Report Number: WUCS-81-05

1981-08-01

Number of Binary Trees

Authors: Will D. Gillett

A data encoding scheme involving binary tree encodements is presented and analyzed. A closed-form formula for the number of n-bit legal memory configurations is developed. It is shown that the storage capacity loss due the use of this scheme is not significant for large n.

Number of Binary Trees

Will Gillett

WUCS-81-05

August 1981

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

This work was supported in part under the NCHSR Grant HSO3792 and the NIH under Grant RR00396.

Number of Binary Trees

ABSTRACT

A data encoding scheme involving binary tree encodements is presented and analyzed. A closed-form formula for the number of n-bit legal memory configurations is developed. It is shown that the storage capacity loss due to the use of this scheme is not significant for large n.

1. Introduction

There are many numeric and non-numeric data encoding schemes in current use. These include BCD, EBCDIC, ASCII, Huffman codes (variable length, uniquely decipherable codes), Hamming codes (error detecting and correcting codes), standard binary, 1's complement, and 2's complement. We propose another variable length system, one based on binary trees.

Our proposed scheme is motivated by an information data model developed at Washington University. The Abstract Database System (ADS) [COX80; KIMU81a; KIMU81b] is a self-referencing data model that can be used to define, analyze and implement information retrieval systems. The definition of ADS uses binary trees for representing (1) data, (2) the schema of data, and (3) the command language itself. Since the binary tree concept is such an intrinsic part of ADS, the use of binary trees encodements may be the most appropriate data encoding scheme for the implementation of systems based on ADS.

Our binary tree encoding scheme has certain advantages over the standard binary encoding (SBE) scheme (i.e., standard binary numbers in which 2^n different values can be held in n bits). Variable length encodements are self-delimiting; thus, no explicit delimiters and/or pointers are necessary. The use of binary trees as the basic building block for command, schema, and

data languages allows a uniform interpretation across all these languages. The algorithms for building, decomposing, delimiting, and manipulating the encodements are simple and have multiple implementations. The scheme is also compatible with a customized associative memory system we are developing. However, there are two major disadvantages: the encoding efficiency is less than that of the SBE scheme (although we will show that this is not significant for large n), and the scheme is not efficiently implemented in word-oriented architectures.

In the past, SBE schemes have been selected as being particularly suitable for von Neumann architectures. However, as VLSI becomes more available and custom LSI becomes a reality, non-von Neumann architectures become more realizable and available. Current research and development objectives in computer science involve the definition and realization of distributed, reliable, expandable, and efficient systems. Choices other than those historically selected for von Neumann architectures may be more appropriate or required, in the light of these objectives. This paper reports research in this direction.

Section 2 defines the binary tree encoding (BTE) scheme and the concept of a legal memory configuration. Section 3 presents a summary of the important results. Sections 4, 5, 6, and 7 present four different methods for computing the number of n-bit legal memory configurations: an exhaustive enumerations solution, a combinatorial solution, a recurrence relation solution, and a generating function solution. (Each of these has been programmed, and all yield identical results.) Section 8 compares the storage capacity loss between our BTE scheme and the SBE scheme, and Section 9 presents our conclusions.

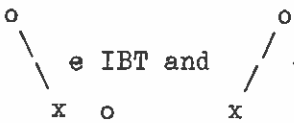
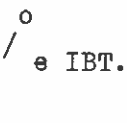
2. The Binary Tree Encoding Scheme

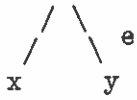
First, we describe our meaning of binary tree. There are many different definitions or restrictions of binary tree. We define two here: incomplete binary trees and complete binary trees. Although we are primarily interested in the second, the first is useful as an intermediate vehicle in the analysis development. In the definitions, we use the symbol 'o' to represent an internal node (i.e., a node that has a son) and 'N' to represent a leaf node (i.e., a nil node, or a node that has no sons). There are no intrinsic data held in any internal or leaf node.

Definition 1

Let IBT be the set of incomplete binary trees. Then,

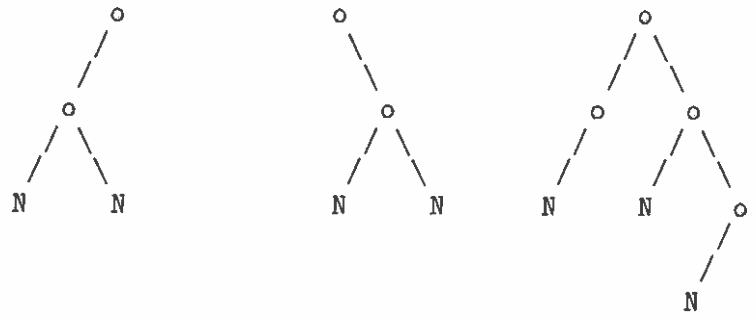
1) N ∈ IBT.

2) If x ∈ IBT, then  ∈ IBT and  ∈ IBT.

3) If x, y ∈ IBT, then  ∈ IBT.

4) Nothing else is in IBT.

Example 1: The following are all different incomplete binary trees:



Definition 1 is meant to imply that each internal node has a distinguishable left son and right son. An internal node may have either a right son or a left son or both. The number of n-node incomplete binary trees is given by:

$$T_n = \frac{1}{n+1} C_n^{2n} \quad . \tag{1}$$

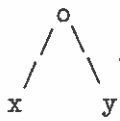
These are known as the Catalan numbers [KNUT73, pp. 388-389; EVEN79, pp. 82-84]. Here, C_k^i is a notation for the combination of i things taken k at a time and is computable by the formula:

$$C_k^i = \frac{i!}{k!(i-k)!} \quad . \tag{2}$$

Definition 2

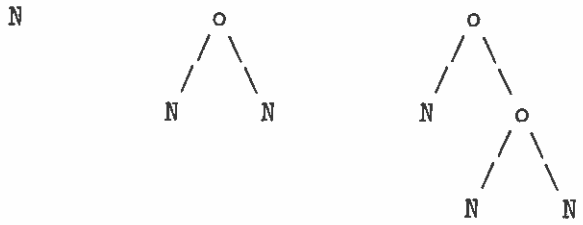
Let CBT be the set of complete binary trees. Then,

1) N e CBT.

2) If x, y e CBT, then  e CBT.

3) Nothing else is in CBT.

Example 2: The following are all complete binary trees:



None of the trees in Example 1 are complete binary trees, although CBT is a subset of IBT.

As before each internal node has a distinguishable left son and right son. However, each internal node must have both a left son and right son (hence, the adjective "complete"). In the remainder of this paper, the term "binary tree" means "complete binary tree" unless otherwise stated (i.e., we drop the adjective "complete" for brevity).

Some pertinent facts about binary trees are:

1) In every binary tree, the number of leaf nodes is one greater than the number of internal nodes.

2) Every binary tree has an odd number of nodes (derivable from fact 1).

3) The number of n-node binary trees, S_n, is given by:

$$S_n = \frac{2}{n+1} C^{n-1}_{\frac{n-1}{2}} \qquad \text{for n odd} \qquad (3)$$

$$= 0 \qquad \text{for n even}$$

In fact, $S_n = T_{\frac{n-1}{2}}$ for n odd [EVEN79, pp. 82-87].

The values of T_n and S_n over a small range of n is presented in Table 1.

Table 1
Values of T_n and S_n

n	T_n	S_n
1	1.000	1.000
2	2.000	0.000
3	5.000	1.000
4	1.400×10^1	0.000
5	4.200×10^1	2.000
6	1.320×10^2	0.000
7	4.290×10^2	5.000
8	1.430×10^3	0.000
9	4.862×10^3	1.400×10^1
10	1.680×10^4	0.000
11	5.879×10^4	4.200×10^1
12	2.080×10^5	0.000
13	7.429×10^5	1.320×10^2
14	2.674×10^6	0.000
15	9.695×10^6	4.290×10^2
16	3.536×10^7	0.000
17	1.296×10^8	1.430×10^3
18	4.776×10^8	0.000
19	1.767×10^9	4.862×10^3
20	6.564×10^9	0.000
21	2.447×10^{10}	1.680×10^4
22	9.148×10^{10}	0.000
23	3.431×10^{11}	5.879×10^4
24	1.290×10^{12}	0.000
25	4.862×10^{12}	2.080×10^5

Next, we describe the encodement of binary trees into their bitstring counterparts. The bitstring encodement of a binary tree corresponds to a preorder traversal [KNUT73, page 344] of the binary tree where a '1' is written for each internal node, and a '0' is written for each leaf node. This description is formalized by the following definition.

Definition 3

Let BTE be the set of binary tree encodements. Then,

1) '0' ∈ BTE.
2) If x, y ∈ BTE, then '1' || x || y ∈ BTE (here '||' means concatenation).
3) Nothing else is in BTE.

Example 3: The binary tree encodements for the three binary trees in Example 2 are the bit strings "0", "100", and "10100".

Notice that all bit strings do not correspond to binary tree encodements. For instance, "010" and "1010" do not meet the definition. Note that, except for the nil tree, every binary tree encodement ends with two '0's, begins with a '1', and contains one more '0' than it does '1's. Also note the close correspondence between Definition 2 and Definition 3. In fact, they are simple different ways of stating the same conceptual formalisms; every binary tree corresponds to one and only one binary tree encodement, and visa versa.

Definition 4

Let $BTE_n = \{x \in BTE \mid length(x) = n\}$.

Because of the correspondence between binary trees and binary tree encodements, the cardinality of BTE_n is also given by equation (3).

Last, we describe the legal memory configurations; these are just concatenations of binary tree encodements.

Definition 5

Let LMC be the set of legal memory configurations. Then,

1) If x e BTE, then x e LMC.

2) If x, y e LMC, then x || y e LMC.

3) Nothing else is in LMC.

Example 4: The bit string "010010100" is a legal memory configuration; it is the concatenation of the three binary tree encodements shown in Example 3 and logically represents the set of three corresponding binary trees shown in Example 2. The bit string "101010" is not a legal memory configuration because it is not the concatenation of any set of binary tree encodements.

Definition 6

Let $LMC_n = \{x \in LMC \mid length(x) = n\}$.

Our problem now reduces to determining the cardinality of LMC_n, i.e., $card(LMC_n)$.

3. The Result

Two major results are presented here prior to the detailed analysis.

Definition 7

Let $d_n = card(LMC_n)$, i.e., the number of n-bit legal memory configurations.

The first result is:

$$d_n = 0 \qquad \text{for } n = 0 \tag{4}$$

$$= \frac{1}{2} C^n_{\frac{n}{2}} \qquad \text{for } n > 0 \text{ and even}$$

$$= C^{n-1}_{\frac{n-1}{2}} \qquad \text{for } n > 0 \text{ and odd.}$$

The second result deals with the storage efficiency. The values of d_n are monotonically increasing; thus, it clearly requires n bits to hold d_n different encodements in the BTE scheme. However, it only requires $\log_2(d_n)$ bits to hold the same number of encodements in the SBE scheme. The difference in the number of bits required to hold the same number of encodements using the different schemes supplies a measure of the encoding efficiency.

Definition 8

Let the absolute storage capacity loss, a_n, be defined as:

$$a_n = n - \log(d_n) \quad .$$

Let the relative storage capacity loss, r_n, be defined as:

$$r_n = \frac{a_n}{n} = 1 - \frac{\log(d_n)}{n} \quad .$$

Then, by applying Stirling's approximation, it can be shown that:

$$L_n = 0.71 + \frac{1}{2}\log(n) <= a_n <= 1.77 + \frac{1}{2}\log(n) = H_n \quad . \tag{5}$$

The values of d_n, $\log(d_n)$, a_n, r_n, and H_n are presented in Table 2 for selected values of n. Clearly, the storage capacity loss is not significant for large n.

Table 2
Selected values of
d_n, $\log(d_n)$, a_n, r_n, and H_n

n	d_n	$\log(d_n)$	a_n	r_n	H_n
1	1.000	0.000	1.000	1.000	1.770
2	1.000	0.000	2.000	1.000	2.270
3	2.000	1.000	2.000	0.667	2.562
4	3.000	1.585	2.415	0.604	2.770
5	6.000	2.585	2.415	0.483	2.931
6	1.000×10^{1}	3.322	2.678	0.446	3.062
7	2.000×10^{1}	4.322	2.678	0.383	3.174
8	3.500×10^{1}	5.129	2.871	0.359	3.270
9	7.000×10^{1}	6.129	2.871	0.320	3.355
10	1.260×10^{2}	6.977	3.023	0.302	3.431
20	9.238×10^{4}	16.495	3.505	0.175	3.931
30	7.756×10^{7}	26.209	3.791	0.126	4.223
40	6.892×10^{10}	36.004	3.996	0.100	4.431
50	6.321×10^{13}	45.845	4.155	0.083	4.592
60	5.913×10^{16}	55.715	4.285	0.071	4.723
70	5.609×10^{19}	65.604	4.396	0.063	4.835
80	5.375×10^{22}	75.509	4.491	0.056	4.931
90	5.191×10^{25}	85.424	4.576	0.051	5.016
10^2	5.045×10^{28}	95.349	4.651	0.047	5.092
10^3	---	---	---	---	6.753
10^6	---	---	---	---	11.736
10^9	---	---	---	---	16.719

4. An Exhaustive Enumeration Solution

This approach entails the enumeration of every possible n-bit configuration to determine which are in IMC_n. We have used it to verify the correctness of other, more instructive approaches.

The algorithm (bin_encode) shown in Figure 1 takes the n-bit string 'string' and scans it, from left to right, starting at position 'pos'. If a binary tree encodement is found within the n bits, it leaves the variable

'pos' pointing at the next (potential) binary tree encodement and returns 'no error' in 'e_flag'; otherwise, since a complete binary tree encodement is not present in the remainder of the n bits, it returns 'error' in 'e_flag'. This is the basic algorithm for decoding (or delimiting the extent of) a binary tree encodement.

```
bin_encode(string,pos,n,e_flag)
     count <- 1;
     while pos <= n do
          j <- position 'pos' of 'string';
          if j = '1' then
               count <- count + 1;
          else  /*  j = '0'  */
               count <- count - 1;
          endif;
          pos <- pos + 1;
          if count = 0 then
               e_flag <- 'no error';
               return;
          endif;
     endwhile;
     e_flag <- 'error';
     return;
```

Figure 1: Delimit Binary Tree

The algorithm (lmc) shown in Figure 2 determines whether an n-bit string is a legal memory configuration. The variable 'string' contains the memory configuration to be tested, 'n' contains the number of bits, and 'e_flag' indicates whether 'string' contains a legal memory configuration.

```
lmc(string,n,e_flag)
     pos <- 1;
     while pos <= n do
          invoke bin_encode(string,pos,n,e_flag);
          if e_flag = 'error' then
               return;
          endif;
     endwhile;
     e_flag <- 'no error';
     return;
```

Figure 2: Determine Legal Memory Configuration

The algorithm shown in Figure 3 determines the number of n-bit legal memory configurations, i.e., card(LMC_n). It uses the fact that integer values are held in a specific manner.

```
number <- 0;
for i <- 0 to 2^n - 1 do
     string <- last n bits of i;
     invoke lmc(string,n,e_flag);
     if e_flag = 'no error' then
          number <- number + 1;
     endif;
endfor;
output number;
```

Figure 3: Determine Number of Legal Memory Configurations

It should be clear that, while this type of algorithm produces the correct results, it is very time consuming. It is not instructive in providing insight into the general nature of the number of n-bit legal memory configurations. There are also implementation problems (although they can be resolved) as n exceeds the number of bits in the word size of the machine upon

which the algorithm executes.

5. A Combinatorial Solution

Consider the stylized memory configuration shown in Figure 4. Each set of square brackets ([]) represents a specific binary tree encodement. The symbol '¦' represents a partitioning of the encodements into different regions of the n total bits available. The number below a partition indicates that each binary tree encodement in the partition has exactly n_i bits; the number above indicates that there are m_i such (n_i-bit) encodements in the specific partition. For our purposes, we will assume that $n_1 < n_2 < \ldots < n_k$. This memory configuration is intended to represent an arbitrary member of IMC_n in which the separate binary tree encodements have been moved to produce the partition described above.

$$\begin{array}{ccccccccc} & m_1 & & m_2 & & \cdots & & m_k & \\ ([] & \ldots & [] & ¦ [] & \ldots & [] & ¦ \ldots ¦ [] & \ldots & []) \\ & n_1 & & n_2 & & \cdots & & n_k & \end{array}$$

Total of n bits, i.e.,

$$\sum_{i=1}^{k} n_i * m_i = n$$

Figure 4: A Specific Legal Memory Configuration

The first step is to determine $A_{n,\{n_i\},\{m_i\}}$, the number of strings in IMC_n that have exactly this form (where the n_i and m_i are fixed for the moment). Since each binary tree encodement is completely independent of all others, $A_{n,\{n_i\},\{m_i\}}$ is the product of all the S_i (see equation (3)) where i ranges over the number of bits in each binary tree encodement; that is,

$$A_{n,\{n_i\},\{m_i\}} = \prod_{j=1}^{k} (S_{n_j})^{m_j} \quad . \tag{6}$$

The next step is to determine $B_{n,\{n_i\},\{m_i\}}$, the number of legal memory configurations when the partitioning restriction is relaxed (but the n_i and m_i are still fixed); i.e., any one of the binary tree encodements can be placed <u>anywhere</u> (in relation to the other binary tree encodements) in the available n bits. A straightforward combinatorial argument shows that for every legal memory configuration when the partitioning restriction is in force, there are $C^{m}_{m_1,m_2,\ldots,m_k}$ legal memory configurations when the partitioning restriction is relaxed. Here,

$$m = \sum_{i=1}^{k} m_i$$

and the C notation represents the generalized multinomial coefficient. Thus, there are

$$B_{n,\{n_i\},\{m_i\}} = A_{n,\{n_i\},\{m_i\}} * C^{m}_{m_1,m_2,\ldots,m_k} \tag{7}$$

legal memory configurations when the partitioning restriction is relaxed.

The third and last step is to enumerate all partitions as n_i and m_i are allowed to range over all possible combinations that produce members of LMC_n. The partitions are those which satisfy the following conditions.

$PARTITIONS_n$:

1) All the n_i are positive odd integers.

2) All the m_i are positive integers.

3) $n_1 < n_2 < \ldots < n_k$.

4) $\sum_{i=1}^{k} n_i * m_i = n$.

Thus, d_n, the cardinality of LMC_n, is just the sum of $B_{n,\{n_i\},\{m_i\}}$ over all these partitions, or

$$d_n = \sum_{(PARTITIONS_n)} \left(\prod_{j=1}^{k} (S_{n_j})^{m_j}\right) * C^{m}_{m_1,m_2,\ldots,m_k} \quad . \tag{8}$$

Theorem 1

The cardinality of LMC_n is given by equation (8).

Proof:

The proof is clear from the above discussion.

Given a technique for enumerating all of these partitions without duplication, the value of d_n can be determined by summing over the corresponding Bs. A simple recursive procedure for enumerating these partitions is shown in Figure 5. It is originally invoked by the statement: build_part(1,0,1). Two global arrays, N[1:n] and M[1:n], are assumed. The algorithm evaluate(k) (no shown here) evaluates the corresponding value of B using the values stored in these arrays, and sums it into an accumulator (initially set to zero).

```
build_part(n_start,sum_in,k)
     for n_k <- n_start step 2 to n do
          N[k] <- n_k;
          for m_k <- 1 to n do
               M[k] <- m_k;
               sum_out <- sum_in + n_k*m_k;
               case sum_out of
               =n:  invoke evaluate(k);
                    exit;  /*  inner for loop  */
               >n:  exit;  /*  inner for loop  */
               <n:  build_part(n_k+2,sum_out,k+1);
               endcase;
          endfor;
     endfor;
```

Figure 5: A Recursive Partitioning Algorithm

6. A Recurrence Relation Solution

The analysis in the preceding section is enlightening; however, the algorithm is again rather time consuming and does not give any significant insight into how d_n increases as n increases.

However, in analyzing the values of d_n in Table 2 (the result of applying the algorithm in Figure 5) an interesting pattern emerges. Notice that, as n increases from an even value to an odd value, d_n exactly doubles; as n increases from an odd value to an even value, d_n almost doubles. Although it is difficult to see that this pattern continues (because Table 2 is sparse), this observation does hold for all the values in the sequence. In fact, it can be shown that the amount by which d_n fails to double (as n increases by 1) is exactly S_{n-1}. This can be captured in the form of a recurrence relation [BRUA79; KNUT73; LUEK80]. It is given by:

$$d_n = 2d_{n-1} - S_{n-1} \quad \text{for } n > 1 \qquad (9)$$
$$= 1 \quad \text{for } n = 1$$
$$= 0 \quad \text{for } n = 0$$

A simple expansion procedure applied to this recurrence relation yields:

$$\begin{aligned} d_n &= 2d_{n-1} - S_{n-1} \\ &= 2(2d_{n-2} - S_{n-2}) - S_{n-1} \\ &= 2^2 d_{n-2} - 2S_{n-2} - S_{n-1} \\ &= 2^2(2d_{n-3} - S_{n-3}) - 2S_{n-2} - S_{n-1} \\ &= 2^3 d_{n-3} - 2^2 S_{n-3} - 2S_{n-2} - S_{n-1} \\ &\;\;\vdots \\ &= 2^k d_{n-k} - \sum_{i=0}^{k-1} 2^i S_{n-1-i} \quad . \end{aligned} \qquad (10)$$

When $k = n-1$ (i.e., $d_{n-k} = d_1 = 1$),

$$d_n = 2^{n-1} - \sum_{i=0}^{n-2} 2^i S_{n-1-i} \quad . \qquad (11)$$

This gives significant insight into the nature of d_n. However, we would like to prove recurrence relation (9) instead of just noticing that it seems to hold. In fact, this is not difficult to do.

Definition 9

Let $LMC'_n = \{x \in LMC \mid x$ is the concatenation of two or more binary tree encodements$\}$.

In other words, $IMC'_n = IMC_n - BTE_n$. Then

$$card(IMC'_n) = card(IMC_n) - card(BTE_n) = d_n - S_n \quad .$$

The two functions f_n and g_n defined below are useful in capturing the relationship between IMC_n and IMC_{n+1}.

Definition 10

f_n: $IMC_n \rightarrow IMC_{n+1}$

$f_n(x) = '0' \,||\, x$

In other words, given a string in IMC_n, the function f_n simply concatenates a '0' to the front of it.

Definition 11

g_n: $IMC'_n \rightarrow IMC_{n+1}$

$g_n(x) = '1' \,||\, x$

In other words, given a string in IMC'_n, the function g_n simply concatenates a '1' to the front of it.

Both f_n and g_n are 1-1 and into IMC_{n+1}. Their respective ranges are disjoint, and their union is all of IMC_{n+1}. Since the cardinality of the domain of f_n is d_n and that of g_n is $d_n - S_n$, d_{n+1} must be $d_n + (d_n - S_n) = 2d_n - S_n$. These observations are proven in the following four lemmas.

Lemma 1

f_n is 1-1 and into IMC_{n+1} .

Proof:

Given x e LMC_n, then length(x) = n. Clearly, length($f_n(x)$) = n + 1 since $f_n(x)$ contains one more bit than x. $f_n(x)$ e LMC_{n+1} by the definitions of LMC and LMC_{n+1}. Thus, f_n is into LMC_{n+1}.

If $f_n(x) = f_n(y)$, then '0' || x = '0' || y . This implies that x = y . Thus, f_n is 1-1.

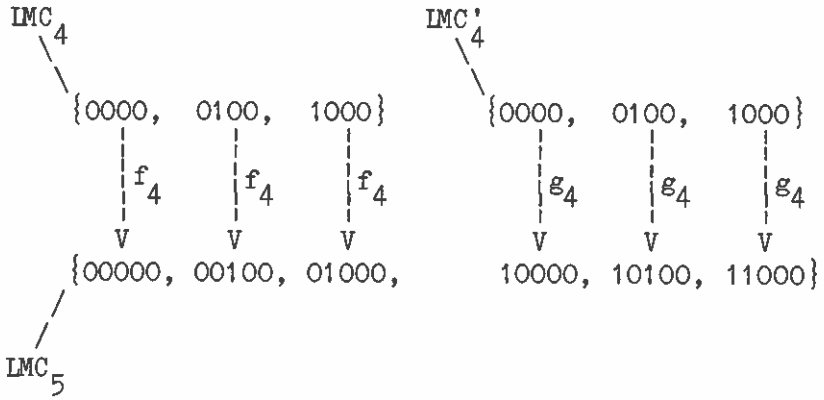
Lemma 2

g_n is 1-1 and into LMC_{n+1} .

Proof:

Since the domain of g_n is LMC'_n, it is only applied to bit configurations that contain two or more binary tree encodements. By placing a '1' in front of such a bit configuration, the first two binary tree encodements are simply combined into one (larger) binary tree encodement. Using this observation and the arguments in the proof of Lemma 1, the proof is completed.

In order to make these lemmas more clear, the mappings from LMC_4 (and LMC'_n) to LMC_5 are shown below:



Definition 12

Let $F_n(LMC_n) = \bigcup_{x \in LMC_n} \{f_n(x)\}$.

Let $G_n(LMC'_n) = \bigcup_{x \in LMC'_n} \{g_n(x)\}$.

F_n and G_n are just the ranges of f_n and g_n, respectively.

Lemma 3

$F_n(LMC_n)$ and $G_n(LMC'_n)$ are disjoint.

Proof:

Everything in $F_n(LMC_n)$ begins with a '0', and everything in $G_n(LMC'_n)$ begins with a '1'. Thus, they can have no elements in common.

Lemma 4

$F_n(LMC_n) \cup G_n(LMC'_n) = LMC_{n+1}$.

Proof:

Every $x \in LMC_{n+1}$ begins with either a '0' or a '1'. If x begins with a '0', then $x = '0' \,||\, y$ for some $y \in LMC_n$; thus, $f_n(y) = x$ and $x \in F_n(LMC_n)$. Similarly, if x starts with a '1', then $x = '1' \,||\, y$ for some $y \in LMC'_n$; thus, $g_n(y) = x$ and $x \in G_n(LMC'_n)$.

Theorem 2

The values of d_n satisfy recurrence relation (9) and equation (11).

Proof:

This is an immediate consequence of Lemma 1 through Lemma 4 and the preceding discussion.

7. A Generating Function Solution

A sequence, $\langle a_n \rangle$, can be summarized by its generating function.

Definition 13

Given a sequence $\langle a_n \rangle$ (n ranging from zero to infinity), its generating function [BRAU79; EISE69; KNUT73; LUEK80; STAN78] is defined by:

$$A(x) = \sum_{n=0}^{inf} a_n x^n \quad .$$

Often, this abstraction of the sequence can be manipulated to extract information about $\langle a_n \rangle$. In our case, we are interested in the sequence $\langle d_n \rangle$ whose generating function is given by:

$$D(x) = \sum_{n=0}^{inf} d_n x^n \quad . \qquad (12)$$

First, we develop generating functions for $\langle T_n \rangle$ and $\langle S_n \rangle$. From these, equation (9) and equation (12), we develop a closed-form expression for D(x), the generating function of $\langle d_n \rangle$. From this, we extract the closed-form values of the d_n, which is our ultimate objective.

Lemma 5

The generating function, T(x), of the sequence $\langle T_n \rangle$ (see equation (1)) is given by:

$$T(x) = \sum_{n=0}^{inf} T_n x^n$$

$$= \sum_{n=0}^{inf} \frac{1}{n+1} C_n^{2n} x^n$$

$$= \frac{(1 - (1-4x)^{1/2})}{2x} \quad . \qquad (13)$$

Proof:

This is proven in [EVEN79] and [KNUT73].

Lemma 6

The generating function, S(x), of the sequence $\langle S_n \rangle$ is given by:

$$S(x) = \sum_{n=0}^{inf} S_n \, x^n$$

$$= \sum_{n=0}^{inf} \frac{2}{n+1} C_{\frac{n-1}{2}}^{n-1} \, x^n$$

$$= xT(x^2)$$

$$= \frac{(1 - (1-4x^2)^{1/2})}{2x} \quad . \qquad (14)$$

Proof:

$$T(x) = \sum_{n=0}^{inf} \frac{1}{n+1} C_n^{2n} \, x^n$$

$$T(x^2) = \sum_{n=0}^{inf} \frac{1}{n+1} C_n^{2n} \, x^{2n}$$

(substitute k = 2n)

$$= \sum_{k=0(2)}^{inf} \frac{1}{\frac{k}{2}+1} C_{\frac{k}{2}}^{k} \, x^k$$

(Here, the notation "k=0(2)" indicates that the summation starts at k=0, and k increases in increments of 2.)

$$xT(x^2) = \sum_{k=0(2)}^{inf} \frac{2}{k+2} C_{\frac{k}{2}}^{k} \, x^{k+1}$$

$$= \sum_{n=1(2)}^{inf} \frac{2}{n+1} C_{\frac{n-1}{2}}^{n-1} x^n \qquad \text{(substitute } n = k + 1\text{)}$$

$$= \sum_{n=0}^{inf} S_n x^n$$

$$= S(x)$$

Thus,

$$S(x) = xT(x^2) = \frac{(1 - (1-4x^2)^{1/2})}{2x} \quad .$$

Lemma 7

The generating function, $D(x)$, of the sequence $\langle d_n \rangle$ (see equation (9)) is given by:

$$D(x) = \sum_{n=0}^{inf} d_n x^n$$

$$= \frac{-1}{2} + \frac{1}{2} \frac{(1-4x^2)^{1/2}}{1-2x} \qquad (15)$$

$$= \frac{-1}{2} + \frac{1}{2} (1+2x)(1-4x^2)^{-1/2} \quad . \qquad (16)$$

Proof:

$$\frac{D(x)}{x} = \sum_{n=0}^{inf} d_{n+1} x^n$$

$$= d_1 + (2d_1 - S_1)x^1 + (2d_2 - S_2)x^2 + \ldots$$

$$= d_1 + 2 \sum_{n=1}^{inf} d_n x^n - \sum_{n=1}^{inf} S_n x^n$$

$$= 1 + 2D(x) - S(x)$$

Thus,

$$D(x) = x + 2xD(x) - xS(x)$$

or

$$D(x) = \frac{x(1-S(x))}{1-2x}$$

$$= \frac{x(1 - [1 - (1-4x^2)^{1/2}/(2x)])}{1-2x}$$

$$= \frac{(2x - 1 + (1-4x^2)^{1/2})}{2(1-2x)}$$

$$= \frac{-1}{2} + \frac{1}{2}\frac{(1-4x^2)^{1/2}}{1-2x}$$

$$= \frac{-1}{2} + \frac{1}{2}(1+2x)(1-4x^2)^{-1/2} \quad .$$

Given this closed-form generating function for $D(x)$, we can determine the coefficients (d_n) in its summation form. By applying convolution techniques to equation (15), equation (11) can be obtained; however, this does not add to our knowledge. By using equation (12), we obtain a closed-form expression for d_n. First, we determine the coefficients in the expansion of $(1-x)^{-1/2}$; next, we determine the coefficients in the expansion of $(1-4x^2)^{-1/2}$; last, we determine the coefficients in the expansion of $D(x)$.

Lemma 8

For $f(x) = (1-x)^{-1/2}$, the summation form of the generating function is given by:

$$f(x) = \sum_{n=0}^{inf} 2^{-2n} C_n^{2n} x^n \quad . \qquad (17)$$

Proof:

Let $f(x) = \sum_{n=0}^{inf} b_n x^n$.

From the calculus, we know that

$$b_n = \frac{f^{(n)}(0)}{n!} \quad .$$

n = 0

$$f^{(0)}(x) = f(x)$$

$$f^{(0)}(0) = (1-0)^{-1/2} = 1$$

n = 1

$$f^{(1)}(x) = (\tfrac{-1}{2})(1-x)^{-3/2}(-1) = (\tfrac{1}{2})(1-x)^{-3/2}$$

$$f^{(1)}(0) = (\tfrac{1}{2})(1-0)^{-3/2} = (\tfrac{1}{2})$$

n = 2

$$f^{(2)}(x) = (\tfrac{1}{2})(\tfrac{-3}{2})(1-x)^{-5/2}(-1) = (\tfrac{1}{2})(\tfrac{3}{2})(1-x)^{-5/2}$$

$$f^{(2)}(0) = (\tfrac{1}{2})(\tfrac{3}{2})(1-0)^{-5/2} = (\tfrac{1}{2})(\tfrac{3}{2})$$

In general,

$$f^{(n)}(0) = \prod_{i=1}^{n} (\tfrac{2i-1}{2})$$

$$= 2^{-n} \prod_{i=1}^{n} (2i-1) \quad .$$

Thus,

$$b_n = \frac{2^{-n}}{n!} \prod_{i=1}^{n} (2i-1)$$

$$= \frac{2^{-n}}{n!} (1*3*5*\ldots*(2n-1))$$

$$= \frac{2^{-n}}{n!} \frac{(1*2*3*\ldots*(2n-1)*2n)}{(2*4*\ldots*2n)}$$

$$= \frac{2^{-n}}{n!} \frac{(2n)!}{2^n n!}$$

$$= 2^{-2n} \frac{(2n)!}{n!n!}$$

$$= 2^{-2n} \, C_n^{2n} \quad .$$

Then,

$$f(x) = \sum_{n=0}^{inf} b_n \, x^n = \sum_{n=0}^{inf} 2^{-2n} \, C_n^{2n} \, x^n \quad .$$

Lemma 9

For $g(x) = f(4x^2) = (1-4x^2)^{-1/2}$, the summation form of the generating function is given by:

$$g(x) = \sum_{n=0(2)}^{inf} C_{\frac{n}{2}}^{n} \, x^n \quad . \qquad (18)$$

Proof:

$$g(x) = f(4x^2)$$

$$= \sum_{n=0}^{inf} 2^{-2n} \, C_n^{2n} \, (4x^2)^n$$

$$= \sum_{n=0}^{inf} 2^{-2n} \, C_n^{2n} \, 2^{2n} \, x^{2n}$$

$$= \sum_{n=0}^{inf} C_n^{2n} \, x^{2n}$$

(substitute $j = 2n$)

$$= \sum_{j=0(2)}^{inf} C_{\frac{j}{2}}^{j} \, x^j$$

Lemma 10

The summation form of D(x) is given by:

$$D(x) = \frac{-1}{2} + \frac{1}{2} \sum_{n=0(2)}^{inf} C_{\frac{n}{2}}^{n} x^n + \sum_{n=1(2)}^{inf} C_{\frac{n-1}{2}}^{n-1} x^n \quad . \tag{19}$$

Proof:

$$(1+2x)(1-4x^2)^{-1/2} = (1+2x) \sum_{n=0(2)}^{inf} C_{\frac{n}{2}}^{n} x^n$$

$$= \sum_{n=0(2)}^{inf} C_{\frac{n}{2}}^{n} x^n + 2 \sum_{n=0(2)}^{inf} C_{\frac{n}{2}}^{n} x^{n+1}$$

$$= \sum_{n=0(2)}^{inf} C_{\frac{n}{2}}^{n} x^n + 2 \sum_{n=1(2)}^{inf} C_{\frac{n-1}{2}}^{n-1} x^n$$

Then,

$$D(x) = \frac{-1}{2} + \frac{1}{2} (1+2x)(1-4x^2)^{-1/2}$$

$$= \frac{-1}{2} + \frac{1}{2} \sum_{n=0(2)}^{inf} C_{\frac{n}{2}}^{n} x^n + \sum_{n=1(2)}^{inf} C_{\frac{n-1}{2}}^{n-1} x^n \quad .$$

Theorem 3

The value of d_n is given by:

$$d_n = 0 \qquad \text{for } n = 0 \tag{4}$$

$$= \frac{1}{2} C_{\frac{n}{2}}^{n} \qquad \text{for } n > 0 \text{ and even}$$

$$= C_{\frac{n-1}{2}}^{n-1} \qquad \text{for } n > 0 \text{ and odd} \quad .$$

Proof:

This is an immediate consequence of Lemma 10 and equation (19).

Theorem 3 gives us a closed-form expression for $d_n = \text{card}(LMC_n)$, and we have obtained our desired result.

8. Analysis

In this section, we analyze the encoding efficiency of the BTE scheme. This is done by comparing the number of bits required to encode the same amount of information in the BTE scheme and the SBE scheme.

Refering to Table 2, notice that $d_1 = 1$ and $d_3 = 2$. Clearly, two (2^1) encodements can be held in $n = 1$ bits in the SBE scheme, and it is not until $n = 3$ that the BTE scheme can encode the same amount of information. Thus, for small n, the BTE scheme requires three times the number of bits to encode the same amount of information as the SBE scheme. This, of course, is unacceptable and might lead one to infer that the BTE scheme is an extremely inefficient one. However, for large n, the storage capacity loss is not significant, as shown by the analysis presented below.

Before presenting a formal analysis, we indicate the magnitude of the storage capacity loss (see Definition 8) based on the results in Table 2. For example (see $n = 10^2$ in Table 2), to obtain 5.045×10^{28} encodements, the SBE scheme requires $\log(5.045 \times 10^{28})$ or 95.349 bits; on the other hand, the BTE scheme requires $n = 100$ bits. The extra 4.651 bits required by the BTE scheme is a small fraction of the entire 100 bits. As can be seen from Table 2 (columns a_n and r_n), this fraction decreases as n increases. The nature of the information loss is formalized by the following analysis.

For n even,

$$d_n = \frac{1}{2} \frac{n!}{(\frac{n}{2})!^2}$$

and

$$\log(d_n) = -1 + \log(n!) - 2\log((\tfrac{n}{2})!) \quad . \tag{20}$$

Using Stirling's approximation

$$(2*PI*n)^{1/2}(\tfrac{n}{e})^n <= n! <= (2*PI**n)^{1/2}(\tfrac{n}{e})^n(1 + \tfrac{1}{12n}) \tag{21}$$

the following bounds can be derived:

$$n - \tfrac{1}{2}\log(n) - 1.77 <= \log(d_n) <= n - \tfrac{1}{2}\log(n) - 1.21 \quad . \tag{22}$$

For n odd,

$$d_n = 2d_{n-1} \quad .$$

Then,

$$\log(d_n) = 1 + \log(d_{n-1})$$

and it can be shown that

$$n - \tfrac{1}{2}\log(n) - 1.77 <= \log(d_n) <= n - \tfrac{1}{2}\log(n) - 0.71 \tag{23}$$

for arbitrary n >= 1. (Note that the third column of Table 2 satisfies this inequality.) Since $a_n = n - \log(d_n)$ (see Definition 8) we obtain

$$0.71 + \tfrac{1}{2}\log(n) <= a_n <= 1.77 + \tfrac{1}{2}\log(n) \tag{24}$$

and

$$\frac{0.71 + \frac{1}{2}\log(n)}{n} <= r_n <= \frac{1.77 + \frac{1}{2}\log(n)}{n} \tag{25}$$

Clearly, the upper and lower bounds in equation (25) go to zero as n goes to infinity. Moreover, the dominant term of log(n)/n tends to zero quickly. For example, when $n = 10^3$, the absolute storage capacity loss is between 5.69 and 6.75 bits; when $n = 10^6$, it is between 10.68 and 11.74; when $n = 10^9$, it is between 15.66 and 16.72. Clearly, for large n, the storage capacity loss is not significant.

9. Summary and Conclusions

Four techniques have been presented for calculating d_n, the number of n-bit legal memory configurations using the BTE scheme. Three equivalent formulas have been developed:

1. Equation (8)

$$d_n = \sum_{(PARTITIONS_n)} \left(\prod_{j=1}^{k} (S_{n_j})^{m_j}\right) * C^{m}_{m_1,m_2,\ldots,m_k}$$

where

$PARTITIONS_n$:

1) All the n_i are positive odd integers.

2) All the m_i are positive integers.

3) $n_1 < n_2 < \ldots < n_k$.

4) $\sum_{i=1}^{k} n_i * m_i = n$.

and

$$m = \sum_{i=1}^{k} m_i$$

2. Equation (11)

$$d_n = 2^{n-1} - \sum_{i=0}^{n-2} 2^i S_{n-1-i}$$

3. Equation (4)

$$d_n = 0 \qquad \text{for } n = 0$$

$$= \frac{1}{2} C^n_{\frac{n}{2}} \qquad \text{for } n > 0 \text{ and even}$$

$$= C^{n-1}_{\frac{n-1}{2}} \qquad \text{for } n > 0 \text{ and odd}$$

It has been shown that, for large n, the storage capacity loss due to using the BTE scheme is not significant. Note that this statement cannot be made for small n. For instance, assuming a byte oriented architecture (of, say, 8 bits) in which binary tree encodements cannot be concatenated across byte boundaries (but must reside in a single byte), the encoding efficiency of the BTE scheme remains at a constant (probably unacceptable) percentage of the SBE scheme, independent of the total memory size.

Although it would seem that the BTE scheme is less efficient than the SBE scheme, pragmatic properties of certain applications may produce exactly the opposite effect. For example, by using this scheme, it may be possible to hold more <u>logical</u> information because delimiters and/or pointers are not necessary; also, variable length codes are implicitly available. Many implementation decisions must be considered to adequately evaluate the viability of the BTE scheme. Such decisions involve the actual encodements selected for specific symbols, the physical memory structure and organization in which it is used, and the applications in which it is used. Although it is beyond the scope of this paper to prove the benefits of using the BTE scheme, we have shown that it is an interesting alternative, and the storage capacity loss due to its use is not significant.

References

[BRUA79] Brualdi, Richard A., Introductory Combinatorics, North Holland, 1979.

[COX80] Cox, J. R., Kimura, T. D., Moore, P., Gillett, W. D. and Stucki, M. J., "Design Studies Suggested by an Abstract Model for a Medical Information System," Proceedings of the Fourth Annual Symposium on Computer Applications in Medical Care, Joseph T. O'Neill, Editor, Vol. 3, pp. 1485-1494, Washington, D. C., November, 1980.

[EISE69] Eisen, Martin M., Elementary Combinatorial Analysis, Gordon & Breach, 1969.

[EVEN79] Even, Shimon, Graph Algorithms, Computer Science Press, 1979.

[KIMU81a] Kimura, T. D., Cox, J. R. and Gillett, W. D., "An Abstract Model of an Unstratified Database System," Proceedings of the Fourteenth Hawaii International Conference on Systems Science, W. Riddle, K. Thurber and R. H. Sprague, Editors, Vol. 1, pp. 115-126, Honolulu, Hawaii, January 1981.

[KIMU81b] Kimura, T. D. and Gillett, W. D., "Formal Specifications of ADS, an Abstract Database System," Technical Report No. WUCS-81-3, Department of Computer Science, Washington University, May 1981.

[KNUT73] Knuth, Donald E., The Art of Computer Programming: Fundamental Algorithms, Volume 1 (second edition), Addison-Wesley, 1973.

[LUEK80] Lueker, George S., "Some Techniques for Solving Recurrences," ACM Computing Surveys 12:4 (Dec. 1980), pp. 419-436.

[STAN78] Stanley, R. P., "Generating Functions," MAA Studies in Mathematics, Vol. 17: Studies in Combinatorics, Gian-Carlo Rota, Editor, The Mathematical Association of America, 1978, pp. 100-141.