

Adaptive Huffman coder

```
PROC construct.tree  
  
PROC create.leaf(VAR new.leaf, VALUE ch)  
  
PROC swap.trees(VALUE i, j)  
    PROC swap.words(VAR p, q)  
    PROC adjust.offspring(VALUE i)  
  
PROC increment.frequency(VALUE ch)  
  
PROC encode.character(CHAN output, VALUE ch)  
  
PROC decode.character(CHAN input, VAR ch)  
  
PROC copy.encoding(CHAN source, end.of.source, sink)  
  
PROC copy.decoding(CHAN source, sink)
```

```

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

DEF root                    = 0,
    size.of.tree           = (2 * number.of.codes) - 1,
    not.a.node             = size.of.tree, :

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

PROC construct.tree =
    -- Create a tree for the encoding in which every character is escaped
    SEQ
        escape := root
        weight[escape] := 1
        eldest[escape] := root      -- it is a leaf
        SEQ ch = [0 FOR number.of.characters]
            representative[ch] := not.a.node          :

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      -- 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          :

```

```

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)          :

```

```

PROC increment.frequency(VALUE ch) =
  -- Adjust the weights of all relevant nodes to account for one more
  -- occurrence of ch, and adjust the shape of the tree if necessary
  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 :

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] :

```

```

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 :
      SEQ
        input ? bit
        ch := - bit
      SEQ i = [2 FOR bits.in.character - 1]
      SEQ
        input ? bit
        ch := (ch << 1) \/ bit          :

DEF end.of.message = -1 :

PROC copy.encoding(CHAN source, end.of.source, sink) =
  -- Read a stream of characters from source, until signalled on
  -- end.of.source, and transmit their encodings in sequence along
  -- sink, followed by that of end.of.message, maintaining throughout
  -- the encoding tree for the encoding determined by the cumulative
  -- frequencies of the characters transmitted
  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)      :

```

```
PROC copy.decoding(CHAN source, sink) =
  -- Read the encodings of a stream of characters, up to and including
  -- the encoding of end.of.message, from source and transmit the
  -- corresponding characters along sink, maintaining the encoding tree
  -- for the encoding determined by the cumulative frequencies of the
  -- characters received
  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 :
```