An Efficient Implementation of Huffman Decode Tables

D. R. McIntyre and F. G. Wolff
Cleveland State University & Case western Reserve University

1. Introduction

Suppose a source string (file), S, consists of a set of distinct characters $\Sigma = (a_1, a_2, \ldots, a_n)$ of equal length $\lceil \log_2 N \rceil$ where N is the universe of possible characters. If for $1 \le i \le n$, w_i is the nonnegative frequency of character a_i in string S then the well-known algorithm of Huffman [1] can be used to construct a binary tree with n leaves (external nodes) corresponding to the distinct characters in the file, and n-1 internal nodes. Huffman's tree minimizes $w_1 * l_1 + w_2 * l_2 + \ldots + w_n * l_n$ where l_i is the level at which a_i occurs in the tree.

Corresponding to each Huffman tree on Σ can be associated n binary codes (strings of 0's and 1's) that form a prefix code. A coding of Σ is a prefix code if no code of a character in Σ is the prefix of any other code of a character in Σ. The correspondence between Huffman trees and codes is achieved by representing the path from a root to each leaf as a string of 0's and 1's, where 0 corresponds to a left branch and 1 to a right branch. We see from this that Huffman's algorithm produces a prefix code such that the resulting sequence of 0's and 1's in the encoding of string S has minimum length.

It is clear from the above discussion that the Huffman encoding (and hence the decoding) is very much a function of the source string S being compressed, and hence some representation of the decode table must be included along with the encoded (compressed) string S.

Decode Table (DT) (representation of codes)	Encoded String S (compressed string S)

The decode table must consist of some representation of the codes followed by a list of the characters being encoded. However, since the codes are a repetition of the branches in the Huffman tree, typically the decode table (DT) is more efficiently stored as some representation of the shape of the Huffman tree followed by a list of the characters being encoded (the Huffman tree leaf labeling) as follows:

Huffman Decode Table (DT)

Huffman Tree Shape (TS)	Leaf Labeling (LL)

In this paper we present an extremely efficient storage implementation of the Huffman tree shape (TS) part of the decode table which is useful for efficient disk storage of compressed files. An outline of the paper is as follows. In Section 2 we introduce some historical techniques for storing the Huffman tree shape as a precursor to better understanding our implementation. Section 3 presents our Base 1 method for storing the Huffman tree shape. Our latest method called Base 2 for storing the Huffman trees shape is given in section 4. Finally in Section 5 the results of experimental runs given comparing the various methods discussed are given.

2. Historical Tree Shape Storage Implementations

2.1 Introduction

To begin with we review the Huffman algorithm [1] for constructing the Huffman encode/decode tree for string S containing distinct characters $\Sigma = \{a_1, a_2, \ldots, a_n\}$.

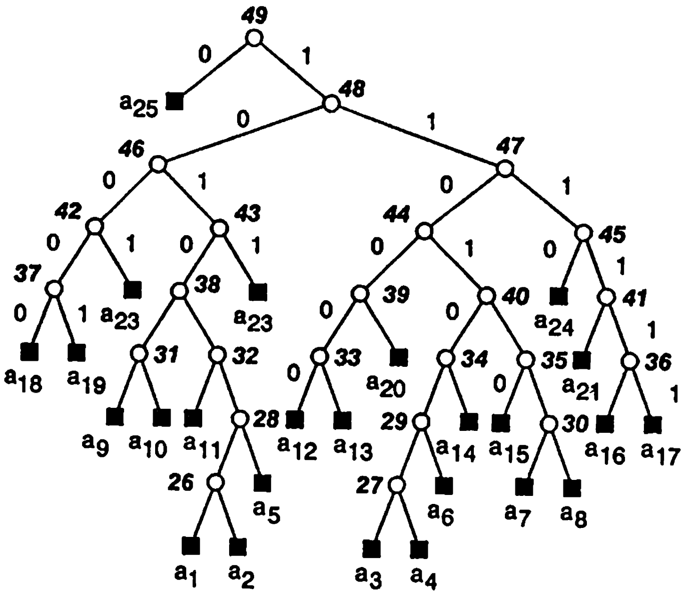
```
0) Scan string S to determine the frequencies, $w_i$, of each
   character $a_i$ in $\Sigma$.
1) Sort $\Sigma$ by frequency and initially let each character $a_i$
   be represented by a one node tree with frequency $w_i$.
   (O(n log n) time complexity)
2) Generate the Huffman tree (O(n) time complexity)
   do n-1 times
      i)  Replace two trees of smallest frequency with a
          new tree
          a) containing a root node with the two smallest
             trees as subtrees, and
          b) with frequency the sum of the frequencies
             of its two subtrees
      ii) Insert the new tree (logically) into the
          sorted list of trees
   enddo
3) Generate the character codes. (O(n) time complexity))
   Arbitrarily assign 0's to the left links and 1's to the
   right links of the Huffman tree. Then scan the Huffman
   tree in NLR order to obtain the prefix codes.
4) Store (O(n) time complexity)
         a) the Huffman tree shape, TS, followed by
         b) the leaf labeling, LL, followed by
         c) the encoding of S
```

For our examples in this section and the two subsequent sections we shall use a source string S containing 25 characters (S was an actual FORTRAN source code file).

2.2 Standard Huffman Implementation

Until recently, most PC archive software stored the Huffman tree shape using an array of records, each record consisting of a LEFT CHILD and RIGHT CHILD fields. Initially the array consists of n elements containing zeros in the left and right child fields corresponding to forest of n trees consisting of single leaf nodes. Then as trees (the two smallest) are combined an additional root node is added to the array at the next available spot with left and right indexes to the roots of the left and right subtrees respectively. We show part of the array generated for the test string S. The array will clearly contain n + (n-1) = 2n-1 elements where the last element added will be the root of the Huffman tree. It should be clear that it is unnecessary to store the first n elements of the array representing leaf nodes (which only contain zeros) as long as the leaf labelings, corresponding to those array elements, are listed in the same order as the first n elements of the array. It should also be clear that the value of n, 25, in this case, must precede the n-1 = 24 array elements storing the tree shape in order to know where the part of the decode table containing the tree shape ends and the leaf labeling begins. Figure 2.2.1 illustrates the example.

For the general case, this method of storing the tree shape requires $\lceil \log_2 N \rceil$ to store n plus $2(n-1)(\lceil \log_2 N \rceil + 1)$ to store the tree.



a_1	0	0
n a_2	0	0
leaves .	.	.
a_{24}	0	0
a_{25}	0	0
a_{26}	1	2
a_{27}	3	4
a_{28}	26	5
a_{29}	27	6
a_{30}	7	8
a_{31}	9	10
	.	.
	.	.
n-1	.	.
interior	.	.
nodes	.	.
	.	.
a_{46}	42	43
a_{47}	44	45
a_{48}	46	47
a_{49}	25	48

Figure 2.2.1 Huffman Tree and Decode Table.

2.3 Schwartz and Kallick Implementation

Schwartz and Kallick in their paper [2] introduce the concept of a canonical tree which leads to a significant reduction in the tree shape storage.

Definition: A Huffman tree is in canonical form if the leaves at each level are left of all internal nodes at that level.

It is easy to convert any Huffman tree, H, into canonical form.

```
procedure canonicalize (H)
  for each level beginning with level 1 do
    while there exists a leaf right of an internal node do
          interchange the leaf with the tree rooted at
          the leftmost internal node at that level
    endwhile
    return (H)
  endfor
endproc
```

The following Figure 2.3.1 illustrates the algorithm for a small Huffman tree.

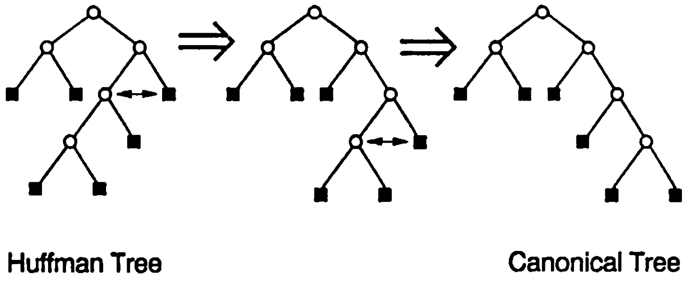
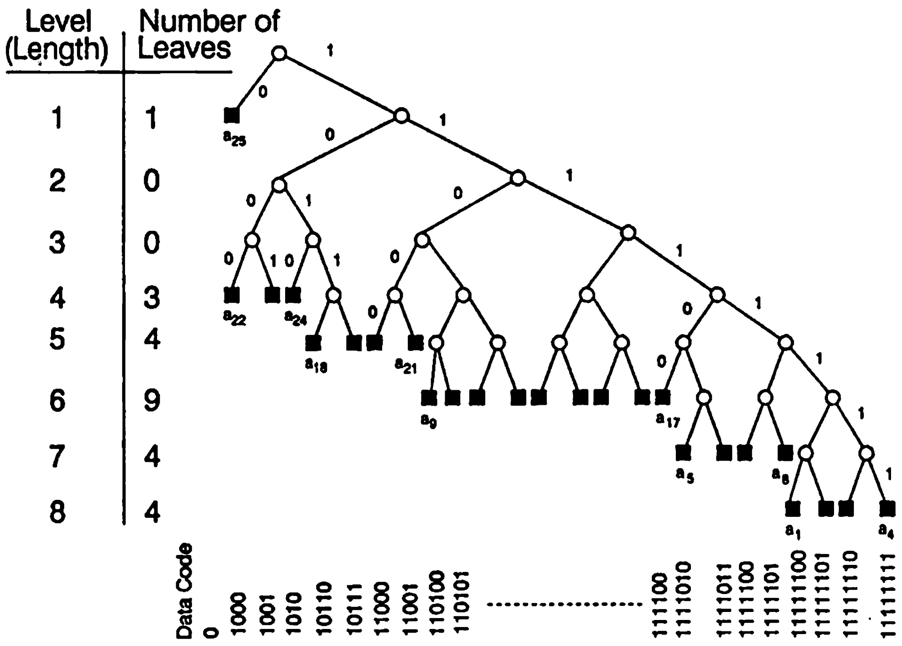


Figure 2.3.1 Converting a Huffman Tree to Canonical Form.

Lemma If H is a Huffman tree for string S and C is the corresponding canonical tree for H then the prefix code produced for C also minimizes the encoding for S.

Proof: Obvious since the level of each leaf in the canonical tree is the same as its level in the Huffman tree. •

Schwartz and Kallick discovered that if the Huffman tree is first converted to its canonical form, then the amount of decode table storage required to store the much simpler tree shape was significantly reduced. The scheme stores the tree shape in a list, TS, of length k where k is the number of levels in the canonical tree - 1 (not including the root level). The ith element in list TS contains the number of leaves at level i. All elements in TS are a fixed number of bits in size. For example Figure 2.3.2 gives the canonical tree, tree shape and leaf labeling.



Tree Shape	Leaf Labeling
1 0 0 3 4 9 4 4	a25 a22 a23 a24 a18 a19 a20

a21 a9-a17 a5-a8 a1-a4	...encoded string...

Figure 2.3.2 The Schwartz & Kallick Canonical Tree, Decode Table and Numerical Sequence Code.

However, the question arises as to how the decoder determines the end of the tree shape section TS without initially storing either k, the number of levels - 1, or n, the number of leaves. It turns out that this is not necessary as the number of nodes in the tree (and hence the end of the tree shape) can be deduced. The basis of this lies in the following observations.

Lemma In any canonical tree if the number of internal nodes at level k is I_k then the total (leaves plus internal) number of nodes at level k+1 is $2I_k$.

Proof: Obvious since in a canonical tree each internal node has exactly 2 child nodes. •

This fact along with the fact that the number of internal nodes at any level is the total number of nodes at that level minus the number of leaves at that level, allows the decoder to be able to reconstruct the shape of the canonical tree from the stored tree shape and also deduce the end of the stored tree shape section within the decode table. Let l_i (the number of leaves at level i) for $1 \le i \le k$ be the elements of the list TS stored for the tree shape. Then the decoder can scan the k elements of TS and determine the structure of the canonical tree as follows.

```
Construct root node at level 0.
Construct 2 child nodes to the root at level 1.
i  := 1      /* i keeps track of the tree level */
$L_i$ := list of nodes from left to right at level i.
while number of nodes at level i = $|L_i| > 0$ do
      1. The first $l_i$ nodes of $L_i$ are leaf nodes.
      2. The remaining $|L_i| - l_i$ nodes of $L_i$ are internal
         nodes and for each internal node construct two
         child nodes and place them in $L_{i+1}$.
      3. i := i + 1
endwhile
```

The decoder can now continue to scan the remaining part of the decode table containing the n (the decoder has determined the number of leaves, n, from the tree generation) leaf labeling character codes and associate the leaves of the canonical tree, level by level from the top of the tree to the bottom level of the tree and left to right within each level, with the leaf labeling character codes in the order they are encountered in the decode table.

In addition, Schwartz and Kallick discovered an interesting numerical sequence property for the codes of leaves in a canonical tree which makes the process of generating at encode time the codes for the characters $\Sigma = \{a_1, a_2, \ldots, a_n\}$ even easier. It takes advantage of the fact that the explicit structure of the canonical tree is not required but only the number of bits (i.e. level) in the code for each leaf and left to right order within level of each of the character codes of the leaves in the tree (this is the leaf labeling list (LL)). This is exactly what Schwartz and Kallick store in the decode table (DT), namely TS followed by LL. The following procedure is used by the encoder to generate the codes $C = c_1, c_2, \ldots, c_n$ corresponding to the elements in LL using $TS = l_1, l_2, \ldots, l_k$ where l_i is the number of leaves at level i in the cannonical tree.

```
procedure code_generation (DT, C);
    /* DT is the decode table and consists of the tree       */
    /* shape section, TS, followed by the leaf labeling      */
    /* section LL which is some permutation of               */
    /* Σ = (a_1, a_2,..., a_n).                              */
    /* TS = l_1, l_2,...  l_k where l_i is the number of     */
    /*         leaves at level i in the canonical tree       */
    /* This procedure uses DT to directly generate C.        */
    /* C = c_1, c_2,.., c_n where c_i is the code of ith leaf */
    /* in the leaf labeling list (LL)                        */
  begin
  i := 1;     /* i is the level of the canonical tree        */
  T_i:= 2;    /* T_i is the total number of nodes at level i */
  k := 0;     /* c_k is the code of the kth leaf label in LL */
  bit_code := 0 bit
  while number of nodes, T_i, at level i not 0 do
       1. for j  := 1 to l_i do
               k  := k + 1;
               c_k := bit_code;
               bit_code := bit_code + 1; /* binary add */
          endfor
          bit_code := bit_code with 0 bit appended on right
       2. T_{i+1} := (T_i - l_i) * 2;
       3. i := i + 1;
  endwhile
  end
endproc
```

Figure 2.3.2 illustrates the use of the numerical sequence property to generate codes.

In general the storage required for the tree shape in a tree with n leaves, k levels, and universal set of N characters is $k\lceil \log_2 N \rceil$ bits. Typically, (on average) k is of order $\log_2 N$ so this represents better than half saving over the standard Huffman trees storage implementation.

3. Base 1 Implementation

The motivation for our first improvement in tree shape storage was to avoid the rather large fixed length fields of size $\lceil \log_2 N \rceil$ bits to store the number of leaves at each level of the canonical tree. The scheme to generate the encode/decode trees works as follows. Use a 0 bit to represent each leaf node and a 1 bit to represent each single downward level change. Then beginning with level 1 in the canonical tree (again trees with level 0 which consist of a single leaf, the root, are not practically interesting), if there are any leaves at this level store a 0 for each occurrence. Then store a 1 to indicate a single downward level change. Continue this process at each subsequent level. The bottom level will store a 0 bit for each leaf occurrence (all nodes at this level will be leaves) however, no 1 bit will be stored since there is no subsequent lower level. Figure 3.1 illustrates the situation for our example string S. The base 1 decode table (DT) can be generated directly from the initial (noncanonical) Huffman tree, H, by the encoder without explicitly generating the canonical tree as follows.

```
procedure Decode_Table_Generation_1 (H,DT)
   /* H  is the initial Huffman tree with k levels.        */
   /* DT is the decode table consisting of TS followed     */
   /* by LL.                                               */
   /* TS is the Tree Shape bit string, and                 */
   /* LL is the leaf labeling character codes.             */
   /* This procedure generates DT from H.                  */
   begin
     /*  perform a breadth first search of H               */
     TS := [];
     LL := [];
     for i := 1 to k /* number of levels in H */ do
         if there are $l_i$ leaves at level i then
            TS := TS with $l_i$ 0's appended to the right;
            LL := LL with the character codes of the $l_i$
                       leaves in the left to right order within
                       the tree at level i appended to the
                       right;
         endif
         if i <> k then
            TS := TS with 1 appended to the right;
         endif
     endfor
     DT = TS with LL appended to the right
   end
endproc
```

The reason for the name "base 1" is simply because the number of leaves at each level is represented as a base 1 string of 0's. Also, base 1 strings of 1's are used to represent changes in level.

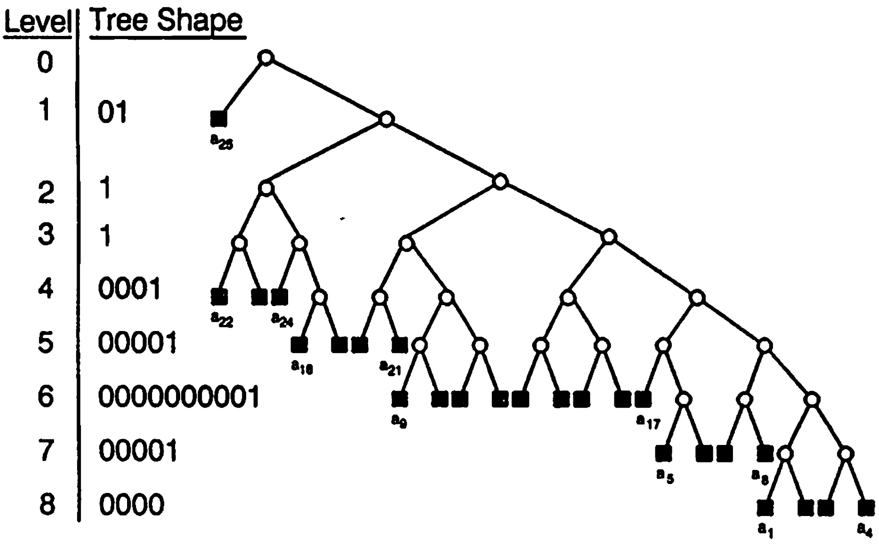
The following procedure is used by the decoder to scan the decode table (DT) and regenerate the canonical Huffman encode/decode tree, H, with labeled leaves. The decode table (DT) is a bit string which contains both the Huffman tree shape (TS) and the fixed length leaf labeling (LL) codes, a permutation of character codes a_1, a_2, ..., a_n. The procedure used is similar to the procedure used in Section 2.3 except that it deduces the number of leaves at each level by counting the number of 0's in contiguous strings of 0's in TS. We assume that each node of the decode tree generated contains (at least) the pointer fields (denoted by ^) left and right which point to the left child and right child respectively.

```
procedure Decode_Tree_Generation_1 (DT,H);
 begin
 /* This section scans the first part of the decode table */
 /* (DT) containing the Huffman tree shape (TS) and from  */
 /* it reconstructs the shape of the Huffman tree H.      */
 i := 1           /* i is the level of the canonical tree */
 Allocate storage for nodes $n_0$, $n_1$ and $n_2$.
 k := 3           /* number of nodes allocated            */
 left ( $n_0$ ) := ^$n_1$
 right( $n_0$ ) := ^$n_2$
 H  :=  ^$n_0$    /* H points to the root of the decode tree */
 $L_i$ := [^$n_1$, ^$n_2$] /* $L_i$ is a list of pointers to nodes in */
                  /* left to right order on the ith level */
                  /* of the Huffman decode tree being     */
                  /* reconstructed.                       */
 LEAVES := [] /* LEAVES will contain a list of pointer to */
              /* leaves in the decode tree stored in     */
              /* such a way that leaves left of other    */
              /* leaves in the tree appear before other  */
              /* elements in the list LEAVES.            */
 while number of nodes at level i = $|L_i|$ > 0 do
       1. Set $L_{i+1}$ = the empty list []
       2. Scan $l_i$ consecutive 0 bits (but no more than $|L_i|$)
            beginning with the next bit in the Huffman tree
            shape (TS).  Also if less than $|L_i|$ zero bits
            were scanned then the subsequent 1 bit is
            scanned ($l_i$ could have value 0 if the next scan
            bit is 1).
       3. Scan the first $l_i$ elements of $L_i$ copying the
          pointers to these nodes to the end of list LEAVES.
       4. Scan the remaining $|L_i|$ - $l_i$ elements of $L_i$ :
          for j = $l_i$ + 1 to $|L_i|$ do
              Allocate storage for nodes $n_k$ and $n_{k+1}$
              left ( jth element of $L_i$ ) := ^$n_k$
              right( jth element of $L_i$ ) := ^$n_{k+1}$
              Add ^$n_k$ and ^$n_{k+1}$ to the end of $L_{i+1}$
              k := k + 2
          endfor
       5. i := i + 1;
 endwhile


 /* This section continues the scan of the DT which      */
 /* contains the fixed length leaf labelling (LL) codes  */
 /* and associates each element of LL with the           */
 /* corresponding element in list LEAVES respectively.   */

 for i := 1 to n do
     Associate label tree leaf pointed to by ith element of
        LEAVES with ith element of LL.
 endfor
 end
endproc
```

The following summarizes the storage costs of storing the tree structure using the base 1 method.

Lemma 3.1 If a canonical tree has n leaves and k levels then the size of the tree structure (TS above) is n + k - 1 bits.

Proof: There is a 0 bit for each leaf and there are n leaves. Also there is a 1 bit for each level below level 1, and since there are k levels there are k-1 1 bits. •



Tree Shape (32 bits) | Leaf Labeling

01110001000010000000001000010000	a25 a22-a24
a18-a21 a9-a17 a5-a8 a1-a4	...encoded string...

Figure 3.1 Base 1 Canonical Tree and Decode Table

4. Base 2 Implementation

The use of base 1 strings consisting of r 0's to represent r leaves at some level in the base 1 tree shape storage method is somewhat wasteful of storage. Hence our attention was directed to searching for a base 2 way of encoding the number of leaves at each level. The idea was that if the canonical tree had k levels then we would store k base 2 binary numbers l_i, $1 \le i \le k$ where l_i represents the number of leaves at level i. This lead to the problem of determining what size field should be used. If a fixed field size is used then the storage method used in Section 2.3 is the result with a large field size of $\lceil \log_2 N \rceil$ bits (to handle the worst case).

The solution is to use variable size fields but in such a way that the decoder at decode time could determine the variable field sizes that were used by the encoder. The solution can be seen by a careful examination of the algorithms Decode_Table_Generation_1 and Decode_Tree_Generation_1 in Section 3. In these algorithms both the encoder and decoder know the total number of nodes, T_i say, at the next level, i, of the canonical tree ($|L_i|$ in the algorithms). Thus at level i in the canonical tree, if T_i is the total number of nodes at this level then l_i, the number of leaves, is stored base 2 using $\lceil \log_2 T_i \rceil$ bits.

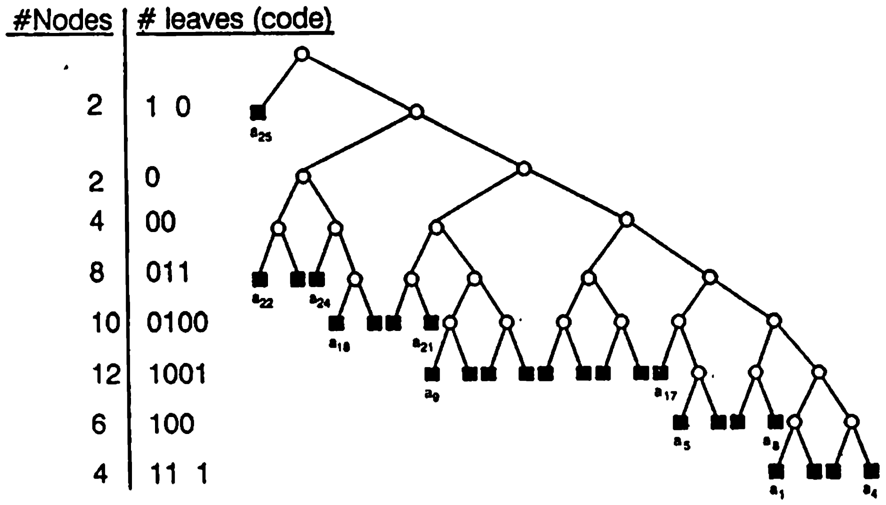
This, however, leads to a further problem. At any given level, i say, containing T_i nodes it is possible for level i to contain l_i leaves where $0 \le l_i \le T_i$. If $T_i = 2^r$ for some integer r then $\lceil \log_2 T_i \rceil = r$ bits will not suffice to store the $T_i+1 = 2^r+1$ values that l_i can assume. The solution to this situation is to let a string of r 1's represent two possibilities. These two cases are resolved by examining the next (r+1 st) bit: r 1's followed by a 1 will denote $l_i = T_i = 2^r$ leaves, whereas r 1's followed by a 0 will denote $l_i = T_i - 1 = 2^r-1$ leaves. Figure 4.1 illustrates the method.

The base 2 method to generate the decode table (DT) is similar to the base 1 method and again can be generated directly from the initial (noncanonical) Huffman tree, H, by the encoder without explicitly generating the canonical tree as follows.

```
procedure Decode_Table_Generation_2 (H,DT)
   /* H  is the initial Huffman tree with k levels.        */
   /* DT is the decode table consisting of TS followed     */
   /* by LL.                                                */
   /* TS is the Tree Shape bit string, and                  */
   /* LL is the leaf labeling character codes.              */
   /* This procedure generates DT from H.                   */
   begin
     /*  perform a breadth first search of H                */
     TS := [];
     LL := [];
     for i := 1 to k /* number of levels in H */ do
         if $l_i$ leaves at level i then
            TS := TS with base 2 encoding of $l_i$ appended to
                     the right;
            LL := LL with the character codes of the $l_i$
                     leaves in the left to right order within
                     the tree at level i appended to the
                     right;
         endif
     endfor
     DT = TS with LL appended to the right
   end
endproc
```

The following procedure is used by the decoder to scan the decode table (DT) and regenerate the canonical Huffman encode/decode tree, H, with labeled leaves. The decode table (DT) is a bit string which contains both the Huffman tree shape (TS) and the fixed length leaf labeling (LL) codes, a permutation of character codes a_1, a_2, ..., a_n. The procedure is similar to the base 1 procedure used in Section 3 except that it deduces the number of leaves at each level by decoding the base 2 value of the number of leaves in TS. Again we assume that each node of the decode tree generated contains (at least) the pointer fields left and right which point to the left child and right child respectively.

```
procedure Decode_Tree_Generation_2 (DT,H);
  begin
  /* This section scans the first part of the decode table */
  /* (DT) containing the Huffman tree shape (TS) and from  */
  /* it reconstructs the Huffman tree shape.               */
  i := 1           /* i is the level of the canonical tree */
  Allocate storage for nodes $n_0$, $n_1$ and $n_2$.
  k := 3           /* number of nodes allocated            */
  left ( $n_0$ ) := ^$n_1$
  right( $n_0$ ) := ^$n_2$
  H  :=  ^$n_0$    /* H points to the root of the decode tree */
  $L_i$ := [^$n_1$, ^$n_2$] /* $L_i$ is a list of pointers to nodes in */
                   /* left to right order on the ith level */
                   /* of the Huffman decode tree being     */
                   /* reconstructed.                       */
  LEAVES := [] /* LEAVES will contain a list of pointer to */
               /* leaves in the decode tree stored in      */
               /* such a way that leaves left of other     */
               /* leaves in the tree appear before other   */
               /* elements in the list LEAVES.             */
  while number of nodes at level i = $|L_i|$ > 0 do
        1. Set $L_{i+1}$ = the empty list []

        2. $l_i$ := value of next $\log_2|L_i|$ bits of TS (if all
                 the bits are 1's then if the next bit is 0
                 the value is $|L_i|$- 1 otherwise the value is
                 $|L_i|$)

        3. Scan the first $l_i$ elements of $L_i$ copying the
           pointers to these nodes to the end of list LEAVES.

        4. Scan the remaining $|L_i|$ - $l_i$ elements of $L_i$ :
           for j = $l_i$ + 1 to $|L_i|$ do
               Allocate storage for nodes $n_k$ and $n_{k+1}$
               left ( jth element of $L_i$ ) := ^$n_k$
               right( jth element of $L_i$ ) := ^$n_{k+1}$
               Add ^$n_k$ and ^$n_{k+1}$ to the end of $L_{i+1}$
               k := k + 2
           endfor

        5. i := i + 1;
  endwhile
  /* This section continues the scan of the DT which         */
  /* contains the fixed length leaf labelling (LL) codes     */
  /* and associates each element of LL with the              */
  /* corresponding element in list LEAVES respectively.      */

  for i := 1 to n do
      Associate label tree leaf pointed to by ith element of
         LEAVES with ith element of LL.
  endfor
  end
endproc
```



Tree Shape (22 bits)	Leaf Labeling
1000001101001001100111	a25 a22-a24
a18-a21 a9-a17 a5-a8 a1-a4	...encoded string...

Figure 4.1 Base 2 Canonical Tree and Decode Table.

The storage costs of storing the tree structure using the base 2 method is harder to analyze. However experimentally the results show a significant decrease in the tree shape storage size of almost 50% over base 1 storage.

5. Experiments

Compression experiments were conducted using 485 files consisting of PL/I, COBOL, FORTRAN and Pascal source programs over the universe of N = 256 characters (i.e. 8 bit codes) for the implementations of Huffman, Schwartz & Kallick, base 1, and base 2 discussed in this paper. For each implementation, the number of bits to store the tree shape divided by the number of leaves was computed per file and then averaged over all files. The results in Figure 5.1 show that base 2 is about twice as efficient as base 1 which in turn is about 6 times as efficient as Schwartz & Kallick which in turn is about twice as efficient as Huffman. The base 2 size of the tree shape (TS) is 22.5 times smaller than standard Huffman methods.

	# bits in Tree Shape / # leaves
Huffman	16.9
Schwartz & Kallick	7.9
Base 1	1.3
Base 2	0.75

Figure 5.1 Comparison of Number of bits to Store the Tree Shape / Number of Leaves averaged over 485 files.

REFERENCES

[1] D. A. Huffman, "A method for the construction of minimum redundancy codes", Proc. IRE, 40, 9, (1952), pp. 1098-1101.

[2] E. S. Schwartz, and B. Kallick, "Generating a canonical prefix encoding" Commun. ACM 7, 3 (1964), pp. 166-169.