created. The ensemble containing this composite message in the place of its two component messages will be called the first auxiliary message ensemble.
>
> This newly created ensemble contains one less message than the original. Its members should be rearranged if necessary so that the messages are again ordered according to their probabilities. It may be considered exactly as the original ensemble was. ...
>
> This procedure is applied again and again until the number of messages in the most recently formed auxiliary ensemble is reduced to two. One of each of the binary digits is assigned to each of these two composite messages. These messages are then combined to form a single composite mes-

> sage with probability unity, and the coding is complete. ...
>
> Having now decided proper lengths of code for each message, the problem of specifying the actual digits remains. Since the combining of messages into their composites is similar to the successive confluences of trickles, rivulets, brooks, and creeks into a final large river, the procedure thus far described might be considered analogous to the placing of signs by a water-borne insect at each of these junctions as he journeys downstream. ...the code we desire is that one which the insect must remember in order to work his way back upstream.
>
> A method for the construction of minimum-redundancy codes, David A. Huffman in Proc I.R.E., 40 (9), September 1952, pp. 1098-1101

Restated more prosaically, the final paragraph identifies the unambiguous set of codewords with a (binary) tree. Each leaf of the tree corresponds to one of the characters. The depth of that leaf, that is its distance from the root, is the length of that character's codeword. The digits of the codeword are the 'address' of the leaf, that is a sequence of instructions for getting to the leaf from the root, say 0 for 'go to the left' and 1 for 'go to the right'.

Representing a coding tree

As usual, the task of representing a data structure in *occam* amounts to choosing an enumeration for the component parts, so as to map the structure onto a linear array. The structure in question this time is a binary tree similar to that in the sorting example, but this tree may be severely imbalanced, and is of unpredictable depth. This means that the simple fixed enumeration, with the children of node i being nodes $2i+1$ and $2i+2$, would be unreasonably wasteful of store, so is unsuitable. A better representation, in this case, roots the tree at node zero

```
DEF root = 0 :
```

and uses an array *eldest*[ ] to record the index of the leftmost offspring of a node, so that the children of node i are indexed $eldest[i]$ and $eldest[i]+1$.

Since the root is by definition not the child of any node, the test $eldest[node] = root$ can be used to signify that *node* is a leaf of the tree. In the case of the leaves, it will be necessary to know to which character they correspond. This is most readily recorded in another array of the same size as *eldest*[ ] in which the value of *character*[*node*] is the character corresponding to that node, if it is a leaf.

The array *eldest*[ ] makes it easy to pass 'upstream' from the root of the tree to the leaves. In order to make the 'downstream' journey as efficient, it will be useful to record the inverse of *eldest*[ ], in an array *parent*[ ], such that

$$parent[eldest[node]] = parent[eldest[node] + 1] = node$$

for each non-leaf node, and the inverse of *character*[ ] in an array *representative*[ ], which records the index of the leaf corresponding to each character.

It remains to be decided how big these arrays must be. This, of course, depends on the size of the character set being encoded. For the purposes of this example, the (unencoded) character set will be signed, eight-bit significant values,

$$-128 \leq ch < 128$$

This allows room for the normal seven-bit characters in the non-negative half range, and room for another, negative, character set which can be used for control information, indicating such things as the end of a message.

```
DEF bits.in.character    = 8,
    number.of.characters = 1 << bits.in.character,
    number.of.codes      = number.of.characters,
    character.mask = NOT((NOT 0)<<bits.in.character) :
```

The *character.mask* has ones as its *bits.in.character* least significant bits, and zeroes elsewhere, for mapping signed characters onto non-negative array indexes, so that, for example,

$$ch = character[representative[ch \wedge character.mask]]$$

Now if there are *number.of.codes* leaves in a binary tree, then there will be one less than that number of non-leaf nodes, so the total number of nodes is given by

```
DEF size.of.tree = (2 * number.of.codes) - 1 :
```

and the declarations of the arrays for representing the tree are

```
VAR eldest[size.of.tree],
    parent[size.of.tree],
    character[size.of.tree],
    representative[number.of.characters] :
```

Constructing a coding tree

Huffman's algorithm proceeds in two stages. First the character set is sorted into descending order of probability of the character's occurrence. Each of the characters will correspond to a leaf of the tree, so you can think of this stage of the process as constructing *number.of.codes* number of leaves. These leaves will be sub-trees of the final coding tree. Since each is just a leaf, they are disjoint, in the sense that they share no nodes with each other, and they are maximal, in the sense that there is not yet any bigger tree of which any is a member.

The second stage of the algorithm repeatedly reduces the size of the collection of maximal disjoint sub-trees, by combining the two lightest trees to make one new composite tree. By 'lightest' I mean of least *weight* where the weight of a leaf is the probability of the corresponding character, and the weight of a larger tree is the sum of the weights of its leaves. Notice that during this second stage, it is guaranteed that any two siblings—children of a common parent—are already adjacent in descending order of weight. This observation, which I take from Robert Gallager

> A prefix condition code is a code with the property that no codeword is a prefix of any other codeword. A binary tree has the sibling property if each node (except the root) has a sibling, and if the nodes of the tree can be arranged in order of non-increasing probability with each node being adjacent to its sibling. A binary prefix condition code is a Huffman code iff the code tree has the sibling property.
>
> Variations on a Theme by Huffman, Robert G. Gallager
> in IEEE Trans. Information Theory, IT-24(6), 1978, pp. 668-674

is in fact a non-algorithmic characterization of Huffman codes. It also shows that in the representation chosen for the coding tree, which allocates adjacent elements of the arrays to siblings, it is possible to keep the arrays sorted in descending order of weight. Gallager's proof that this property holds is, essentially, an informal proof of correctness of Huffman's algorithm.

Keeping the arrays sorted by weight of node in this way simplifies the finding of the two lightest sub-trees, and if the arrays are filled from the high-index, light, end towards the root, then sub-trees once constructed need not be moved again.

I have divided the algorithm into three parts

```
PROC construct.tree(VALUE probability[]) =
  VAR left.limit, right.limit, weight[size.of.tree] :

  PROC construct.leaves =
    ... build the leaves of the tree

  PROC construct.other.nodes =
    ... join pairs of subtrees until only one remains

  PROC invert.representation =
    ... set parent[] and representative[]

  SEQ
    left.limit   := size.of.tree + 1
    right.limit  := size.of.tree + 1

    -- left.limit = (size.of.tree + 1)
    -- (right.limit - left.limit) = 0

    construct.leaves

    -- left.limit = number.of.codes
    -- (right.limit - left.limit) = number.of.codes

    construct.other.nodes

    -- left.limit = root
    -- (right.limit - left.limit) = 1

    invert.representation     :
```

Throughout, the collection of maximal disjoint sub-trees consists of those trees rooted at nodes for which

$$\mathit{left.limit} \leq \mathit{node} < \mathit{right.limit}$$

The initialization of the limits makes this collection empty. The process *construct.leaves* introduces a new sub-tree into the collection for each of the characters of the character set, setting its weight according to the probability of the character, maintaining the arrangement of the leaves in descending order, so that

$$\mathit{left.limit} \le i \le j < \mathit{size.of.tree} \quad \Longrightarrow \quad \mathit{weight}[i] \ge \mathit{weight}[j]$$

The process *construct.other.nodes* combines the two lightest leaves, which are those nearest to *right.limit*. It introduces a new node with the combined weight of these two, adjusting the limits of the collection, and filling in the shape of the tree in *eldest*[ ]. Finally, the process *invert.representation* constructs the arrays *parent*[ ] and *representative*[ ].

Both *construct.leaves* and *construct.other.nodes* repeatedly create a new node of some given weight, and insert it into the right place between the limits to maintain the weight ordering of the nodes. The determination of this right place, and the consequent adjustment of the lighter nodes is done by

```
PROC insert.new.node( VAR new.node,
                      VALUE weight.of.new.node,
                      VAR left.limit,
                      VALUE right.limit ) =
  VAR weight.limit :
  SEQ
    IF
      IF node = [left.limit FOR right.limit-left.limit]
        weight[node] <= weight.of.new.node
          weight.limit := node
      TRUE
        weight.limit := right.limit
    SEQ node = [left.limit FOR weight.limit-left.limit]
      SEQ
        character[node - 1] := character[node]
        eldest[node - 1]    := eldest[node]
        weight[node - 1]    := weight[node]
    left.limit       := left.limit - 1
    new.node         := weight.limit - 1
    weight[new.node] := weight.of.new.node      :
```

Recall that the collection of maximal disjoint sub-trees of the coding tree so far constructed consists of those rooted at nodes

$$\mathit{left.limit} \le \mathit{node} < \mathit{right.limit}$$

and that they are in descending order of weight. This means that the conditional sets the *weight.limit* so that

$$\begin{aligned} \mathit{left.limit} \le \mathit{node} < \mathit{weight.limit} \quad &\Longrightarrow \\ &\mathit{weight}[\mathit{node}] > \mathit{weight.of.new.node} \\ \mathit{weight.limit} \le \mathit{node} < \mathit{right.limit} \quad &\Longrightarrow \\ &\mathit{weight.of.new.node} \ge \mathit{weight}[\mathit{node}] \end{aligned}$$

The sequential loop then displaces each of the heavier nodes one place to the left to make room for the *new.node*, and the *left.limit* of the collection is adjusted to compensate. The shapes of the sub-trees remain the same, so apart from being shifted up by one no changes are necessary to the value in *eldest*[ ]. This is because only nodes to the left of the *weight.limit* are moved, but

$$\mathit{node} < \mathit{weight.limit} \quad \Longrightarrow \quad \mathit{node} < \mathit{right.limit}$$

and the construction guarantees

$$(\mathit{eldest}[\mathit{node}] = \mathit{root}) \vee (\mathit{eldest}[\mathit{node}] \ge \mathit{right.limit})$$

so that no node which is moved is yet a child.

Using this process, *insert.new.node*, the process that creates the leaf nodes can be written

```
PROC construct.leaves =
  DEF minimum.character = -(number.of.characters / 2) :
  SEQ ch = [minimum.character FOR number.of.characters]
    VAR new.node :
    SEQ
      insert.new.node(new.node,
                 probability[ch /\ character.mask],
                 left.limit, right.limit)
      eldest[new.node]    := root
      character[new.node] := ch                :
```

This inserts a new leaf into the collection, increasing the size of the collection by decreasing the *left.limit*. The process to combine the leaves into a tree

```
PROC construct.other.nodes =
  WHILE (right.limit - left.limit) <> 1
    VAR new.node :
    SEQ
      right.limit := right.limit - 2
      insert.new.node(new.node,
           weight[right.limit] + weight[right.limit+1],
           left.limit, right.limit)
      eldest[new.node] := right.limit     :
```

first removes the two lightest sub-trees from the collection, by decreasing *right.limit*, then joins them under a parent whose weight is the sum of their individual weights. Notice that the assignment to *eldest*[*new.node*] maintains the property that there are no children to the left of the *right.limit*. The process is complete when only one tree remains.

Inverting the representation of the tree is a simple task, which involves assigning to *representative*[ ] the indexes of the leaf nodes, and to *parent*[ ] the indexes of the nodes that are not leaves, thus

```
PROC invert.representation =
  SEQ node = [root FOR size.of.tree]
    IF
      eldest[node] = root
        representative[character[node] /\
                                character.mask] := node
      eldest[node] <> root
        SEQ child = [eldest[node] FOR 2]
          parent[child] := node                   :
```

That completes the code to construct a coding tree from a given probablity distribution.

Encoding and decoding using a coding tree

The encoding of any given character *ch* is the sequence of 'go left' and 'go right' instructions that Huffman's insect must follow to pass upstream from the root node to the representative node of that character. It is easy enough to construct this code backwards, since floating downstream involved passing from *node* to *parent*[*node*] in succession from the representative node until the root is reached. The process

```
SEQ
  length := 0
  node := representative[ch /\ character.mask]
  WHILE node <> root
    SEQ
      encoding[length] := node - eldest[parent[node]]
      length           := length + 1
      node             := parent[node]
```

establishes the condition that

$$\bigwedge_{0 \leq i < length} node_i = (eldest[node_{i+1}] + encoding[i])$$

where

$$\begin{aligned} node_0 &= representative[ch \wedge character.mask] \\ node_{length} &= root \end{aligned}$$

so that the encoding of *ch* can be transmitted in the right order by

```
SEQ i = [1 FOR length]
  output ! encoding[length - i]
```

It remains only to decide how much room needs to be allocated to store the *encoding*[ ] whilst it is being constructed. Assume that you are decoding a Huffman encoded character. Before you receive the first bit of the encoding, there are *number.of.codes* possible codes that you might be about to receive. Each bit that you receive divides the set of possible characters into two non-empty sub-sets: those that are still possible, and those that are now precluded. This means that at most $number.of.codes - 1$ bits will suffice. In fact, in the worst case, this limit is achieved: if each character is more than twice as probable as the next most likely, then the Huffman codes are, in decreasing order of probability

$$0, \quad 10, \quad 110, \quad 1110, \quad 11110, \quad \ldots$$

with the two least probable characters both having encodings that are $number.of.codes - 1$ bits long. With this knowledge, the encoding process is written

```
PROC encode.character(CHAN output, VALUE ch) =
  -- Transmit the encoding of ch along output
  DEF size.of.encoding = number.of.codes - 1 :
  VAR encoding[size.of.encoding], length, node :
  SEQ
    length := 0
    node   := representative[ch /\ character.mask]
    WHILE node <> root
      SEQ
        encoding[length] := node - eldest[parent[node]]
        length := length + 1
        node   := parent[node]
    SEQ i = [1 FOR length]
      output ! encoding[length - i]              :
```

Decoding a stream of bits to determine the character consists of following the ‘go left’ and ‘go right’ instructions as they arrive

```
SEQ
  input ? bit
  node := eldest[node] + bit
```

Enough bits must be accepted to pass ‘upstream’, starting from the root node until a leaf is reached. That leaf indicates the decoded character

```
PROC decode.character(CHAN input, VAR ch) =
  VAR node :
  SEQ
    node := root
    WHILE eldest[node] <> root
      VAR bit :
      SEQ
        input ? bit
        node := eldest[node] + bit
    ch := character[node]               :
```

Having once fixed on a coding tree, the encoding process can be applied repeatedly to a sequence of characters to produce a sequence of bits which, by repeated application of the decoding process, can be turned back into the original sequence of characters. In this way, you can code a pair of processes that when executed in parallel duplicate the behaviour of a buffer like *copy*:

```
PROC copy(CHAN source, end.of.source, sink) =
  -- Copy characters from source to sink until a
  -- signal is received from end.of.source
  VAR more.characters.expected :
  SEQ
    more.characters.expected := TRUE
    WHILE more.characters.expected
      VAR ch :
      ALT
        source ? ch
          sink ! ch
        end.of.source ? ANY
          more.characters.expected := FALSE :
```

transmitting the minimum number of bits necessary to communicate the information in the sequence of characters.

I will assume that the probabilities of the characters are fixed in advance, perhaps by considering an average over many messages of the type to be sent.

```
DEF probability = TABLE[ ... ] :
            -- indexed by [0 FOR number.of.characters]
```

In order to keep all the arithmetic in integers, the probabilities should be scaled and rounded so that the total of the 'probabilities'

$$\sum_{ch} probability[ch]$$

is a large integer. If it is possible to read the message through before sending it, then you can count actual character frequencies, and produce an optimal Huffman code for the message, but of course you will then have to transmit a description of the code with your message!

If one of the character codes is laid aside to indicate the end of a transmission,

```
DEF end.of.message = -1 :
```

then that can be transmitted after the last real character of the message. That means that the end of the message is marked within the sequence of Huffman encodings of the characters.

The process

```
PROC copy.encoding(CHAN source, end.of.source, sink) =
  -- Read characters from source, sending their
  -- encodings along sink, until a signal is received
  -- along end.of.source.
  VAR more.characters.expected :
  SEQ
    construct.tree(probability)
    more.characters.expected := TRUE
    WHILE more.characters.expected
      VAR ch :
      ALT
        source ? ch
          encode.character(sink, ch)
        end.of.source ? ANY
          more.characters.expected := FALSE
    encode.character(sink, end.of.message)           :
```

translates a stream of characters into a stream of bits representing their Huffman encodings, and marks the end of the stream by sending the encoding of *end.of.message*. The corresponding decoding process would be

```
PROC copy.decoding(CHAN source, sink) =
  -- Read a bit stream from source, decoding it into
  -- characters and send these along sink until
  -- end.of.message is decoded
  VAR more.characters.expected :
  SEQ
    construct.tree(probability)
    more.characters.expected := TRUE
    WHILE more.characters.expected
      VAR ch :
      SEQ
        decode.character(source, ch)
        IF
          ch <> end.of.message
            sink ! ch
          ch = end.of.message
            more.characters.expected := FALSE    :
```

These processes can be used at the opposite ends of a serial communications medium

```
PROC copy.over.serial.medium(CHAN source,
                                  end.of.source, sink) =
  -- Copy characters from source to sink until a
  -- signal is received from end.of.source
  DEF end.of.message = -1,
      probability = TABLE[ ... ] :
  CHAN serial.medium :
  PAR
    copy.encoding(source, end.of.source, serial.medium)
    copy.decoding(serial.medium, sink)                :
```

where each communication on *serial.medium* is the transmission of a single bit. The process *copy.over.serial.medium* is a buffer: provided a sequence of bytes is communicated on *source* which does not include *end.of.message* (if for example they are all seven-bit character codes) either *copy* or *copy.over.serial.medium* can be substituted for the other in any program without affecting the behaviour of the program.

Many communications media are most efficiently used by sending fixed sized messages, consisting of large numbers of bits. You already have most of the components necessary to implement a process like *copy* which makes efficient use of such a medium. Encoding characters and packing the bits into blocks can be done by *copy.encoding* executing concurrently with *pack.bits.into.blocks* from an earlier section.

```
PROC encode.into.blocks(CHAN source, end.of.source,
                                          block.sink) =
  CHAN bit.stream, end.of.bit.stream :
  PAR
    SEQ
      copy.encoding(source, end.of.source, bit.stream)
      end.of.bit.stream ! ANY
    pack.bits.into.blocks(bit.stream,
                    end.of.bit.stream, block,sink) :
```

Decoding the characters from the stream of blocks is a slightly trickier task, since the end of the message is determined from the decoded data. The most elegant solution, as seems common in parallel programs, involves a process that throws away unwanted information

```
PROC discard(CHAN source, end.of.source) =
  VAR more.expected :
  SEQ
    more.expected := TRUE
    WHILE more.expected
      ALT
        source ? ANY
          SKIP
        end.of.source ? ANY
          more.expected := FALSE      :
```

This inputs successively from *source*, ignoring the values that it receives, until a signal is sent to it on *end.of.source*. With this, the process for decoding the bits in a stream of blocks can be written

```
PROC decode.from.blocks(CHAN block.source, sink) =
  CHAN end.of.block.source,
       bit.stream, end.of.bit.stream :
  PAR
    SEQ
      unpack.bits.from.blocks(block.source,
                       end.of.block.source, bit.stream)
      end.of.bit.stream ! ANY         -- 'feed-forward'

    SEQ
      copy.decoding(bit.stream, sink)
      PAR
        discard(bit.stream, end.of.bit.stream)
        end.of.block.source ! ANY  :  -- 'feed-back'
```

When *copy.decoding* decodes an *end.of.message* it terminates, causing a signal to be offered for output on *end.of.block.stream*, which is a feed-back path to the block unpacking process. At the same time, *discard* absorbs any bits that were left in the last block of the message. When all of the bits of the last block have gone, *unpack.bits.from.blocks* accepts the *end.of.block.source* signal, and terminates, causing an *end.of.bit.stream* signal to be sent to terminate the *discard* process.

These two processes can be operated concurrently to simulate a buffer

```
PROC copy.over.blocked.medium(CHAN source,
                                  end.of.source, sink) =
  -- Copy characters from source to sink until a
  -- signal is received from end.of.source
  DEF end.of.message = -1,
      probability = TABLE[ ... ] :
  CHAN blocked.medium :
  PAR
    encode.into.blocks(source, end.of.source,
                                       blocked.medium)
    decode.from.blocks(blocked.medium, sink)          :
```

with each communication on the *blocked.medium* being a transfer of a block of the predetermined fixed size.

The *copy.over.blocked.medium* is not quite as good an implementation of a buffer as is the bit-serial copy. Consider what communications must happen before the first character is output on *sink*. This is a restriction on the contexts in which *copy.over.blocked.medium* can be substituted for *copy*.

Adapting the code to the message

So far, I have accepted Huffman's assumption that the code is predetermined and remains fixed throughout the transmission of a given message. This is reasonable in case the probability distribution of the characters in the message is known in advance, or if the message can be read through in advance. Gallager suggests an alternative encoding that tends in the long run towards the fixed Huffman encoding, but which starts with no knowledge of the probability distribution of the characters, adapting the code as the message is being sent.

Each character is encoded with a Huffman code that would be optimal for a message consisting of all those characters that have gone before it. This encoding technique has the startling property that, since the decoder has already decoded the preceding characters, it can deduce from the received message what code should be used to decode each character. There is no longer a problem in communicating the code as well as the message!

As I have presented it, it might seem that Gallager's adaptive Huffman coder requires that a new coding tree be constructed for each character of the transmitted and received message. Fortunately, this is not the case: the accumulated character frequencies change little, so the shape of the tree tends to settle down; successive trees are sufficiently similar that it is fairly easy to construct each from its predecessor.

The idea is to write a process *increment.frequency*(*ch*) which modifies the coding tree so as to be consistent with a frequency distribution with one more occurrence of the character *ch* than previously. The encoding process becomes

```
PROC copy.encoding(CHAN source, end.of.source, sink) =
  VAR more.characters.expected :
  SEQ
    construct.tree
    more.characters.expected := TRUE
    WHILE more.characters.expected
      VAR ch :
      ALT
        source ? ch
          SEQ
            encode.character(sink, ch)
            increment.frequency(ch)
        end.of.source ? ANY
          more.characters.expected := FALSE
    encode.character(sink, end.of.message)         :
```

and the corresponding decoding process would be

```
PROC copy.decoding(CHAN source, sink) =
  VAR more.characters.expected :
  SEQ
    construct.tree
    more.characters.expected := TRUE
    WHILE more.characters.expected
      VAR ch :
      SEQ
        decode.character(source, ch)
        IF
          ch <> end.of.message
            SEQ
              sink ! ch
              increment.frequency(ch)
          ch = end.of.message
            more.characters.expected := FALSE    :
```

To keep track of the accumulated frequencies, the *weight*[ ] must become a permanent part of the representation of the tree

```
VAR weight[size.of.tree] :
```

In order to increment the recorded frequency of a character, it is necessary to increment the weight of its representative leaf

```
VAR node :
SEQ
  node := representative[ch /\ character.mask]
  weight[node] := weight[node] + 1
```

There are two ways in which this may have damaged the structure of the tree. First of all, unless the tree has only the one node, the weight of the parent of node is no longer the sum of the weights of its children: it will be necessary to increment the weights of the parent of the node, and all of its ancestors up to the root

```
VAR node :
SEQ
  node := representative[ch /\ character.mask]
  WHILE node <> root
    SEQ
      weight[node] := weight[node] + 1
      node := parent[node]
  weight[root] := weight[root] + 1
```

Secondly, each time the weight of a node, be that the original leaf or one of its ancestors, is increased there is a danger that the ordering of the weights may be upset. If this is the case then it is time to reorganize the tree, and change the encoding.

Assuming that the tree is initially properly ordered, then the ordering will first fail when $weight[node-1] = weight[node]$ and the weight of *node* is about to be incremented. Now, the trees rooted at nodes of equal weight must be disjoint trees, that is either the nodes are siblings, or they have ancestors which are siblings. This follows from the fact that the weight of a node is always less than that of its ancestors, and greater than that of its descendants, so another node with the same weight is neither an ancestor nor a descendant.

To preserve the ordering on the nodes, it would be possible to exchange the trees rooted at *node* and $node-1$, and then to increment the weight of the light node in its new position. Since there might be many nodes with the same weight, however, this would have to be done repeatedly, shuffling the imminently overweight node leftwards in the tree.

```
WHILE weight[node-1] = weight[node]
  SEQ
    swap.trees(node, node - 1)
    node := node - 1
```

An alternative solution is to look for the leftmost node of the given weight, and exchange with that node, directly. The same argument about the weight of a node being less than that of its ancestors shows that there is always a sequence of nodes for which

$$weight[(node - i) - 1] > weight[node - i] = \ldots = weight[node]$$

This leftmost node, indexed $node - i$, is identified, and the exchange performed, by

```
IF i = [1 FOR (node - root) - 1]
  weight[(node - i) - 1] > weight[node]
    SEQ
      swap.trees(node, node - i)
      node := node - i
```

Having moved the node, it is possible to increment its weight, and that of each of its ancestors.

```
VAR node :
SEQ
  node := representative[ch /\ character.mask]
  WHILE node <> root
    IF
      weight[node-1] > weight[node]
        SEQ
          weight[node] := weight[node] + 1
          node := parent[node]
      weight[node-1] = weight[node]
        IF i = [1 FOR (node - root) - 1]
          weight[(node - i) - 1] > weight[node]
            SEQ
              swap.trees(node, node - i)
              node := node - i
              weight[root] := weight[root] + 1
```

The process for exchanging a pair of disjoint sub-trees is simply coded

```
PROC swap.trees(VALUE i, j) =
  -- Exchange disjoint sub-trees rooted at i and j

  PROC swap.words(VAR p, q) =
    -- Exchange values stored in p and q
    VAR t :
    SEQ
      t := p
      p := q
      q := t            :

  PROC adjust.offspring(VALUE i) =
    -- Restore downstream pointers to node i
    IF
      eldest[i] = root
        representative[character[i]/\character.mask]:=i
      eldest[i] <> root
        SEQ child = [eldest[i] FOR 2]
          parent[child] := i                    :

  SEQ
    swap.words(eldest[i], eldest[j])
    swap.words(character[i], character[j])
    adjust.offspring(i)
    adjust.offspring(j)                             :
```

First, the 'upstream' pointers, *eldest*[ ] and *character*[ ], to the nodes are exchanged, then the process *adjust.offspring* restores the 'downstream' pointers that are no longer correct. There is, of course, no need to exchange the weights of the nodes, since they were known to be equal.

The only remaining problem is to decide the shape of the initial coding tree: what encoding should be used to send the first character? The simplest solution would be to construct the initial tree on the assumption that all characters are equally likely to turn up, say

$$\mathit{eldest}[\mathit{node}] = \mathit{root} \quad \Longrightarrow \quad \mathit{weight}[\mathit{node}] = 1$$

This means that, to begin with, the code is a fixed length one, each character being encoded by *bits.in.character* number of bits.

An alternative technique is to keep in the coding tree only representations of characters that have actually been sent and received. Whenever a character is to be sent for the first time in the message, the code of a special escape 'character' is sent, followed by some standard representation of the new character, say its ASCII code. A new leaf must then be added to the tree to represent the new character.

In order to accommodate the escape character, the space allocated for the tree must be enlarged

```
DEF number.of.codes = number.of.characters + 1 :
```

and, since the tree grows, some way must be found of recording that size. As each escape is the representation of a character that has never occurred at all (you may not yet know which character, but you do know this), it should be given a very low weight. This means that it is reasonable to represent it by the rightmost (least likely) leaf of the tree. Doing this means that a single variable

```
VAR escape :
```

serves the purpose of recording which node represents the escape, and which is the rightmost node of the tree.

Since the value of *escape* changes, it will not do to use it as an initial value for *representative*[ ]. Define, instead,

```
DEF not.a.node = size.of.tree :
```

then creating the initial tree is just a matter of making the escape leaf, and initializing the array of representatives

```
PROC construct.tree =
  SEQ
    escape := root
    weight[escape] := 1     -- minimum legal weight
    eldest[escape] := root  -- it is a leaf
    SEQ ch = [0 FOR number.of.characters]
      representative[ch] := not.a.node          :
```

Encoding using the new tree is substantially unchanged, excepting in that some provision must be made for sending escaped characters. First of all, the encoding is potentially larger by the *bits.in.character* number of bits in the unencoded representation, so

```
DEF size.of.encoding = bits.in.character +
                         (number.of.codes - 1)  :
```

The bits of the unencoded character representation can then be stored before the encoding of escape, to be transmitted after it.

```
SEQ i = [0 FOR bits.in.character]
  encoding[i] := (ch >> i) /\ 1
```

The encoding of the escape changes as the tree is modified. It is of course to be found by applying the same algorithm to the node representative of escape, as would be applied to the representative of any character.

```
PROC encode.character(CHAN output, VALUE ch) =
  -- Transmit the encoding of ch along output
  DEF size.of.encoding = bits.in.character +
                            (number.of.codes - 1)  :
  VAR encoding[size.of.encoding], length, node :
  SEQ
    IF
      representative[ch/\character.mask] <> not.a.node
        SEQ
          length := 0
          node   := representative[ch/\character.mask]
      representative[ch/\character.mask] = not.a.node
        SEQ
          SEQ i = [0 FOR bits.in.character]
            encoding[i] := (ch >> i) /\ 1
                               -- i'th bit of unencoded ch
          length := bits.in.character
          node   := escape
    WHILE node <> root
      SEQ
        encoding[length] := node - eldest[parent[node]]
        length := length + 1
        node   := parent[node]
    SEQ i = [1 FOR length]
      output ! encoding[length - i]                :
```

The very first character to be sent will be escaped, and since the representative node for *escape* is initially *root* the encoding of the escape will be the null sequence of bits. This means that the first transmitted bit will be the first bit of the unencoded character representation.

Decoding is also similar to decoding with a fixed code, excepting that on receipt of the encoding of escape, the bits of the unencoded escaped character must be read and the character reassembled.

The first bit of an escaped sequence is the sign bit of the character code, which the assignment

```
SEQ
  input ? bit
  ch := - bit
```

extends to the left, and subsequent bits can be shifted in from the right.

In any case, the first action is to accept enough bits to represent a leaf node: initially a sequence of no bits is enough to represent the only leaf, which is *escape*. Then, if the leaf represents a character, that character has been decoded; if the leaf represents the escape, the character code is reconstructed

```
PROC decode.character(CHAN input, VAR ch) =
  -- Receive an encoding along input and store the
  -- corresponding character in ch
  VAR node :
  SEQ
    node := root
    WHILE eldest[node] <> root
      VAR bit :
      SEQ
        input ? bit
        node := eldest[node] + bit
    IF
      node < escape
        ch := character[node]
      node = escape
        VAR bit :       -- read signed character code
        SEQ
          input ? bit
          ch := - bit
          SEQ i = [2 FOR bits.in.character - 1]
            SEQ
              input ? bit
              ch := (ch << 1)  bit                :
```

In order to increment the frequency of a character not yet in the tree, it is necessary to be able to construct a new leaf to be the representative of the new character. The new character has never previously been received, so its initial weight should be as low as possible, which means that it belongs next to the *escape* node, at the low-probability end of the tree.

The simplest way of adding a node to the tree is to create a node of zero weight, and then increment its weight (to one) exactly as though it had previously existed. This process divides the escape leaf into three: two new leaves and their parent

```
PROC create.leaf(VAR new.leaf, VALUE ch) =
  -- Extend the tree by fision of the escape leaf
  -- into two new leaves
  VAR new.escape :
  SEQ
    new.leaf            := escape + 1
    new.escape          := escape + 2

    eldest[escape]      := new.leaf
                -- the old escape is the new parent

    weight[new.leaf]    := 0
    eldest[new.leaf]    := root
    parent[new.leaf]    := escape
    character[new.leaf] := ch
    representative[ch /\ character.mask] := new.leaf

    weight[new.esape]   := 1
    eldest[new.escape]  := root
    parent[new.escape]  := escape

    escape              := new.escape              :
```

The new leaf has no weight when created, so there is no effect on the weights of its ancestors. Notice that a brand new leaf having no weight, the earlier data invariant—that no node has the same weight as its parent—is breached by the *escape* node and its parent. This invariant was used to show that the sub-tree exchanging was correct, so that argument has to be re-constructed. The statement of the invariant must be strengthened: no node, excepting the *escape* node has the same weight as its parent. This is sufficient (as you should check) because it is never required that the one-node tree rooted at *escape* be exchanged with any other (why not?).

Creating a new representative with zero weight makes new characters like those already present in the tree. In either case, recording the arrival of a character is a matter of finding a representative leaf for it, and then incrementing the weight of that leaf and making the consequential adjustments to the tree.

```
PROC increment.frequency(VALUE ch) =
  VAR node :
  SEQ
    IF
      representative[ch/\character.mask] <> not.a.node
        node := representative[ch /\ character.mask]
      representative[ch/\character.mask] = not.a.node
        create.leaf(node, ch)
    WHILE node <> root
      IF
        weight[node-1] > weight[node]
          SEQ
            weight[node] := weight[node] + 1
            node := parent[node]
        weight[node-1] = weight[node]
          IF i = [1 FOR (node - root) - 1]
            weight[(node - i) - 1] > weight[node]
              SEQ
                swap.trees(node, node - i)
                node := node - i
    weight[root] := weight[root] + 1            :
```

That completes the adaptive coder. Notice that, since the processes *copy.encoding* and *copy.decoding* have the same interfaces as the corresponding processes in the fixed-code coder, they may be substituted into the example *copy*... processes. There is no need to change the processes that convey the bit stream from encoder to decoder.