Adaptive Huffman coder

```
--  This chapter contains routines to transmit and receive character
--  streams encoded by an adaptive Huffman code. The code used for
--  each character is one given by Huffman's algorithm for those
--  characters which have preceded it.
--

--
--  character set description
--

VAL INT bits.in.character    IS 8 :
VAL INT number.of.characters IS 1 << bits.in.character :
VAL INT number.of.codes      IS number.of.characters + 1 :

VAL INT character.mask IS BITNOT ((BITNOT 0) << bits.in.character) :

INT FUNCTION index(VAL INT char) IS char /\ character.mask :

--
--  structure of the tree
--

VAL INT root IS 0 :
VAL INT size.of.tree IS (2 * number.of.codes) - 1 :

VAL INT not.a.node IS size.of.tree :

VAL INT number.of.arrays IS 5 :

VAL INT eldest.index         IS 0 :
VAL INT parent.index         IS 1 :
VAL INT character.index      IS 2 :
VAL INT representative.index IS 3 :
VAL INT weight.index         IS 4 :
```

```
--
--  create.leaf
--

PROC create.leaf(INT new.leaf, [][]INT tree, VAL INT char)
  --
  --  extend the tree by fission of the escape into two new leaves
  --
  []INT eldest IS tree[eldest.index] :
  []INT parent IS tree[parent.index] :
  []INT character IS tree[character.index] :
  []INT representative IS
      [tree[representative.index] FROM 0 FOR number.of.characters] :
  []INT weight IS tree[weight.index] :
  INT escape IS tree[representative.index][number.of.characters] :

  INT new.escape :
  SEQ
    new.leaf, new.escape := escape + 1, escape + 2
    eldest[escape] := new.leaf -- old escape is new parent
    weight[new.leaf],   eldest[new.leaf],   parent[new.leaf] :=
        0,                  root,               escape
    character[new.leaf], representative[index(char)] :=
        char,               new.leaf
    weight[new.escape], eldest[new.escape], parent[new.escape] :=
        1,                  root,               escape
    escape := new.escape
:

--
--  compare.weights
--

VAL INT lighter     IS -1 :
VAL INT same.weight IS  0 :
VAL INT heavier     IS  1 :

INT FUNCTION compare.weights(VAL [][]INT tree, VAL INT a, b)
  INT result :
  VALOF
    VAL []INT weight IS tree[weight.index] :
    IF
      weight[a] < weight[b]
        result := lighter
      weight[a] = weight[b]
        result := same.weight
      weight[a] > weight[b]
        result := heavier
    RESULT result
:
```

```
--
--  initialize.tree, swap.trees
--

PROC initialize.tree([][]INT tree)

  []INT eldest IS tree[eldest.index] :
  []INT representative IS
      [tree[representative.index] FROM 0 FOR number.of.characters] :
  []INT weight IS tree[weight.index] :
  INT escape IS tree[representative.index][number.of.characters] :
  SEQ
    escape := root
    weight[escape] := 1     -- minimum legal weight
    eldest[escape] := root  -- it is a leaf
    SEQ char = 0 FOR number.of.characters
      representative[char] := not.a.node
:

PROC swap.trees([][]INT tree, VAL INT i, j)
  --
  --  exchange disjoint sub-trees rooted at i and j
  --
  PROC swap.ints(INT a, b)
    a, b := b, a
  :

  []INT eldest IS tree[eldest.index] :
  []INT parent IS tree[parent.index] :
  []INT character IS tree[character.index] :
  []INT representative IS
      [tree[representative.index] FROM 0 FOR number.of.characters] :

  PROC adjust.offspring(VAL INT i)
    --
    --  restore downstream pointers to node i
    --
    IF
      eldest[i] = root
        representative[index(character[i])] := i
      eldest[i] <> root
        SEQ child = eldest[i] FOR 2
          parent[child] := i
  :

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

```
--
--  increase.weight
--

PROC increase.weight([][]INT tree, VAL INT char)
  INT node :
  SEQ
    --
    --  check that the tree is not at its maximum weight
    --
    VAL INT limiting.weight IS MOSTPOS INT :
    INT FUNCTION heaviest.weight(VAL [][]INT tree) IS
                                    tree[weight.index][root] :
    IF
      heaviest.weight(tree) < limiting.weight
        SKIP
      heaviest.weight(tree) = limiting.weight
        initialize.tree(tree)
    --
    --  find a leaf to represent 'char'
    --
    []INT representative IS
      [tree[representative.index] FROM 0 FOR number.of.characters] :
    node := representative[index(char)]
    IF
      node <> not.a.node
        SKIP
      node = not.a.node
        create.leaf(node, tree, char)
    --
    --  increment the weight of 'node' and all of its ancestors
    --
    WHILE node <> root
      CASE compare.weights(tree, node - 1, node)
        heavier
          --  abbreviate parent and weight from tree
          []INT parent IS tree[parent.index] :
          []INT weight IS tree[weight.index] :

          SEQ
            weight[node] := weight[node] + 1
            node := parent[node]
        same.weight
          IF i = 1 FOR (node - root) - 1
            compare.weights(tree, (node-i)-1, node) = heavier
              SEQ
                swap.trees(tree, node, node - i)
                node := node - i
    []INT weight IS tree[weight.index] :
    weight[root] := weight[root] + 1
:
```

```
--
--  encode.character
--

PROC encode.character(CHAN OF BIT output,
                      VAL [][]INT tree, VAL INT char )

  VAL []INT eldest IS tree[eldest.index] :
  VAL []INT parent IS tree[parent.index] :
  VAL []INT representative IS
      [tree[representative.index] FROM 0 FOR number.of.characters] :
  VAL INT escape IS
                  tree[representative.index][number.of.characters] :

  VAL INT size.of.encoding IS
                          bits.in.character + (number.of.codes - 1) :
  [size.of.encoding]BOOL encoding :
  INT length, node :
  SEQ
    VAL INT leaf IS representative[index(char)] :
    IF
      leaf <> not.a.node
        length, node := 0, leaf
      leaf = not.a.node
        SEQ
          SEQ i = 0 FOR bits.in.character
            encoding[i] := ((char >> i) /\ 1) = 1
                          -- i'th bit of unencoded char
          length, node := bits.in.character, escape
    WHILE node <> root
      SEQ
        encoding[length] := node <> eldest[parent[node]]
        length, node := length + 1, parent[node]
    SEQ i = 1 FOR length
      output ! encoding[length - i]
:
```

```
--
--  decode.character
--

PROC decode.character(CHAN OF BIT input, VAL [][]INT tree, INT char)

  VAL []INT eldest IS tree[eldest.index] :
  VAL []INT character IS tree[character.index] :
  VAL INT escape IS
                tree[representative.index][number.of.characters] :

  INT node :
  SEQ
    node := root
    WHILE eldest[node] <> root
      BOOL go.right :
      SEQ
        input ? go.right
        IF
          go.right
            node := eldest[node] + 1
          NOT go.right
            node := eldest[node]
    IF
      node < escape
        char := character[node]
      node = escape
        BOOL bit :   -- read bits of signed character code
        SEQ
          input ? bit
          IF
            bit
              char := BITNOT 0
            NOT bit
              char := 0
          SEQ i = 2 FOR bits.in.character - 1
            SEQ
              input ? bit
              char := (char << 1) \/ (INT bit)
:
```

```
--
--  copy.encoding
--

VAL INT end.of.message IS -1 :

PROC copy.encoding(CHAN OF INT source,
                   CHAN OF SIGNAL end.of.source,
                   CHAN OF BIT sink             )
  --
  -- read characters from source, sending their encodings along sink,
  -- until a signal is received along end.of.source
  --
  BOOL more.characters.expected :
  [number.of.arrays][size.of.tree]INT tree :
  SEQ
    initialize.tree(tree)
    more.characters.expected := TRUE
    WHILE more.characters.expected
      ALT
        INT char :
        source ? char
          SEQ
            encode.character(sink, tree, char)
            increase.weight(tree, char)
        end.of.source ? CASE signal
          more.characters.expected := FALSE
    encode.character(sink, tree, end.of.message)
:
```

```
--
--  copy.decoding
--

PROC copy.decoding(CHAN OF BIT source, CHAN OF INT sink)
  --
  --  read a bit stream from source, decoding it into characters and
  --  send these along sink until end.of.message is decoded
  --
  BOOL more.characters.expected :
  [number.of.arrays][size.of.tree]INT tree :
  SEQ
    initialize.tree(tree)
    more.characters.expected := TRUE
    WHILE more.characters.expected
      INT char :
      SEQ
        decode.character(source, tree, char)
        IF
          char <> end.of.message
            SEQ
              sink ! char
              increase.weight(tree, char)
          char = end.of.message
            more.characters.expected := FALSE
:
```

```
--
--  The interfaces of the copy.encoding and copy.decoding procedures
--  in this chapter are the same as those in the preceding chapter,
--  so we could here define
--
--    PROC encode.into.blocks(...)
--    PROC decode.from.blocks(...)
--    PROC copy.over.serial.medium(...)
--    PROC copy.over.blocked.medium(...)
--
--  by repeating the texts of their definitions from that chapter.
--
```