INTERNATIONAL JOURNAL OF COMMUNICATION SYSTEMS
Int. J. Commun. Syst. 2008; **21**:1091–1120
Published online 4 June 2008 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/dac.943

An efficient compression scheme for data communication which uses a new family of self-organizing binary search trees

Luis Rueda[1,‡] and B. John Oommen[2,3,*,†,§,¶,‖]

[1]*Department of Computer Science, University of Concepcion, Edmundo Larenas 215, Concepcion 4030000, Chile*
[2]*School of Computer Science, Carleton University, 1125 Colonel By Dr., Ottawa, Ont., Canada K1S 5B6*
[3]*Department of Information and Communication Technology, University of Agder, Grimstad, Norway*

SUMMARY

In this paper, we demonstrate that we can effectively use the results from the field of adaptive *self-organizing* data structures in enhancing compression schemes. Unlike *adaptive* lists, which have already been used in compression, to the best of our knowledge, adaptive *self-organizing* trees have not been used in this regard. To achieve this, we introduce a new data structure, the *partitioning binary search tree* (PBST) which, although based on the well-known *binary search tree* (BST), also appropriately partitions the data elements into mutually exclusive sets. When used in conjunction with Fano encoding, the PBST leads to the so-called *Fano binary search tree* (FBST), which, indeed, incorporates the required Fano coding (nearly equal probability) property into the BST. We demonstrate how both the PBST and the FBST can be maintained adaptively and *in a self-organizing manner*. The updating procedure that converts a PBST into an FBST, and the corresponding new *tree*-based operators, namely the shift-to-left and the shift-to-right operators, are explicitly presented. The encoding and decoding procedures that also update the FBST have been implemented and rigorously tested. Our empirical results on the files of the well-known benchmarks, the Calgary and Canterbury Corpora, show that the adaptive Fano coding using FBSTs, the Huffman, and the greedy adaptive Fano coding achieve similar compression ratios. However, in terms of encoding/decoding speed, the new scheme is *much* faster than the latter two in the encoding phase, and they achieve approximately the same speed in the decoding phase. We believe that the same philosophy, namely that of using an adaptive self-organizing BST to maintain the frequencies, can also be utilized for other data encoding mechanisms, even as the Fenwick scheme has been used in arithmetic coding. Copyright © 2008 John Wiley & Sons, Ltd.

Received 17 January 2008; Revised 19 March 2008; Accepted 19 March 2008

*Correspondence to: B. John Oommen, School of Computer Science, Carleton University, 1125 Colonel By Dr., Ottawa, Ont., Canada K1S 5B6.
†E-mail: oommen@scs.carleton.ca
‡Member of the IEEE.
§Chancellor's Professor.
¶IEEE and IAPR Fellow.
‖Adjunct Professor.

Contract/grant sponsor: CONICYT; contract/grant number: 1060904
Contract/grant sponsor: NSERC

KEY WORDS: online encoding; adaptive data compression; binary search trees; adaptive data structures

1. INTRODUCTION

This paper demonstrates how techniques applicable for defining and maintaining adaptive *self-organizing* data structures can be incorporated into 'traditional' compression techniques to yield enhanced superior schemes. However, as opposed to the adaptive list-based structures that have been reported in the literature (for example, in block sorting [1] and in [2]) we argue that adaptive *tree-based* schemes have their distinct advantages, and this claim has been demonstrated by using the principles on a Fano scheme, which has recently attracted fascinating research attention [3, 4].

1.1. Adaptive lists and trees

Adaptive lists have been investigated for more than three decades, and schemes such as the move-to-front (MTF), transposition, move-*k*-ahead, the move-to-rear families [5], and randomized algorithms [6] have been proposed. A complete survey of these methods and their applications can be found in [7], and their applicability in compression has also been acclaimed, for example, of the MTF in block sorting [1], and by Albers *et al.* in [2]. As opposed to this, a number of adaptive tree-based algorithms have also been presented over the years, and we are unaware of any reported strategy that utilizes adaptive tree-based algorithms in compression. In the interest of completeness, it is prudent to briefly survey the latter methods here.

Binary search trees (BSTs) have been used in a wide range of applications that include storage, dictionaries, databases, and symbol tables. BSTs can be maintained statically when the statistical information about the number of accesses to the records is known *a priori*. When the probabilistic distribution about the records is unknown, adaptive schemes are the most appropriate ones. These schemes update the BST during the search process. Consider a set of records whose keys are given by the ordered set of distinct elements $\mathscr{K} = \{k_1, \ldots, k_m\}$, where $k_1 < \cdots < k_m$. By following the procedure given in [8], the optimal BST can be constructed using dynamic programming in $O(m^2)$ time and space. Alternatively, using dynamic programming and divide-and-conquer techniques [9], a nearly optimal BST can be constructed in $O(m)$ space and $O(m \log m)$ time. These two approaches can be used whenever the statistical information about the access to the records is known beforehand. As opposed to this, we assume that these probabilities are unknown, and the structure and the content of the BST are dynamically changed while the records are searched for, in the tree.

The *move-to-root* heuristic, proposed by Allen and Munro [10], is a very simple approach to maintain an adaptive BST. The aim of this approach is to maintain the most frequently accessed records near the root, and consequently, to minimize the average cost of searching. Another approach, the *simple exchange rule*, was also introduced by Allen and Munro [10]. It consists of rotating the accessed record one level up towards the root. Although this approach is not very efficient, it has the advantage that it does not use extra space. *Splaying* is another technique according to Sleator and Tarjan [11–13]. It uses its own tree structure called *splay tree*. The main idea of this technique is to move the accessed record towards the root and to simultaneously allow accesses to each record by an in-order traversal of the tree. The splaying tree-structuring techniques have reported good results even for highly time-variant access probabilities. Another scheme found in the literature is known as the *monotonic tree* [14]. Each record maintains an extra memory location to count the number of times it has been accessed. This approach performs poorly

DOI: 10.1002/dac

for key sets with high entropy. Empirical results have also shown that, on the average, it behaves poorly. Other adaptive BST approaches are *biasing* [15], *dynamic binary search* [16], *weighted randomization* [17], *deepsplaying* [18], and the technique that uses *conditional rotations* [19]. The basic idea of the latter approach is to maintain certain key pieces of information in each node. These are used by the heuristic called the *conditional rotation*, based on the fundamental *rotation operation* (also known as the *promotion operation* [20]) introduced by Adel'son-Vel'skii and Landis [21].

1.2. Available compression schemes

Since we intend to propose a 'marriage' between the fields of adaptive tree-based data structures and compression, a brief introduction of the latter field is not out of place. Clearly, being so vast, the latter field cannot be surveyed here—it probably contains tens of thousands of books and articles.** However, to place our results in the right perspective, we *briefly* mention the salient points of interest.

Adaptive coding is important in many applications that require online data compression and transmission. This modality is advantageous since the data is encoded by performing a single pass, as opposed to the strategy used in static algorithms which requires two passes—the first to learn the probabilities and the second to accomplish the encoding.

Most of the well-known static encoding techniques have been extended to also function in an adaptive manner. The most well-known adaptive coding technique is Huffman's algorithm [24], which was first presented by Faller in 1973 [25]. Being unaware of the work done by Faller, Gallager presented an alternate adaptive version of Huffman's algorithm in 1978 [26]. The latter was later augmented by Knuth in 1985, who presented a more efficient algorithm to adaptively maintain the Huffman tree [27]. The most recent and efficient version of the adaptive Huffman coding (AHC) is the one introduced by Vitter in 1987 [28].

Another important encoding method that has been extended for its adaptive version is the *arithmetic coding* scheme. Details of its modeling and its implementation can be found in [22, 29]. Other important adaptive methods are the *interval* and *recency rank encoding* [29] and the *Elias omega codes* [30]. While the former methods are efficient for a particular source distribution only, the latter have been found to use less memory than Huffman's adaptive coding and are applicable to compress data from universal sources. On the other hand, adaptive coding approaches that use higher-order statistical models, and other structural models, include *dictionary techniques* (LZ and its enhancements) [31, 32], *prediction with partial matching* (PPM) [33], and *grammar-based compression* (GBC) [34]. Splay trees have also been used in adaptive data compression [35].

Adaptive methods for Fano coding have been recently introduced (for the binary and multi-symbol code alphabets) [36], which have been shown to work faster than AHC and consume one-sixth of the resources required by the latter. Although these methods are efficient, they need to maintain a *list* of the source symbols and the respective probabilities from which the Fano coding tree can be *partially* reconstructed at each encoding step. Closely related to this are two excellent works by Gagie [3, 4], which describe a new efficient one-pass algorithm based on Shannon's coding. The latter is simpler to implement and analyze than Knuth's or Vitter's [27, 28] (which use codewords longer than $\log n$ bits) and is faster and easier when each codeword fits in a

**A good recent text book surveying the field is [22], and the proceedings of the following conferences on data compression [23] also give very good perspectives of the state of the art.

DOI: 10.1002/dac

machine word. Observe that in [3] Gagie introduced a new data structure to maintain a code-tree explicitly—which is also the spirit of our present work.[††]

1.3. Research hypothesis and contributions

Our primary hypothesis is that we can effectively use results from the field of adaptive *self-organizing* data structures in enhancing compression schemes. *Adaptive* lists have been used earlier in compression [2], and the Fenwick tree [37, 38] has brilliantly used the list of probabilities, maintained as a tree, to maintain probability estimates.[‡‡] However, to the best of our knowledge, adaptive *self-organizing* trees have not been used in this regard, and this is what we shall endeavor to do.

To achieve our goal, we shall show that adaptive *self-organizing* BSTs can be used advantageously, and in doing so, we circumvent the issue of maintaining the probabilities as *lists*. Rather, we introduce a new structure called the partitioning BST (PBST), which partitions the probabilities into mutually exclusive subsets possessing the 'BST' property. Applying this to the Fano scheme leads to the so-called *Fano binary search tree* (FBST), which is a generalization of the BST, having its own associated *shift operators*. The latter are the resultant tree-modifying operations designed for the specific data structure, the PBST. The maintenance of the FBST, in turn, is used to adaptively and efficiently encode an input sequence, where we assume that the probabilities of the source symbols are unknown. At the beginning of the encoding process, the uncertainty about the occurrence of the next symbol will cause many statistical and structural changes in the tree during the so-called *transient phase*. After this phase,[§§] fewer changes are expected, and hence the structural changes are dramatically reduced so as to achieve the desired behavior, namely the computation of the *optimal* Fano encoding and decoding mechanisms. Thus, the first advantage gleaned of using trees (instead of lists) is the less expensive (logarithmic as opposed to linear) update mechanism. Additionally, the advantage gained by using the PBST and the associated shift operators is that as the PBST converges, the asymptotic incremental cost goes to zero.

Although the principles of using adaptive *self-organizing* data structures have been demonstrated on a Fano scheme, we believe that the same principles can also be extended to other compression methods, even as the Fenwick scheme has been used in arithmetic coding [37, 38], and list update algorithms have been used in data compression [2]. The combination of adaptive tree-based structures with other statistical and/or dictionary-based methods could undoubtedly lead to more efficient compression schemes implied by the higher-order models being augmented with the additional speed-enhanced updating procedure provided by the former principles. This is a problem that we are currently investigating.

2. FANO BINARY SEARCH TREES

The basis for the particular 'species' of BSTs introduced in this paper, the FBST, comes from the structure used in the *conditional rotation heuristic* [19], which we briefly introduce below.

[††]Gagie's approach is, indeed, very impressive, and as he himself states, is related to our previous work of [36]. Although the work was theoretically sound, it did not include any experimental verification, which our present work does.

[‡‡]We should mention, however, that the Fenwick tree [37] does not use a *self-organizing* adaptive structure.

[§§]This assumes the *stationarity* of the source.

DOI: 10.1002/dac

Consider a (complete) BST, $\mathscr{T}=\{t_1,\ldots,t_{2m-1}\}$. If t_i is any internal node of $\mathscr{T}$, then:

- P_i is the parent of t_i,
- L_i is the left child of t_i,
- R_i is the right child of t_i,
- B_i is the sibling of t_i, and
- P_{P_i} is the parent of P_i (or the grandparent of t_i).

Using these primitives, B_i can also be defined as follows:

- L_{P_i} if t_i is the right child of P_i and
- R_{P_i} if t_i is the left child of P_i.

The first heuristic introduced in [19] requires three extra memory locations for each node, which represent the number of accesses to the record, the number of accesses to the subtree rooted at that record, and the weighted path length (WPL) of the subtree rooted at that record.

The aim of this approach is to minimize the WPL of the subtree rooted at t_i, where the node t_i contains the record being accessed. The fields that contain the information about the number of accesses to t_i, the number of accesses to the subtree rooted at t_i, etc., are updated, and a rotation on t_i is performed whenever the WPL decreases as a result of the rotation. Since the WPL of the entire tree depends also on that of t_i, the authors of [19] showed that this also results in a decrease in the WPL of the *entire* tree.

Since we will require the same formalism, we introduce the notation used in the conditional rotation heuristic: $\alpha_i(n)$ is the total number of accesses to node t_i up to time n; $\tau_i(n)$ is the total number of accesses to $\mathscr{T}_i$, the subtree rooted at t_i, up to time n, and is calculated as follows:

$$\tau_i(n)=\sum_{t_j\in\mathscr{T}_i}\alpha_j(n) \tag{1}$$

$\kappa_i(n)$ is the WPL of $\mathscr{T}_i$, the subtree rooted at t_i, at time n, and is calculated as follows:

$$\kappa_i(n)=\sum_{t_j\in\mathscr{T}_i}\alpha_j(n)\lambda_j(n) \tag{2}$$

where $\lambda_j(n)$ is the path length from t_j up to node t_i.

By using simple induction, it can be shown that

$$\kappa_i(n)=\sum_{t_j\in\mathscr{T}_i}\tau_j(n) \tag{3}$$

In order to simplify the notation, we let α_i, τ_i, and κ_i be the corresponding values (as defined in the conditional rotation heuristic) contained in node t_i at time n, i.e. $\alpha_i(n)$, $\tau_i(n)$, and $\kappa_i(n)$, respectively.

Broadly speaking, an FBST is a BST in which the number of accesses of each internal node is set to zero, and the number of accesses of each leaf represents the number of times that the symbol associated with that leaf has appeared so far in the input sequence. The aim is to maintain the tree balanced in such a way that for every internal node, the weight of the left child is as nearly equal as possible to that of the right child.

Definition Structure_PBST

Consider the source alphabet $\mathscr{S}=\{s_1,\ldots,s_m\}$ whose probabilities of occurrence are $\mathscr{P}=[p_1,\ldots,p_m]$, where $p_1\geqslant\cdots\geqslant p_m$, and the code alphabet $\mathscr{A}=\{0,1\}$. A *PBST* is a binary tree,

DOI: 10.1002/dac

$\mathscr{T}=\{t_1,\ldots,t_{2m-1}\}$, whose nodes are identified by their indices (for convenience, also used as the keys $\{k_i\}$), and whose fields are the corresponding values of τ_i. Furthermore, every PBST satisfies

(i) Each node t_{2i-1} is a leaf, for $i=1,\ldots,m$, where s_i is the ith alphabet symbol satisfying $p_i \geqslant p_j$ if $i<j$.
(ii) Each node t_{2i} is an internal node, for $i=1,\ldots,m-1$.

Remark 1
Given a PBST, $\mathscr{T}=\{t_1,\ldots,t_{2m-1}\}$, the number of accesses to a leaf node, α_{2i-1}, is a counter, and p_i refers to either α_{2i-1} (the frequency counter) or the probability of occurrence of the symbol associated with t_{2i-1}. We shall use both representations interchangeably. In fact, the probability of occurrence of s_i can be estimated (in a maximum likelihood manner) as follows:

$$p_i=\frac{\alpha_{2i-1}}{\sum_{j=1}^{m}\alpha_{2j-1}} \tag{4}$$

We now introduce a particular case of the PBST, the *FBST*. This tree has the added property that each partitioning step is performed by following the principles of the Fano coding, i.e. the weights of the two new nodes are as nearly equal as possible. This is formally defined below.

Definition Structure_FBST
Let $\mathscr{T}=\{t_1,\ldots,t_{2m-1}\}$ be a PBST. $\mathscr{T}$ is an *FBST*, if for every internal node, t_{2i}, the following conditions are satisfied:

(a) $\tau_{R_i}-\tau_{L_i}\leqslant\tau_{2i+1}$ if $\tau_{L_i}<\tau_{R_i}$,
(b) $\tau_{L_i}-\tau_{R_i}\leqslant\tau_{2i-1}$ if $\tau_{L_i}>\tau_{R_i}$, or
(c) $\tau_{L_i}=\tau_{R_i}$.

The procedure for constructing an FBST from the source alphabet symbols and their probabilities of occurrence are depicted in Algorithm *Fano_BST_Construction*. The partitioning procedure is similar to that of the greedy adaptive Fano coding presented in [36]. Each time a partitioning is performed, two sublists are obtained, and two new nodes are created, t_{n_0} and t_{n_1}, which are assigned to the left child and the right child of the current node, t_n. This partitioning is recursively performed until a sublist with a single symbol is obtained. Each time the procedure FanoBST(...) is invoked, two sublists of $\mathscr{S}$ and $\mathscr{P}$, respectively, are sent as parameters. These sublists are specified by a *lower-bound* index and an *upper-bound* index, u and l, respectively. We use the sub-index n to refer to the fields of node t_n. For example, τ_n represents the total number of accesses to node t_n.

In order to ensure that $\mathscr{T}$ satisfies properties (i) and (ii) of Definition Structure_PBST and also the conditions of Definition Structure_FBST, the FBST generated by procedure FanoBST(...) must be rearranged as if each node were accessed in a traversal order, from left to right. The sorted FBST is generated by invoking procedure FanoBSTSort(...), which produces a list of nodes in the desired order, $\mathscr{T}=\{t_1,\ldots,t_{2m-1}\}$.

We present below an example that helps to clarify the procedures given in Algorithm *Fano_BST_Construction*.

Example 1
Consider the source alphabet $\mathscr{S}=\{a,b,c,d,e\}$ whose frequency counters are $\mathscr{P}=[8,3,3,3,3]$, and the code alphabet $\mathscr{A}=\{0,1\}$. A PBST, $\mathscr{T}=\{t_1,\ldots,t_9\}$, constructed with $\mathscr{S}$, $\mathscr{P}$, and $\mathscr{A}$ is depicted in Figure 1.

DOI: 10.1002/dac

Algorithm 1 Fano_BST_Construction

Input: The source alphabet and probabilities, $\mathscr{S}$ and $\mathscr{P}$.
Output: The FBST, $\mathscr{T} = \{t_1, \ldots, t_{2m-1}\}$.
Method:

```
procedure FanoBST(𝒮, 𝒫 : list; u, l : integer; t_n : node; τ_n : real);
  i ← u − 1; p ← 0
  while p < τ_n/2 do
    i ← i + 1; p ← p + p_i
  endwhile
  if τ_n/2 − p + p_i/2 < 0 then   // Are the properties of Definition Structure_FBST satisfied?
    i ← i − 1; p ← p − p_i
  endif
  k_n ← 2i
  Create nodes t_{n_0} and t_{n_1}
  L_n ← t_{n_0}; R_n ← t_{n_1}
  if u < i then
    τ_{L_n} ← p; FanoBST(𝒮, 𝒫, u, i, L_n, τ_{L_n})
  else
    α_{L_n} ← p_t; σ_{L_n} ← s_t; k_{L_n} ← 2i − 1
  endif
  if l > i + 1 then
    τ_{R_n} ← τ_n − p; FanoBST(𝒮, 𝒫, i + 1, l, R_n, τ_{R_n})
  else
    α_{R_n} ← p_b; σ_{R_n} ← s_b; k_{R_n} ← 2i + 1
  endif
endprocedure
procedure FanoBSTSort(t_n : node; var 𝒯 : list; var m : integer);
  if L_n ≠ NIL then
    FanoBSTSort(L_n, 𝒯, m)
  endif
  m ← m + 1; t_i ← t_n
  if R_n ≠ NIL then
    FanoBSTSort(R_n, 𝒯, m)
  endif
endprocedure
Create node root
τ_root ← Σ_{i=1}^{m} p_i
FanoBST(𝒮, 𝒫, 1, m, root, τ_root)
𝒯 ← {}; m ← 0; FanoBSTSort(root, 𝒯, m)
end Algorithm Fano_BST_Construction
```

This PBST is also an FBST, i.e. it also satisfies properties of Definition Structure_FBST for every internal node, t_{2i}, for $i = 1, \ldots, m-1$. The corresponding FBST construction procedure is depicted in Figure 2.

DOI: 10.1002/dac

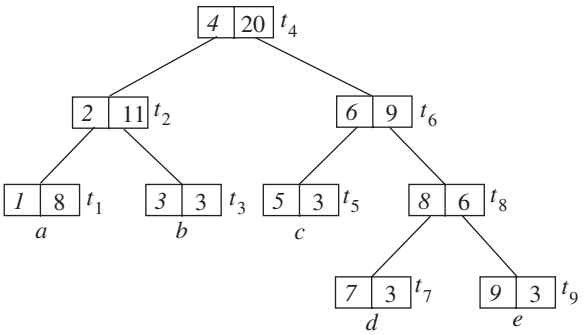

Figure 1. An example of an FBST constructed from $\mathscr{S}=\{a,b,c,d,e\}$ and $\mathscr{P}=[8,3,3,3,3]$.

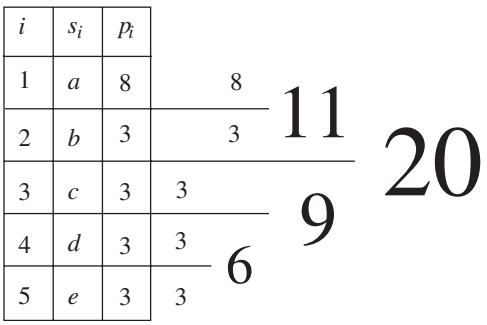

Figure 2. PBST construction procedure for $\mathscr{S}=\{a,b,c,d,e\}$, $\mathscr{P}=[8,3,3,3,3]$, and $\mathscr{A}=\{0,1\}$. The resulting tree is an FBST.

The internal nodes are t_2, t_4, t_6, and t_8, i.e. t_{2i} for $i=1,\ldots,4$. The leaves are the nodes t_1, t_3, t_5, t_7, and t_9. For every node, the cell on the left contains the key, and the cell on the right contains the total number of accesses to the subtree rooted at that node. Each leaf is also associated with a source alphabet symbol, s_i, where $2i-1$ is the key of that leaf.

All the nodes of $\mathscr{T}$ are sorted from left to right, in an ascending order of key, from 1 to 9. The leaves are also sorted from left to right in a descending order of frequency counter.

For each internal node, the key contains the value $2i$, where i represents the index of the last source symbol of the top-most list derived from the partitioning. For example, the first partitioning produces $\{a,b\}$ and $\{c,d,e\}$ whose weights are 11 and 9, respectively. The node at which the partitioning is done is the root. Since the last symbol of $\{a,b\}$ is b whose index in $\mathscr{S}$ is $i=2$, the key for the root is $2i=4$. As we will see later, the correspondence between the index of the source symbol and the key of the internal node is very useful in the implementation of the encoding algorithm.

Remark 2

The structure of the FBST is similar to the structure of the BST used in the conditional rotation heuristic introduced in [19]. The difference, however, is that since every internal node does not represent an alphabet symbol, the values of α_{2i} are all set to zero, and the quantities for the leaf nodes, $\{\alpha_{2i-1}\}$, are set to $\{p_i\}$ or to frequency counters representing them.

DOI: 10.1002/dac

Clearly, the total number of accesses to the subtree rooted at node t_{2i}, τ_{2i}, is obtained as the sum of the number of accesses to all the leaves of $\mathscr{T}_{2i}$. This is stated in the lemma below, whose proof is given in Appendix A. The result is fairly straightforward but included for the sake of completeness. Also, it is included so that the parallel between FBSTs and traditional BSTs becomes clear to the reader.

Lemma 1
Let $\mathscr{T}_{2i} = \{t_1, \ldots, t_{2s-1}\}$ be a subtree rooted at node t_{2i}. The total number of accesses to $\mathscr{T}_{2i}$ is given by

$$\tau_{2i} = \sum_{j=1}^{s} \alpha_{2j-1} = \tau_{L_{2i}} + \tau_{R_{2i}} \tag{5}$$

We now present a result that relates the WPL of an FBST and the average code word length of the encoding schemes generated from *that* tree. For any FBST, $\mathscr{T}$, κ is calculated using (3). By optimizing on the relative properties of κ and τ, we can show that the average code word length of the encoding schemes generated from $\mathscr{T}$, $\bar{\ell}$, can be calculated from the values of κ and τ that are maintained at the root. Note that this is done with a single access—without traversing the entire tree. This result is stated and proved in Theorem 1.

This is quite an 'intriguing' result. The issue at stake is to compute the expected value of a random variable, in this case the *expected code word length*. In general, this can be done if we are given the values that the random variable assumes and their corresponding probabilities. The actual computation would involve the summation (or an integral in the case of continuous random variables) of the product of the values and their associated probabilities. Theorem 1 shows how this expected code word length can be computed quickly—without *explicitly* computing either the product or the summation. However, this is done *implicitly*, since κ and τ take these factors into consideration. Since the FBST is maintained adaptively, the average code word length is also maintained adaptively. Invoking this result, we can obtain the average code word length by a single access to the root of the FBST. The proof of this theorem can be found in Appendix A.

Theorem 1
Let $\mathscr{T} = \{t_1, \ldots, t_{2m-1}\}$ be an FBST constructed from the source alphabet $\mathscr{S} = \{s_1, \ldots, s_m\}$ whose probabilities of occurrence are $\mathscr{P} = [p_1, \ldots, p_m]$. If $\phi : \mathscr{S} \rightarrow \{w_1, \ldots, w_m\}$ is an encoding scheme generated from $\mathscr{T}$, then

$$\bar{\ell} = \sum_{i=1}^{m} p_i \ell_i = \frac{\kappa_{\text{root}}}{\tau_{\text{root}}} - 1 \tag{6}$$

where ℓ_i is the length of w_i, τ_{root} is the total number of accesses to $\mathscr{T}$, and κ_{root} is as defined in (2).

From Theorem 1, we see that the WPL and $\bar{\ell}$ are closely related. The smaller the WPL, the smaller the value of $\bar{\ell}$. Consequently, the problem of minimizing the WPL of an FBST is equivalent to minimizing the average code word length of the encoding schemes obtained from that tree.

DOI: 10.1002/dac

3. SHIFTING OPERATIONS IN PARTITIONING BSTs

The aim of our online encoding/decoding is to maintain a structure that maximally contains and utilizes the statistical information about the source. Using this structure, the current symbol is encoded and the structure is updated in such a way that the next symbol is expected to be encoded as optimally as possible. Various structure models have been proposed, including Markov models (higher-order models), dictionaries, lists, Huffman trees, etc. The latter can be combined with the other models to achieve even more efficient compression.

Alternatively, we propose to use our new structure, namely the FBST defined in Section 2, which is adaptively maintained by simultaneously encoding and learning details about the relevant statistics of the source.[¶¶] The learning process requires that two separate phases are sequentially performed. The first consists of updating the frequency counters of the current symbol, and the second involves changing the structure of the PBST so as to maintain an FBST. Other adaptive encoding techniques, such as Huffman coding or arithmetic coding, utilize the same sequence of processes: encoding and then learning.

After the encoder updates the frequency counter of the current symbol and the corresponding nodes, the resulting PBST may need to be changed so that the FBST is maintained consistently. To achieve this, we introduce two new *shift* operators that can be performed on a PBST: the *shift-to-left* (*STL*) operator and the *shift-to-right* (*STR*) operator.[|||] Broadly speaking, these operators consist of removing a node from one of the sublists obtained from the partitioning and inserting it into the other sublist, in such a way that the new partitioning satisfies the properties of Definition Structure_PBST.

3.1. The STL operator

The STL operator, performed on an internal node of a PBST, consists of removing the *left*-most leaf of the subtree rooted at the *right* child of that node and inserting it as the *right*-most leaf in the subtree rooted at the *left* child of that node.

The relationship between the STL operator and the Fano code construction procedure is the following. Suppose that a list $\mathscr{P}=[p_1,\ldots,p_m]$ has already been partitioned into two new sublists, $\mathscr{P}_0=[p_1,\ldots,p_k]$ and $\mathscr{P}_1=[p_{k+1},\ldots,p_m]$. The equivalent to the STL operator for this scenario consists of deleting p_{k+1} from $\mathscr{P}_1$ and inserting it into the last position of $\mathscr{P}_0$, yielding $\mathscr{P}'_0=[p_0,\ldots,p_k,p_{k+1}]$ and $\mathscr{P}'_1=[p_{k+2},\ldots,p_m]$.

In order to introduce the formal procedure for the STL operator, we give below the assumptions under which this operator is performed.

[¶¶] The structure that we introduce here, namely the FBST, could also be combined with other structure models, such as Markov models, dictionary-based compression, PPM schemes, etc., to achieve much more efficient compression. In this paper, we consider the zeroth-order model. The use of FBSTs with higher-order models is currently being investigated.

[|||] The reader will observe that we have defined these operators in terms of various cases. This is, conceptually, similar to the zig-zig and zig-zag cases of the tree-based operations already introduced in the literature [11, 13, 20]. It is, of course, conceivable that we can include all the possible cases under a single umbrella, and then 'pick and choose' those which have to be used in each scenario, i.e. for the *STL* and the *STR* operators. However, we feel that although this would make the cases more compact, it would lead to a less elegant presentation. Indeed, it would occlude the real underlying process of what is actually taking place in the tree.

DOI: 10.1002/dac

Notation STL: Consider a PBST, $\mathscr{T}=\{t_1,\ldots,t_{2m-1}\}$, in which the weight of each node, t_l, is τ_l, and the key for each internal node is k_l, for $l=1,\ldots,m$. Let

- t_i be an internal node of $\mathscr{T}$,
- R_i be also an internal node of $\mathscr{T}$,
- t_j be the left-most leaf of the subtree rooted at R_i,
- B_j be the sibling of t_j, and
- t_k be the right-most leaf of the subtree rooted at L_i.

Using this notation, we can identify three mutually exclusive cases in which the STL operator can be applied. These cases are listed below, and the rules for performing the STL operation and the corresponding examples are discussed thereafter.

STL-1: $P_{P_j}=t_i$ and L_i is a leaf.
STL-2: $P_{P_j}\neq t_i$ and L_i is a leaf.
STL-3: L_i is *not* a leaf.
The STL operator performed in the scenario of Case STL-1 is discussed below.

Rule 1 (*STL-1*)
Consider a PBST, $\mathscr{T}$, described using *Notation STL*. Suppose that the scenario is that of Case STL-1. The STL operator applied to the subtree rooted at node t_i consists of the following operations:

(a) the value $\tau_k-\tau_{B_j}$ is added to τ_{P_j},
(b) k_i and k_{P_j} are swapped,***
(c) B_j becomes the right child of t_i,
(d) P_j becomes the left child of t_i,
(e) t_k becomes the left child of P_j, and
(f) t_j becomes the right child of P_j.

Remark 3
The node on which the STL operator is applied, t_i, can be any internal node or the root satisfying *Notation STL*. The tree resulting from the STL-1 operator is a PBST. This is stated for the operator, in general, in Lemma 2 given below for which the proof can be found in Appendix A.

Lemma 2 (*STL-1 validity*)
Consider a PBST, $\mathscr{T}=\{t_1,\ldots,t_{2m-1}\}$, specified as per *Notation STL*. If an STL operation is performed on the subtree rooted at node t_i as per Rule 1, then the resulting tree, $\mathscr{T}'=\{t'_1,\ldots,t'_{2m-1}\}$, is a PBST.

From the proof of Lemma 2, we see that the weights of the internal nodes in the new tree, $\mathscr{T}'$, are consistently obtained as the sum of the weights of their two children. This is achieved in only *two* local operations, as opposed to re-calculating *all* the weights of the tree in a bottom-up fashion.

***In the actual implementation, the FBST can be maintained in an array in which the node t_l, $1\leqslant l\leqslant 2m-1$, can be stored at position l. In this case, and in all the other cases of STL and STR, swapping these two keys could be avoided, and searching the node t_l could be done in a single access to position l in the array.

DOI: 10.1002/dac

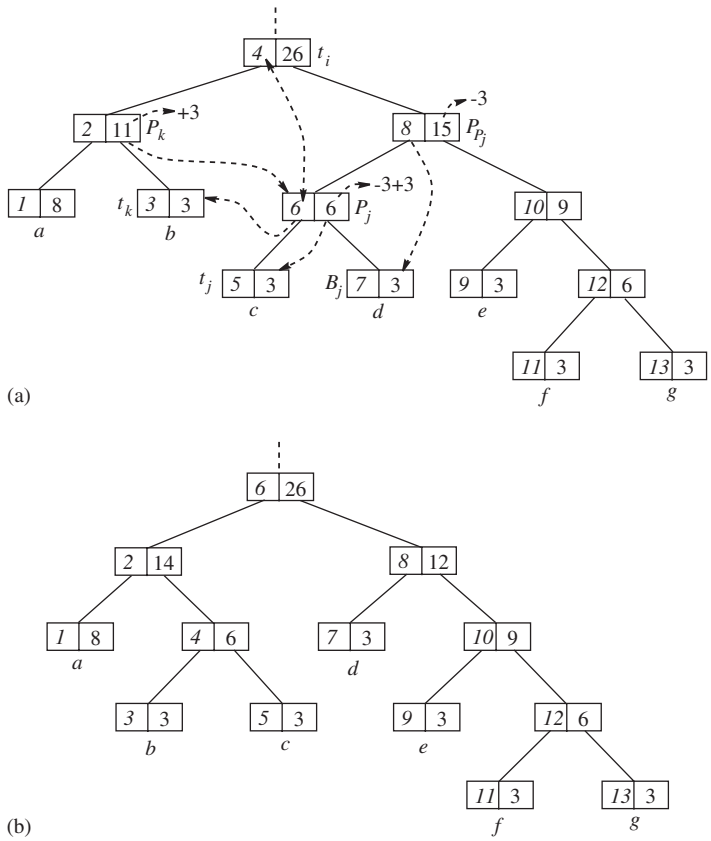

Figure 3. A PBST, $\mathscr{T}$, constructed from $\mathscr{S}=\{a,b,c,d,e,f,g\}$ and $\mathscr{P}=[8,3,3,3,3,3,3]$, and the corresponding PBST, $\mathscr{T}'$, after performing an STL-3 operation. The dotted line above the top-most node indicates that this node could be a left child or a right child of its parent or could be the root node itself. (a) The PBST, $\mathscr{T}$, before performing the STL (Case STL-3) operation and (b) the resulting tree, $\mathscr{T}'$, after the STL-3 operation is performed. $\mathscr{T}'$ is a PBST.

We now provide the mechanisms required to perform an STL operation when we are in the scenario of Case STL-2.

Rule 2 (*STL-2*)
Consider a PBST, $\mathscr{T}=\{t_1,\ldots,t_{2m-1}\}$, described using *Notation STL*. Suppose that we are in the scenario of Case STL-2. The STL operator performed on node t_i involves the following operations:

(a) $\tau_k-\tau_{B_j}$ is added to τ_{P_j},
(b) τ_j is subtracted from all the τ's in the path from P_{P_j} to R_i,
(c) k_i and k_{P_j} are swapped,
(d) B_j becomes the left child of P_{P_j},

DOI: 10.1002/dac

(e) t_j becomes the right child of P_j,
(f) t_k becomes the left child of P_j, and
(g) P_j becomes the left child of t_i.

Note that the resulting tree is a PBST. The general case is stated in Lemma 3 for which the proof can be found in Appendix A.

Lemma 3 (*STL-2 validity*)
Consider a PBST, $\mathscr{T} = \{t_1, \ldots, t_{2m-1}\}$, as per *Notation STL*. The resulting tree, $\mathscr{T}' = \{t'_1, \ldots, t'_{2m-1}\}$, obtained after performing an STL-2 operation as per Rule 2 is a PBST.

The corresponding rule for the scenario of Case STL-3 satisfying *Notation STL* is in Rule 3.

Rule 3 (*STL-3*)
Consider a PBST, $\mathscr{T} = \{t_1, \ldots, t_{2m-1}\}$, specified using the notation of *Notation STL*, and the scenario of Case STL-3. The STL operator performed on the subtree rooted at t_i consists of shifting t_j to the subtree rooted at L_i in such a way that

(a) $\tau_k - \tau_{B_j}$ is added to τ_{P_j},
(b) τ_j is subtracted from all the τ's in the path from P_{P_j} to R_i,
(c) τ_j is added to all the τ's in the path from P_k to L_i,
(d) k_i and k_{P_j} are swapped,
(e) B_j becomes the left child of P_{P_j},
(f) t_j becomes the right child of P_j,
(g) t_k becomes the left child of P_j, and
(h) P_j becomes the right child of P_k.

Observe that in the STL-3 operation, all the nodes in the entire path from P_k to the left child of t_i have to be updated by adding τ_j to the weight of those nodes. As in the other two cases, the weight of t_i is not changed. We show below an example that helps to understand how the STL-3 operator works.

Example 2
Let $\mathscr{S} = \{a, b, c, d, e, f, g\}$ be the source alphabet whose frequency counters are $\mathscr{P} = [8, 3, 3, 3, 3, 3, 3]$. A PBST, $\mathscr{T}$, constructed from $\mathscr{S}$ and $\mathscr{P}$ is the one depicted in Figure 3(a). After applying the STL operator to the subtree rooted at node t_i (in this case, the root node of $\mathscr{T}$), we obtain $\mathscr{T}'$, the tree depicted in Figure 3(b). Observe that $\mathscr{T}'$ is a PBST. The general result is stated in Lemma 4, whose proof can be found in Appendix A.

Lemma 4 (*STL-3 validity*)
Let $\mathscr{T} = \{t_1, \ldots, t_{2m-1}\}$ be a PBST in which an STL-3 operation is performed as per Rule 3, resulting in a new tree, $\mathscr{T}' = \{t'_1, \ldots, t'_{2m-1}\}$. Then, $\mathscr{T}'$ is a PBST.

In order to facilitate the implementation of the STL operator, we present the corresponding algorithm that considers the three mutually exclusive cases discussed above. This procedure is depicted in Algorithm *STL_Operation*. When performing an assignment operation by means of the left arrow, '$\leftarrow$', the operand on the left is the new value of the pointer or weight, and the operand on the right is either the value of the weight or the value of the actual pointer to a node.

DOI: 10.1002/dac

For example, $L_{P_{P_j}} \leftarrow B_j$ implies that the value of the pointer to the left child of P_{P_j} acquires the value 'B_j'.†††

Algorithm 2 STL_Operation

Input: A PBST, $\mathscr{T} = \{t_1, \ldots, t_{2m-1}\}$.
The node on which the STL is performed, t_i.
The left-most leaf of the subtree rooted at R_i, t_j.
The right-most leaf of the subtree rooted at L_i, t_k.
Output: The modified PBST, $\mathscr{T}$.
Assumptions: Those found in *Notation STL*.
Method:

```
procedure STL(var T : partitioningBST; t_i, t_j, t_k : node);
  k_i ↔ k_{P_j}   // swap keys
  τ_{P_j} ← τ_{P_j} + τ_k − τ_{B_j}
  for l ← P_{P_j} to R_i step l ← P_l do
    τ_l ← τ_l − τ_j
  endfor
  for l ← P_k to L_i step l ← P_l do
    τ_l ← τ_l + τ_j
  endfor
  if t_i = P_k then   // Move P_j to the subtree on the left
    L_i ← P_j   // STL-1 and STL-2
  else
    R_{P_k} ← P_j   // STL-3
  endif
  if t_i = P_{P_j} then   // B_j remains in the subtree on the right
    R_i ← B_j   // STL-1
  else
    L_{P_{P_j}} ← B_j   // STL-2 and STL-3
  endif
  R_{P_j} ← t_j   // t_j becomes the right child of its parent
  L_{P_j} ← t_k; P_k ← P_j   // P_j becomes the parent of t_k
  P_{B_j} ← P_{P_j}   // Update the parent of B_j
  P_{P_j} ← P_k   // The new parent of P_j is now the old parent of t_k
endprocedure
```

end Algorithm STL_Operation

3.2. The STR operator

The STL operator and the STR operator use similar principles for shifting nodes. Broadly speaking, the STR operator applied to an internal node, t_i, consists of removing the right-most leaf of the

†††In the actual implementation, 'B_j' contains the memory address of the node B_j or the position of B_j in a list if the tree is stored in an array.

DOI: 10.1002/dac

subtree rooted at the left child of t_i and inserting it as the left-most leaf into the subtree rooted at the right child of t_i.

Consider a list, $\mathscr{P}=[p_1,\ldots,p_m]$, which has been partitioned into two sublists, $\mathscr{P}_0=[p_1,\ldots,p_k]$ and $\mathscr{P}_1=[p_{k+1},\ldots,p_m]$. The STR operator applied to an internal node of a PBST is equivalent to removing the last element of $\mathscr{P}_0$ and inserting it as the first element of $\mathscr{P}_1$. After applying the STR operation, the resulting sublists are $\mathscr{P}'_0=[p_1,\ldots,p_{k-1}]$ and $\mathscr{P}'_1=[p_k,p_{k+1},\ldots,p_m]$.

We now introduce the notation for the STR operator.

Notation STR: Consider a PBST, $\mathscr{T}=\{t_1,\ldots,t_{2m-1}\}$. Let

- t_i be an internal node of $\mathscr{T}$,
- L_i be also an internal node of $\mathscr{T}$,
- t_j be the right-most leaf of the subtree rooted at L_i,
- B_j be the sibling of t_j, and
- t_k be the left-most leaf of the subtree rooted at R_i.

As in the STL operator, we identify three mutually exclusive cases in which the STR operator can be applied.

STR-1: $P_{P_j}=t_i$ and R_i is a leaf.

STR-2: $P_{P_j}\neq t_i$ and R_i is a leaf.

STR-3: R_i is *not* a leaf.

We again state the rules for performing the STR operation for the three cases listed above. The STR operator performed in the scenario of Case STR-1 is formalized below.

Rule 4 (*STR-1*)
Let $\mathscr{T}=\{t_1,\ldots,t_{2m-1}\}$ be a PBST specified using *Notation STR* and the scenario of Case STR-1. The STR operator applied to the subtree rooted at node t_i involves the following operations:

(a) $\tau_k-\tau_{B_j}$ is added to τ_{P_j},
(b) k_i and k_{P_j} are swapped,
(c) B_j becomes the left child of t_i,
(d) P_j becomes the right child of t_i,
(e) t_k becomes the right child of P_j, and
(f) t_j becomes the left child of P_j.

Remark 4
The node t_i can be any internal node of $\mathscr{T}$ (not necessarily the root) that is specified by *Notation STR*. The next example includes an STR operation performed on an internal, non-root node, t_i, in the scenario of Case STR-1.

After performing the STR operation, the resulting tree, $\mathscr{T}'$, is a PBST. This is a general result, which is stated in Lemma 5 for which the proof is given in Appendix A.

Lemma 5 (*STR-1 validity*)
Let $\mathscr{T}=\{t_1,\ldots,t_{2m-1}\}$ be a PBST. If $\mathscr{T}'=\{t'_1,\ldots,t'_{2m-1}\}$ is the resulting tree obtained after performing an STR-1 operation on t_i, then $\mathscr{T}'$ is a PBST.

The formal definition of the operations required to perform an STR operation for the scenario of the second case, STR-2, is formalized below.

DOI: 10.1002/dac

Rule 5 (*STR-2*)
Consider a PBST, $\mathscr{T}=\{t_1,\ldots,t_{2m-1}\}$, specified as per *Notation STR*, and the scenario of Case STR-2. The STR operator applied on t_i involves the following operations:

(a) $\tau_k-\tau_{B_j}$ is added to τ_{P_j},
(b) τ_j is subtracted from all the τ's from P_{P_j} to L_i,
(c) k_{P_j} and k_i are swapped,
(d) B_j becomes the right child of P_{P_j},
(e) P_j becomes the right child of t_i,
(f) t_j becomes the left child of P_j, and
(g) t_k becomes the right child of P_j.

The tree, $\mathscr{T}'$, produced by applying the STR-2 operator is a PBST. This result is stated in Lemma 6, whose proof can be found in Appendix A.

Lemma 6 (*STR-2 validity*)
Consider a PBST, $\mathscr{T}=\{t_1,\ldots,t_{2m-1}\}$, specified as per *Notation STR*. An STR-2 operation on t_i produces a PBST, $\mathscr{T}'=\{t'_1,\ldots,t'_{2m-1}\}$.

The last case that we consider for performing STR operations on PBSTs is defined below.

Rule 6 (*STR-3*)
Consider a PBST, $\mathscr{T}=\{t_1,\ldots,t_{2m-1}\}$, described using *Notation STR*, and the scenario of Case STR-3. Applying an STR operator on t_i consists of the following operations:

(a) $\tau_k-\tau_{B_j}$ is added to τ_{P_j},
(b) τ_j is subtracted from all the τ's from P_{P_j} to L_i,
(c) τ_j is added to all the τ's from P_k to R_i,
(d) k_i and k_{P_j} are swapped,
(e) B_j becomes the right child of P_{P_j},
(f) t_j becomes the left child of P_j,
(g) t_k becomes the right child of P_j, and
(h) P_j becomes the left child of P_k.

The next example depicts how the STR operator works in the scenario of Case STR-3.

Example 3
Let $\mathscr{S}=\{a,b,c,d,e,f,g\}$ be the source alphabet whose frequency counters are $\mathscr{P}=[3,1,1,1,1,1,1]$. Let $\mathscr{T}$ be the PBST shown in Figure 4(a). Suppose that we perform an STR-3 operation on node t_i. The changes to $\mathscr{T}$ are marked with dashed arcs, and the resulting tree is the one shown in Figure 4(b). The resulting tree is a PBST. This is a general result, which is stated in Lemma 7 for which the proof is given in Appendix A.

Lemma 7 (*STR-3 validity*)
Consider a PBST, $\mathscr{T}=\{t_1,\ldots,t_{2m-1}\}$, specified as per *Notation STR*, and the scenario of Case STR-3. The tree $\mathscr{T}'=\{t'_1,\ldots,t'_{2m-1}\}$, which results from applying an STR operator on t_i, is a PBST.

The implementation of the three cases in which the STR operator can be applied is included in Algorithm *STR_Operation*. This algorithm contains the procedure STR(...), which takes a PBST,

DOI: 10.1002/dac

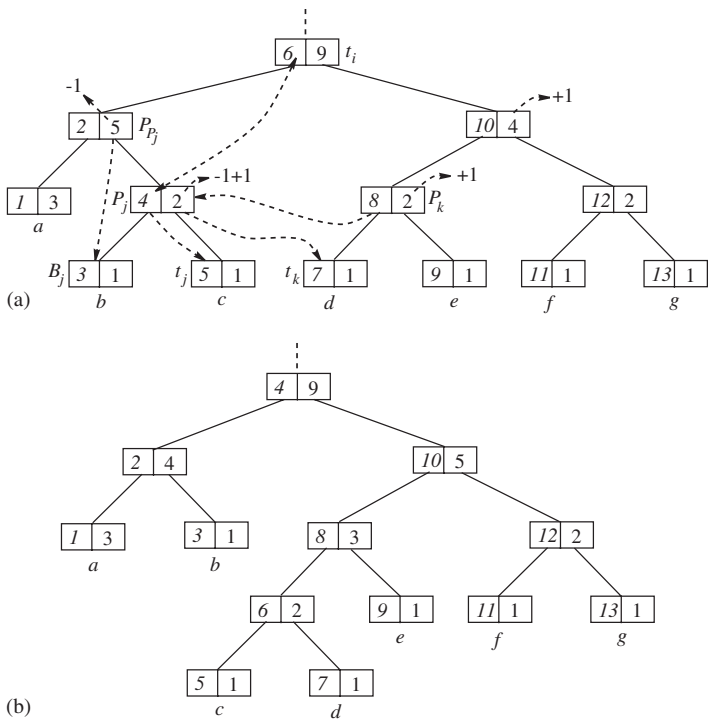

Figure 4. Two PBSTs. The tree on the top was constructed from $\mathscr{S}=\{a,b,c,d,e,f,g\}$ and $\mathscr{P}=[3,1,1,1,1,1,1]$ and includes the changes to be done when applying an STR-3 operation on t_i. The tree on the bottom is the resulting PBST after the STR-3 operation. The dotted line above the top-most node indicates that this node could be a left child or a right child of its parent or could be the root node itself. (a) The PBST before performing the STR-3 operation, the dashed arcs indicate the changes to be done by the STR operator and (b) the resulting PBST after the STR-3 operation.

$\mathscr{T}$, and a node on which the STR operation is performed, generating a new PBST. We refer to the new PBST as $\mathscr{T}$, and we use the modifier var so as to return the new tree.

4. FANO BST CODING

Using the PBST and the underlying tree operations (discussed in Section 3), we now apply them to the adaptive data encoding problem. Given an input sequence, $\mathscr{X}=x[1]\ldots x[M]$, which has to be encoded, the idea is to maintain an FBST at all times—at the encoding and decoding stages. In the encoding algorithm, at time 'k', the symbol $x[k]$ is encoded using an FBST, $\mathscr{T}(k)$, which is identical to the one used by the decoding algorithm to retrieve $x[k]$. $\mathscr{T}(k)$ must be updated in such a way that at time '$k+1$', both algorithms maintain the same tree $\mathscr{T}(k+1)$.

To maintain, at each time instant, an FBST, i.e. the tree obtained after the updating procedure, $\mathscr{T}(k+1)$, must satisfy the conditions stated in Definition Structure_FBST. Since the PBST

DOI: 10.1002/dac

Algorithm 3 STR_Operation

Input: A PBST, $\mathscr{T} = \{t_1, \ldots, t_{2m-1}\}$.
The node on which the STR is performed, t_i.
The right-most leaf of the subtree rooted at L_i, t_j.
The left-most leaf of the subtree rooted at R_i, t_k.
Output: The modified PBST, $\mathscr{T}$.
Assumptions: Those in *Notation STR*.
Method:

```
procedure STL(var T : partitioningBST; t_i, t_j, t_k : node);
  k_i ↔ k_{P_j}  // swap keys
  τ_{P_j} ← τ_{P_j} + τ_k − τ_{B_j}
  for l ← P_{P_j} to L_i step l ← P_l do
    τ_l ← τ_l − τ_j
  endfor
  for l ← P_k to R_i step l ← P_l do
    τ_l ← τ_l + τ_j
  endfor
  if t_i = P_k then  // Move P_j to the subtree on the right
    R_i ← P_j  // STR-1 and STR-2
  else
    L_{P_k} ← P_j  // STR-3
  endif
  if t_i = P_{P_j} then  // B_j remains in the subtree on the right
    L_i ← B_j  // STR-1
  else
    R_{P_{P_j}} ← B_j  // STR-2 and STR-3
  endif
  L_{P_j} ← t_j  // t_j becomes the left child of its parent
  R_{P_j} ← t_k; P_k ← P_j  // P_j becomes the parent of t_k
  P_{B_j} ← P_{P_j}  // Update the parent of B_j
  P_{P_j} ← P_k  // The new parent of P_j is now the old parent of t_k
endprocedure
```

end Algorithm STR_Operation

structures are maintained at both the encoder and decoder sides, it is up to the updating procedure to ensure that the resulting tree satisfies the conditions stated in Definition Structure_FBST.

The updating procedure is based on a *conditional shifting heuristic* and used to transform a PBST into an FBST. The conditional shifting heuristic is based on the principles of the Fano coding—the nearly equal-probability property [39]. This heuristic used in conjunction with the STL and the STR operators defined in this paper are used to transform a PBST into an FBST, as per the following rule.

Rule 7
Consider a partitioning BST, $\mathscr{T} = \{t_1, \ldots, t_{2m-1}\}$. Let t_i be an internal node of $\mathscr{T}$, t_j be the left-most leaf of the subtree rooted at R_i, and t_k be the right-most leaf of the subtree rooted at L_i.

DOI: 10.1002/dac

(i) If

$$\theta_1 = \tau_{R_i} - \tau_{L_i} - \tau_j > 0 \tag{7}$$

then perform an STL operation on t_i.

(ii) If

$$\theta_2 = \tau_{L_i} - \tau_{R_i} - \tau_k > 0 \tag{8}$$

then perform an STR operation on t_i.

We now introduce a definition that is important in the formalization of the algorithms that implements the updating procedure. We let a *two-leaf internal node* be an internal node whose left and right children are leaves.

Definition 1
Let $\mathscr{T} = \{t_1, \ldots, t_{2m-1}\}$ be a partitioning BST. An internal node, t_i, is a *two-leaf internal node* if and only if L_i and R_i are both leaves.

The procedure for updating the Fano BST, procedure updateFanoBST(...), is formalized in Algorithm *Fano_BST_Updating*. This procedure receives as parameters a partitioning BST, $\mathscr{T} = \{t_1, \ldots, t_{2m-1}\}$, the root node, *root*, and the node associated with the symbol being encoded, t_n. The weight of t_n at time 'k', $\tau_n(k)$, is updated by adding unity to it. In order to maintain the source symbols of $\mathscr{T}$ in a non-increasing order of probability from left to right, t_n is swapped with the left-most leaf (if there is any) whose weight is less than $\tau_n(k)$. Let $\hat{\mathscr{P}}(k) = \{\hat{p}_1(k), \ldots, \hat{p}_m(k)\}$ be the estimated probabilities of $\mathscr{S}$, and $\eta(k) = \{\eta_1(k), \ldots, \eta_m(k)\}$ be the frequency counters of $\mathscr{S}$. In order to avoid duplicating operations, $\hat{\mathscr{P}}(k)$ is not updated. In fact, incrementing the weight of t_n by unity is equivalent to incrementing $\eta_i(k)$ by unity and consequently updating $\hat{p}_i(k) = \eta_i(k)/\eta + k = \tau_n(k)/\eta + k$, where s_i is the symbol associated with t_n, and $\eta = \sum_{i=1}^{m} \eta_i(1)$. We show later in this section that using this updating procedure, the adaptive Fano BST asymptotically converges to the static Fano tree.

The weights of all the internal nodes in the path from P_n to the root are updated. After this, the path from the root downwards t_n is inspected to see if there is a non-two-leaf internal node that does not satisfy the conditions stated in Definition 2. This is achieved by invoking the procedure checkShift(...) explained below. The path from the root downwards t_n is traced by using the key of each internal node in such a way that the key of t_n, k_n, is searched as in a BST.

The procedure checkShift(...) of Algorithm *Fano_BST_Updating* is responsible for checking that all the non-two-leaf internal nodes satisfy Definition 2. This procedure receives as parameters the partitioning BST, $\mathscr{T}$, and the non-two-leaf internal node to be inspected, t_i. By following Rule 7, if (7) is true (i.e. $\theta_1 > 0$), an STL operator is performed on t_i, and the procedure checkShift(...) is recursively invoked for the left and right children of t_i. If $\theta_1 \leqslant 0$, (8) is evaluated, and then if $\theta_2 > 0$, an STR operation is performed on t_i, and, as in the case of the STL operation, the procedure checkShift(...) is recursively invoked for the left and right children of t_i. After the procedure checkShift(...) is executed on all the non-two-leaf internal nodes of the path from the root to the leaf associated with $x[k]$, t_n, the partitioning BST is transformed into a Fano BST.

The encoding procedure that uses the Fano BST data structure is given in Algorithm *Fano_BST_Encoding*. Let $\mathscr{T}(k)$ be the Fano BST at time 'k', which, in the algorithm, is represented by root. The encoding process proceeds, as usual, by scanning all the symbols of $\mathscr{X}$ from left to

DOI: 10.1002/dac

Algorithm 4 Fano_BST_Updating

Input: A partitioning BST, $\mathscr{T} = \{t_1, \ldots, t_{2m-1}\}$. The root of $\mathscr{T}$, root.
The leaf node associated with $x[k]$ whose weight has to be updated, t_n.
Output: A Fano BST, $\mathscr{T}$.
Method:

```
procedure updateFanoBST(T : partitioningBST; root, t_n : node);
  τ_n ← τ_n + 1  // Increment the weight of t_n
  Swap t_n with the left-most leaf, t_v, where τ_n > τ_v
  t_i ← t_n
  while t_i ≠ root do  // Walk up towards the root
    t_i ← P_i
    τ_i ← τ_i + 1  // Update the internal node weights
  endwhile
  while k_i ≠ k_n do  // Walk downwards t_n
    checkShift(T, t_i)
    if k_n < k_i then
      t_i ← L_i  // Go to the left child
    else
      t_i ← R_i  // Go to the right child
    endif
  endwhile
endprocedure
procedure checkShift(T : partitioningBST; t_i : node);
  if t_i ≠ 'two-leaf internal node' then return endif
  t_j ← left-most leaf of the subtree rooted at R_i
  t_k ← right-most leaf of the subtree rooted at L_i
  while τ_{R_i} − τ_{L_i} − τ_j > 0 do  // θ_1 > 0
    STL(T, t_i, t_j, t_k)
    checkShift(T, L_i)
    checkShift(T, R_i)
  endwhile
  while τ_{L_i} − τ_{R_i} − τ_k > 0 do  // θ_2 > 0
      STR(T, t_i, t_k, t_j)  // In the STR, t_j and t_k interchange roles
      checkShift(T, L_i)
      checkShift(T, R_i)
  endwhile
endprocedure
end Algorithm Fano_BST_Updating
```

right. At time 'k', i is obtained as the index of $x[k]$ in $\mathscr{S}$. By taking advantage of Property (i) of a Fano BST, $2i-1$ is searched in $\mathscr{T}(k)$, as if it were searched in a BST: Going to the left child when $2i-1 < k_n$ or to the right child otherwise. The value $2i-1$ is always found since, as mentioned earlier, we assume that all the keys are already in the BST. Besides when going to the left child, '0' is sent to the output, and when going to the right child, '1' is sent to the output. Any of the 2^{m-1} labeling strategies other than the one used here can also be used. Once the kth symbol from the input is encoded, $\mathscr{T}(k)$ must be updated. The updating procedure, updateFanoBST(...),

DOI: 10.1002/dac

involves two phases: increment the frequency counter of s_i, $\eta_i(k)$ and make the necessary changes in such a way that $\mathscr{T}(k+1)$ is a Fano BST.

Algorithm 5 Fano_BST_Encoding

Input: The source alphabet, $\mathscr{S}$. A source sequence, $\mathscr{X} = x[1] \ldots x[M]$.
Output: The encoded sequence $\mathscr{Y} = y[1] \ldots y[R]$.
Assumptions: The Fano BST is constructed by invoking FanoBST(...), and maintained correctly by invoking updateFanoBST(...).
Method:

```
η_i(1) ← 1 for i = 1, ..., m
Create node root
η(1) ← Σ_{i=1}^{m} η_i(1); τ_root ← Σ_{i=1}^{m} η_i(1)
FanoBST(S, η(1), 1, m, root, τ_root)
j ← 1
for k ← 1 to M do
   i ← position of x[k] in T from left to right (counting the leaves only)
   t_n ← root  // For each symbol, start again from the root
   while t_n ≠ 'leaf' do
      if 2i − 1 < k_n then  // Perform a binary search
         y[j] ← '0'; t_n ← L_n // Go to left child
      else
         y[j] ← '1'; t_n ← R_n // Go to right child
      endif
      j ← j + 1
   endwhile
   updateFanoBST(T, root, t_n)
endfor
end Algorithm Fano_BST_Encoding
```

The decoding procedure, given in Algorithm *Fano_BST_Decoding*, works as follows. Let $\mathscr{T}(k)$ be the Fano BST used to decode $x[k]$. In the algorithm, this tree is represented by the variable root. The decoding procedure proceeds by scanning the encoded sequence, $\mathscr{Y}$, from left to right. An auxiliary pointer, n, is maintained, which stores the pointer to the current node being inspected at time 'j'. Each time a bit from the input, $y[j]$, is received, the pointer t_n is moved to the left child of t_n, L_n, if $y[j]$ is a '0', or to the right child of t_n, R_n, if $y[j]$ is a '1'. When a leaf node is reached, the symbol associated with it, $s_{(n+1)/2}$, is recovered and sent to the output. At this point the tree is immediately updated by invoking the procedure updateFanoBST(...), discussed earlier, so that a Fano BST is maintained, which is the one used to decode the next source symbol, $x[k+1]$.

5. EMPIRICAL RESULTS

To analyze the speed and compression efficiency of our newly introduced adaptive coding scheme, we report the results obtained after running the scheme on files of the Calgary and Canterbury corpora. We also ran the greedy adaptive Fano coding presented in [36] and the AHC algorithm

DOI: 10.1002/dac

Algorithm 6 Fano_BST_Decoding

Input: The source alphabet, $\mathscr{S}$. The encoded sequence $\mathscr{Y} = y[1] \ldots y[R]$.
Output: The original source sequence, $\mathscr{X} = x[1] \ldots x[M]$.
Assumptions: The Fano BST is constructed by invoking FanoBST(...), and maintained correctly by invoking updateFanoBST(...).
Method:

```
η_i(1) ← 1 for i = 1, ..., m
Create node root
η(1) ← Σ_{i=1}^{m} η_i(1); τ_root ← Σ_{i=1}^{m} η_i(1)
FanoBST(S, η(1), 1, m, root, τ_root)
k ← 1
t_n ← root
for j ← 1 to R do
   if y[j]='0' then   // Equivalent to the binary search done in the encoder
      t_n ← L_n // Go to left child
   else
      t_n ← R_n // Go to right child
   endif
   if t_n = 'leaf' then
      x[k] ← s_{(n+1)/2} // The symbol associated with t_n is retrieved
      k ← k+1
      updateFanoBST(T, root, t_n)
   endif
endfor
end Algorithm Fano_BST_Decoding
```

introduced in [27] on the same benchmarks. In the subsequent tables, the first column contains the name of the file. The second column, labeled $l_{\mathscr{X}}$, represents the size (in bytes) of the original file. The next columns correspond to the speed and compression ratio for AHC, the *adaptive greedy Fano coding* (GFC), and the adaptive Fano coding that uses FBSTs (FBSTC). Each of the two groups of three columns contains the compression ratio, ρ, calculated as $\rho = (1 - \ell_{\mathscr{Y}}/l_{\mathscr{X}})100$, the time (in seconds) required to compress the file, and the time (in seconds) needed to recover the original file.

The results for the tests on the Calgary corpus are shown in Table I. From the weighted average (the row labeled 'Total'), we observe that GFC compresses slightly more (but only 0.04%) than FBSTC. In fact, this behavior is reasonable since they are expected to achieve the same compression ratio, as they use the principles of the Fano coding implemented with different structures. The slight variations are due to the way in which the 'ties' are broken while constructing the Fano codes. In all files, the compression ratios achieved by GFC and FBSTC are slightly lower than AHC. In terms of compression speed, GFC is faster than FBSTC on small files, namely *obj1*, *paper1*, *progc*, *progl*, and *progp*. On large files, FBSTC performs faster than GFC. This is expected, since for large files less changes are expected in the Fano BST. For all the files, FBSTC performs much faster than AHC, duplicating the compression speed in most of the cases. In the decompression stage, GFC is marginally faster than FBSTC, mainly in the small files mentioned above. In the decompression stage, FBSTC, however, is slower than AHC for all files except *progp*.

DOI: 10.1002/dac

Table I. Speed and compression ratio for the Fano BST coding, the greedy adaptive Fano method, and the adaptive Huffman coding, which were tested on files of the Calgary corpus.

		AHC			GFC			FBSTC		
File name	$l_{\mathcal{X}}$ (bytes)	ρ (%)	T. enc. (s)	T. dec. (s)	ρ (%)	T. enc. (s)	T. dec. (s)	ρ (%)	T. enc. (s)	T. dec. (s)
bib	111 261	34.35	2.51	0.41	34.21	1.31	0.86	34.21	1.32	1.24
book1	768 771	42.92	5.92	2.62	42.83	6.97	5.10	42.84	4.00	3.55
book2	610 856	39.64	5.62	2.07	39.47	6.04	4.41	39.48	4.14	3.81
geo	102 400	28.97	2.53	0.47	28.66	2.24	0.85	28.65	1.54	1.47
news	377 109	34.57	4.20	1.44	34.45	4.58	2.88	34.46	2.76	2.57
obj1	21 504	24.67	1.92	0.12	24.20	0.49	0.20	23.96	0.86	0.80
obj2	246 814	21.26	3.63	1.16	21.21	5.39	2.43	21.16	2.16	2.07
paper1	53 161	36.82	2.03	0.19	36.80	0.57	0.38	36.80	1.06	1.00
progc	39 611	34.00	1.93	0.16	33.64	0.50	0.28	33.62	1.00	0.93
progl	71 646	39.64	2.17	0.25	39.20	0.71	0.52	39.21	1.65	1.59
progp	49 379	38.29	2.04	0.18	37.80	0.54	0.35	37.80	1.16	1.13
trans	93 695	30.10	2.43	0.37	30.04	1.41	0.77	30.03	1.11	1.06
Total	2 547 207	36.82	36.93	9.44	36.68	30.75	19.03	36.64	22.76	21.22

Table II. Speed and compression ratio obtained after running the adaptive Huffman coding, the greedy adaptive Fano coding, and the adaptive coding that uses FBST on files of the Canterbury corpus.

		AHC			GFC			FBSTC		
File name	$l_{\mathcal{X}}$ (bytes)	ρ (%)	T. enc. (s)	T. dec. (s)	ρ (%)	T. enc. (s)	T. dec. (s)	ρ (%)	T. enc. (s)	T. dec. (s)
alice29.txt	148 481	42.84	2.49	0.48	42.59	1.20	0.99	42.59	1.44	1.33
asyoulik.txt	125 179	39.21	2.37	0.42	39.05	1.25	0.89	39.06	1.31	1.23
cp.html	24 603	33.27	1.87	0.10	33.19	0.30	0.19	33.19	0.88	0.82
fields.c	11 150	35.31	1.77	0.05	34.92	0.14	0.08	34.89	0.52	0.49
grammar.lsp	3721	37.70	1.70	0.02	37.29	0.05	0.02	37.24	0.32	0.29
kennedy.xls	1 029 744	55.04	6.71	2.90	54.66	11.72	5.76	54.81	4.50	3.86
lcet10.txt	419 235	41.74	4.18	1.41	41.72	3.83	2.84	41.72	2.41	2.19
plrabn12.txt	471 162	43.43	4.40	1.52	43.32	4.20	3.08	43.32	2.33	2.16
ptt5	513 216	79.18	2.57	0.67	79.16	1.68	1.29	79.16	1.86	1.60
sum	38 240	32.48	1.93	0.18	31.50	0.66	0.30	31.42	0.92	0.88
xargs.1	4227	34.77	1.74	0.02	34.81	0.06	0.03	34.79	0.42	0.38
Total	2 788 958	53.53	31.73	7.77	53.33	25.09	15.47	53.38	16.91	15.23

The results for the tests on the Canterbury corpus are shown in Table II. As in the tests for the Calgary corpus, the compression ratios for GFC and FBSTC are quite similar, as expected. We observe that the FBSTC obtains compression ratios slightly higher (only 0.15%) than AHC. In terms of compression speed, on small files, GFC is faster than FBSTC, and the latter is significantly faster than AHC. On large files (e.g. *kennedy.xls*), however, FBSTC is faster than GFC. In fact, the reason why GFC is faster on *ptt5* is due to the fact that a high compression ratio is achieved, and the input file contains only a few different symbols, which makes the FBST look like a list. Consequently, its behavior is similar to that of GFC, with the additional burden of maintaining

DOI: 10.1002/dac

a complex structure, the FBST. Observe that to compress the file *kennedy.xls*, GFC takes more than 2.5 times as that of FBSTC. In terms of decompression speed, FBSTC is slower than AHC. However, the former achieves similar speed values for both compression and decompression, which implies an advantage when the entire process has to be synchronized.

6. CONCLUSIONS

In this paper, we have demonstrated that we can effectively use results from the field of adaptive *self-organizing* data structures in enhancing compression schemes. Unlike *adaptive* lists, which have already been used in compression, to the best of our knowledge, adaptive *self-organizing* trees have not been used in this regard.

To achieve this, we have introduced a new data structure, the *partitioning binary search tree* (PBST) which, although based on the well-known *binary search tree* (BST), also appropriately partitions the data elements into mutually exclusive sets. The PBST is a BST in which the source alphabet symbols are incorporated in such a way that their location can be determined using their indices as the 'keys'. When used in conjunction with a Fano encoding, we have shown that the PBST leads to the so-called *Fano binary search tree* (FBST), which also incorporates the required Fano coding (nearly equal probability) property into the BST, and have given detailed algorithms to demonstrate how both the PBST and FBST can be maintained adaptively. In order to maintain a PBST while encoding, we have introduced the updating procedure that performs the two new *tree*-based operators, namely the shift-to-left (STL) operator and the shift-to-right (STR) operator. For these two operators, we have identified all the mutually exclusive cases in which they can be applied, and provided the formal rules that implement these operators, as well as the respective scenarios for their validity.

The encoding and decoding procedures that also update the FBST have been implemented and rigorously tested. Our empirical results on files of the Calgary and Canterbury corpora show the salient advantages of our strategy. An open problem that deserves investigation is that of combining the adaptive self-organizing BST methods and other statistical and/or dictionary-based methods, which undoubtedly would lead to more efficient compression schemes implied by the higher-order models. The resulting advantage can be obtained as a consequence of the additional speed-enhanced updating procedure provided by the adaptive self-organizing *tree*-based principles.

APPENDIX A: PROOFS

Proof of Lemma 1

By using (1), the total number of accesses to the subtree rooted at t_{2i} is calculated as follows:

$$\tau_{2i} = \sum_{j=1}^{2s-1} \alpha_j \tag{A1}$$

The first equality holds because

$$\tau_{2i} = \sum_{j=1}^{s-1} \alpha_{2j} + \sum_{j=1}^{s} \alpha_{2j-1} = \sum_{j=1}^{s} \alpha_{2j-1} \tag{A2}$$

the last step being a consequence of the fact that $\alpha_{2j} = 0$ for all j's.

DOI: 10.1002/dac

The second equality of (5) uses Lemma 1 of [19], whence,

$$\tau_{2i} = \alpha_{2i} + \tau_{L_{2i}} + \tau_{R_{2i}} \tag{A3}$$

The result again follows by invoking the property $\alpha_{2i} = 0$, which is true for all i's. The lemma is thus proved. □

Proof of Theorem 1

From Lemma 1, we know that $\tau_{\text{root}} = \sum_{j=1}^{m} \alpha_{2j-1}$. Since, from Remark 2, for every internal node, $\alpha_{2i} = 0$, we can write (2) for κ_{root} as follows:

$$\kappa_{\text{root}} = \sum_{i=1}^{m-1} \alpha_{2i}\lambda_{2i} + \sum_{i=1}^{m} \alpha_{2i-1}\lambda_{2i-1} = \sum_{i=1}^{m} \alpha_{2i-1}\lambda_{2i-1} \tag{A4}$$

Consider the ratio $\kappa_{\text{root}}/\tau_{\text{root}}$:

$$\frac{\kappa_{\text{root}}}{\tau_{\text{root}}} = \sum_{i=1}^{m} \frac{\alpha_{2i-1}}{\sum_{j=1}^{m} \alpha_{2j-1}} \lambda_{2i-1} \tag{A5}$$

By invoking (4), we have $p_i = \alpha_{2i-1}/\sum_{j=1}^{m} \alpha_{2j-1}$, and hence (A5) can be written as follows:

$$\frac{\kappa_{\text{root}}}{\tau_{\text{root}}} = \sum_{i=1}^{m} p_i \lambda_{2i-1} = \sum_{i=1}^{m} p_i(\ell_i + 1) = \bar{\ell} + 1 \tag{A6}$$

The last equality is a consequence of the fact that $\lambda_{2i-1} = \ell_i + 1$, which is true because λ_i involves counting the *nodes* along the path from the root to the leaf associated with s_i, and ℓ_i is obtained by counting the *edges* in the corresponding path. □

Proof of Lemma 2

To clarify the notation of the nodes involved, t_i is an internal node, whose left child, t_k, has a sibling that is the parent of t_j, and additionally, t_k and t_j are adjacent leaves.

We have to show that $\mathcal{T}'$, the tree obtained after invoking Rule 1, satisfies Definition Structure_PBST. There are three issues that must be proven:

(a) Each leaf node of $\mathcal{T}'$ is at position $2u-1$, $1 \leqslant u \leqslant m$.
(b) Each internal node of $\mathcal{T}'$ is at position $2u$, $1 \leqslant u \leqslant m$.
(c) The weight of each internal node, τ'_{2u}, in $\mathcal{T}'$, must be obtained as $\tau_{L_{2u}} + \tau_{R_{2u}}$, where L_{2u} and R_{2u} are the left and right children of t_{2u}, respectively.

Without loss of generality, let t_i be t_{2u}, for some integer u, $1 \leqslant u \leqslant m$. This implies that $t_k = t_{2u-1}$ in $\mathcal{T}$. Besides $P_{P_j} = t_i$, which implies that $R_i = P_j$, and hence the next node to be enumerated after t_{2u} in $\mathcal{T}$ is $t_j = t_{2u+1}$. As a result, $P_j = t_{2u+2}$ is the next node to be enumerated after t_j in $\mathcal{T}$. In summary: $\mathcal{T} = \{t_1, \ldots, t_{2m-1}\}$, where $t_{2u-1} = t_k$, $t_{2u} = t_i$, $t_{2u+1} = t_j$, and $t_{2u+2} = P_j$.

(a) *Identity of leaf nodes*: After performing the STL-1 operation on t_i, P_j will become the left child of t_i, which implies that P_j will continue to be an internal node, since t_k and t_j become its left and right children, respectively. Since t_k and t_j are leaves, they will correspond to t'_{2u-1} and t'_{2u+1} in $\mathcal{T}'$, the modified tree, respectively. Additionally, all the other leaves, $\{t'_1, \ldots, t'_{2u-3}, t'_{2u+3}, \ldots, t'_{2m-1}\}$, remain unchanged in $\mathcal{T}'$, and hence (i) of Definition Structure_PBST will be satisfied.

DOI: 10.1002/dac

(b) *Identity of internal nodes*: Since P_j becomes the left child of t_i in $\mathscr{T}'$, P_j and t_i will be t'_{2u} and t'_{2u+2}, respectively in $\mathscr{T}'$. Besides all the other internal nodes of $\mathscr{T}'$, $\{t'_2, \ldots, t'_{2u-2}, t'_{2u+4}, \ldots, t_{2m-2}\}$, remain unchanged, and hence (ii) of Definition Structure_PBST is satisfied.

(c) *Updating of* τ: On the other hand, $\tau_{P_j} = \tau'_{2u}$ is the only weight that is modified in $\mathscr{T}'$. Since t_k becomes the left child of P_j, τ_k is added to τ_{P_j} (this is done in (a) of STL-1). Additionally, after performing the STL, t_j remains as a child of P_j, and B_j is not a child of P_j anymore. Consequently, since τ_{B_j} is subtracted in (a) of STL-1, τ_{P_j} becomes the sum of $\tau_k = \tau_{2u-1}$ and $\tau_j = \tau_{2u+1}$, and the updated value of τ'_{P_j} is

$$\tau'_{P_j} \leftarrow \tau_{P_j} + \tau_k - \tau_{B_j}$$

The remaining weights of $\mathscr{T}'$, $\{\tau'_1, \ldots, \tau'_{2u-2}, \tau'_{2u+2}, \ldots, \tau'_{2m-2}\}$, are unchanged, and hence all the properties of Definition Structure_PBST are satisfied. □

The result follows.

Proof of Lemma 3

The proof of this lemma follows the steps of that of Lemma 2.

As in the proof of Lemma 2, we have to show that $\mathscr{T}'$, the tree obtained after invoking Rule 2, satisfies the properties found in Definition Structure_PBST. We again assume that $t_{2u-1} = t_k$, $t_{2u} = t_i$, $t_{2u+1} = t_j$, and $t_{2u+2} = P_j$. We have to prove the following.

(a) *Identity of leaf nodes*: This part of the proof is identical to (a) of Lemma 2.

(b) *Identity of internal nodes*: In this case, the proof follows the exact same steps of (b) of Lemma 2.

(c) *Updating of* τ: The updating of τ_{P_j} is achieved, as shown in (c) of Lemma 2. We prove the consistency of the operation.

First of all, note that after invoking STL-2, P_j becomes the left child of t_i in $\mathscr{T}'$, and t_j becomes the right child of P_j. This implies that t_j is in the subtree rooted at L_i, and consequently, τ_j must be *subtracted* from the weights of *all* the nodes in the path from P_{P_j} to R_i. This is done in Step (b) of Rule STL-2. Since t_j is not in the subtree rooted at R_i in $\mathscr{T}'$ (after the rotation), we see from (A2) that $\tau_{2u} = \sum_{v=1}^{s} \tau_{2v-1}$, and hence all the nodes in that path satisfy $\tau'_{2u} = \tau'_{L_{2u}} + \tau'_{R_{2u}}$. □

Consequently, $\mathscr{T}'$ satisfies the properties of Definition Structure_PBST, and the lemma follows.

Proof of Lemma 4

The proof of this lemma follows the steps of that of Lemma 2.

As in the proof of Lemma 2, we have to show that $\mathscr{T}'$, the tree obtained after invoking Rule STL-3 satisfies the properties of Definition Structure_PBST. We again assume the notation that $t_{2u-1} = t_k$, $t_{2u} = t_i$, $t_{2u+1} = t_j$, and $t_{2u+2} = P_j$ (see Figure 3). We have to prove the following.

(a) *Identity of leaf nodes*: This part of the proof is identical to (a) of Lemma 2.

(b) *Identity of internal nodes*: In this case, the proof follows the exact same steps as (b) of Lemma 2.

(c) *Updating of* τ: The updating of τ_{P_j} and the weights of all the nodes in the path from P_{P_j} to R_i is achieved, as shown in (c) of Lemma 3. This is proved below.

DOI: 10.1002/dac

In the case of STL-3, the parent of t_k, P_k, is not t_i, and hence, an additional operation is performed, which consists of *adding* τ_j to all the τ's in the path from P_k to L_i. Since t_j is incorporated into the subtree rooted at L_i in $\mathcal{T}'$, we see from (A2) that $\tau_{2u} = \sum_{v=1}^{s} \tau_{2v-1}$. Thus, all the nodes in that path satisfy $\tau'_{2u} = \tau'_{L_{2u}} + \tau'_{R_{2u}}$. This implies that $\mathcal{T}'$ will satisfy all the properties of Definition Structure_PBST. □

The result follows.

Proof of Lemma 5

The proof of this lemma is similar to the proof of Lemma 2 (STL-1 validity). In this lemma, t_j and t_k interchange roles in $\mathcal{T}$, being t_{2u-1} and t_{2u+1}, respectively. First of all, we clarify the notation for the nodes involved in this proof, t_i is an internal node, whose right child, t_k, has a sibling which is the parent of t_j, and additionally, t_k and t_j are adjacent leaves.

In order to ensure the validity of Rule STR-1, we must show that the resulting tree, $\mathcal{T}'$, satisfies the properties of Definition Structure_PBST. We achieve this by proving the following three statements:

(a) Each leaf node of $\mathcal{T}'$ is at position $2u-1$, $1 \leqslant u \leqslant m$.
(b) Each internal node of $\mathcal{T}'$ is at position $2u$, $1 \leqslant u \leqslant m$.
(c) The weight of each internal node, τ'_{2u}, in $\mathcal{T}'$, must be obtained as $\tau_{L_{2u}} + \tau_{R_{2u}}$, where L_{2u} and R_{2u} are the left and right children of t_{2u}, respectively.

Without loss of generality, we let t_i be t_{2u}, for some integer u, where $1 \leqslant u \leqslant m$. As a consequence of this, the next node to be enumerated after t_{2u} in $\mathcal{T}$ will be $t_k = t_{2u+1}$. Besides $P_{P_j} = t_i$, which implies that $L_i = P_j$, and hence t_j (a leaf) will be t_{2u-1} in $\mathcal{T}$. As a result, P_j will be t_{2u-2} in $\mathcal{T}$. Thus, we have $\mathcal{T} = \{t_1, \ldots, t_{2m-1}\}$, where $t_{2u-2} = P_j$, $t_{2u-1} = t_j$, $t_{2u} = t_i$, and $t_{2u+1} = t_k$.

(a) *Identity of leaf nodes*: After performing the STR-1 operation on t_i, P_j will become the right child of t_i. This implies that P_j will remain as an internal node in $\mathcal{T}'$, because t_j and t_k become its left and right children, respectively. Since t_j and t_k are both leaves, they will correspond to t'_{2u-1} and t'_{2u+1} in $\mathcal{T}'$, the resulting tree after performing the STR-1 operation, respectively. In addition, all the other leaves of $\mathcal{T}'$, $\{t'_1, \ldots, t'_{2u-3}, t'_{2u+3}, \ldots, t'_{2m-1}\}$, will remain unchanged. Therefore, (i) of Definition Structure_PBST is satisfied.
(b) *Identity of internal nodes*: From STR-1, we see that P_j becomes the right child of t_i in $\mathcal{T}'$. Consequently, t_i and P_j will be t'_{2u-2} and t'_{2u}, respectively, in $\mathcal{T}'$. Additionally, all the other internal nodes, $\{t'_2, \ldots, t'_{2u-4}, t'_{2u+2}, \ldots, t_{2m-2}\}$, will remain unchanged in $\mathcal{T}'$, and hence (ii) of Definition Structure_PBST will be satisfied.
(c) *Updating of* τ: As a result of the STR-1 operation, $\tau_{P_j} = \tau'_{2u}$ is the only weight that is modified in $\mathcal{T}'$. Since t_k becomes the right child of P_j, τ_k is added to τ_{P_j} (this is achieved in Step (a) of Rule STR-1). In addition, after performing the STR-1 operation, t_j will remain as a child of P_j, and B_j will not be child of P_j anymore. Therefore, since τ_{B_j} is subtracted in Step (a) of Rule STR-1, τ_{P_j} becomes the sum of $\tau_j = \tau_{2u-1}$ and $\tau_k = \tau_{2u+1}$, and hence the updated value of τ'_{P_j} is

$$\tau'_{P_j} \leftarrow \tau_{P_j} + \tau_k - \tau_{B_j}$$

Since the weights of *all* the other nodes in $\mathcal{T}'$, $\{\tau'_1, \ldots, \tau'_{2u-3}, \tau'_{2u+2}, \ldots, \tau'_{2m-2}\}$, remain unchanged, *all* the properties of Definition Structure_PBST will be satisfied. □

The result follows.

DOI: 10.1002/dac

Proof of Lemma 6
This proof is similar to the proof of Lemma 5 and is omitted to avoid repetition. □

Proof of Lemma 7
The proof of this lemma follows the steps of the proof of Lemma 5 and is again omitted to avoid repetition. □

ACKNOWLEDGEMENTS

A preliminary version of this paper was presented at ISCIS 2007, the 2007 International Symposium on Computer and Information Sciences, Ankara, Turkey, November 2007 [40]. We sincerely thank the anonymous referees of this paper. Their comments helped to improve the readability of the paper. The work of L. Rueda was partially supported by CONICYT, the Chilean National Council for Research in Science and Technology, FONDECYT Grant No. 1060904. The work of B. J. Oommen was partially supported by NSERC, the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

1. Deorowicz S. Second step algorithms in the burrows-wheeler compression algorithm. *Software—Practice and Experience* 2002; **32**(2):99–111.
2. Albers S, Mitzenmacher M. Average case analyses of list update algorithms, with applications to data compression. *Algorithmica* 1998; **21**:312–329.
3. Gagie T. Dynamic Shannon coding. *Proceedings of the 12th Annual European Symposium on Algorithms*, Bergen, Norway, 2004; 359–370.
4. Gagie T. Dynamic Shannon coding. *Information Processing Letters* 2007; (2–3):113–117.
5. Oommen BJ, Zgierski J. A learning automaton solution to breaking substitution ciphers. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 1993; **15**:185–192.
6. Albers S. Improved randomized on-line algorithms for the list update problem. *SIAM Journal on Computing* 1998; **27**:670–681.
7. Albers S, Westbrook J. Self-organizing data structures. In *Online Algorithms*: *The State of the Art*, Amos F, Gerhard W (eds). Lecture Notes in Computer Science, vol. 1442. Springer: Berlin, 1998; 13–51.
8. Knuth D. *The Art of Computer Programming*, vol. 3. Addison-Wesley: Reading, MA, 1973.
9. Walker W, Gotlieb C. A top-down algorithm for constructing nearly optimal lexicographical trees. *Graph Theory and Computing*. Academic Press: New York, 1972.
10. Allen B, Munro I. Self-organizing binary search trees. *Journal of Association for Computing Machinery* 1978; **25**:526–535.
11. Iacono J. Alternatives to splay trees with $o(\log n)$ worst-case access times. *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms* (*SODA-01*), Washington, DC, U.S.A., 2001; 516–522.
12. Sleator D, Tarjan R. Self-adjusting binary search trees. *Journal of the Association for Computing Machinery* 1985; **32**:652–686.
13. Williams H, Zobel J, Heinz S. Self-adjusting trees in practice for large text collections. *Software—Practice and Experience* 2001; **31**(10):925–939.
14. Bitner J. Heuristics that dynamically organize data structures. *SIAM Journal on Computing* 1979; **8**:82–110.
15. Bent S, Sleator D, Tarjan R. Biased search trees. *SIAM Journal on Computing* 1985; **14**:545–568.
16. Mehlhorn K. Dynamic binary search. *SIAM Journal on Computing* 1979; **8**:175–198.
17. Aragon C, Seidel R. Randomized search trees. *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science*, Duke, NC, U.S.A., 1989; 540–545.
18. Sherk M. Self-adjusting *k*-ary search trees and self-adjusting balanced search trees. *Technical Report 234/90*, University of Toronto, Toronto, Canada, February 1990.
19. Cheetham R, Oommen BJ, Ng D. Adaptive structuring of binary search trees using conditional rotations. *IEEE Transactions on Knowledge and Data Engineering* 1993; **5**(4):695–704.

DOI: 10.1002/dac

20. Lai T, Wood D. Adaptive heuristics for binary search trees and constant linkage cost. *SIAM Journal on Computing* 1998; **27**(6):1564–1591.
21. Adel'son-Vel'skii G, Landis E. An algorithm for the organization of information. *Soviet Mathematics—Doklady* 1962; **3**:1259–1262.
22. Sayood K. *Introduction to Data Compression* (2nd edn). Morgan Kaufmann: Los Altos, CA, 2000.
23. Storer J, Cohn M (eds). *Proceedings*, *Data Compression Conference*, Los Alamitos, CA. IEEE Computer Society Press: Silver Spring, MD, 2004.
24. Huffman DA. A method for the construction of minimum redundancy codes. *Proceedings of IRE* 1952; **40**(9): 1098–1101.
25. Faller N. An adaptive system for data compression. *Seventh Asilomar Conference on Circuits*, *Systems*, *and Computers*, Pacific Grove, CA, U.S.A., 1973; 593–597.
26. Gallager R. Variations on a theme by Huffman. *IEEE Transactions on Information Theory* 1978; **24**(6):668–674.
27. Knuth D. Dynamic Huffman coding. *Journal of Algorithms* 1985; **6**:163–180.
28. Vitter J. Design and analysis of dynamic Huffman codes. *Journal of the ACM* 1987; **34**(4):825–845.
29. Hankerson D, Harris G, Johnson Jr P. *Introduction to Information Theory and Data Compression*. CRC Press: Boca Raton, FL, 1998.
30. Ahlswede R, Han TS, Kobayashi K. Universal coding of integers and unbounded search trees. *IEEE Transactions on Information Theory* 1997; **43**(2):669–682.
31. Ziv J, Lempel A. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* 1977; **23**(3):337–343.
32. Ziv J, Lempel A. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory* 1978; **25**(5):530–536.
33. Cleary J, Witten I. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications* 1984; **32**(4):396–402.
34. Kieffer JC, Yang E. Grammar-based codes: a new class of universal lossless source codes. *IEEE Transactions on Information Theory* 2000; **46**(3):737–754.
35. Jones D. Application of splay trees to data compression. *Communications of the ACM* 1988; **31**(8):996–1007.
36. Rueda L, Oommen BJ. A fast and efficient nearly-optimal adaptive Fano coding scheme. *Information Sciences* 2006; **176**:1656–1683.
37. Fenwick P. A new data structure for cumulative frequency tables. *Software—Practice and Experience* 1994; **24**(3):327–336.
38. Moffat A. An improved data structure for cumulative probability tables. *Software—Practice and Experience* 1999; **29**(7):647–659.
39. Rueda L, Oommen BJ. A nearly optimal Fano-based coding algorithm. *Information Processing and Management* 2004; **40**(2):257–268.
40. Rueda L, Oommen BJ. A new approach to adaptive encoding data using self-organizing data structures. *Proceedings of the 22nd International Symposium on Computer and Information Sciences*, Ankara, Turkey, 2007; 15–20.

AUTHORS' BIOGRAPHIES

Luis Rueda received the degree of 'Licenciado' in Informatics at the National University of San Juan, Argentina, in 1993, and his Master's and PhD in Computer Science at Carleton University, Canada, in 1998 and 2002, respectively. He was with the School of Computer Science, University of Windsor, Canada, from 2002 to 2005. He is currently an associate professor in the Department of Computer Science, University of Concepción, Chile. His research interests include bioinformatics, DNA microarray data analysis, and pattern recognition. He has published more than 50 papers in well-known journals and conferences, has organized and been a member of the program committee for various conferences in his research areas, and has also refereed for prestigious journals in computer science. Luis Rueda holds a patent on encryption with perfect secrecy and stealth, and is a member of the IEEE, the Pattern Recognition Society and the IAPR.

DOI: 10.1002/dac

B. John Oommen was born in Coonoor, India on 9 September 1953. He obtained his BTech degree from the Indian Institute of Technology, Madras, India in 1975. He obtained his ME from the Indian Institute of Science in Bangalore, India in 1977. He then went on for his MS and PhD which he obtained from Purdue University, in West Lafayettte, Indiana in 1979 and 1982, respectively. He joined the School of Computer Science at Carleton University in Ottawa, Canada, in the 1981–1982 academic year. He is still at Carleton and holds the rank of a Full Professor. In July 2006, he has awarded the honorary rank of Chancellor's Professor, which is a lifetime award from Carleton University. His research interests include automata learning, adaptive data structures, statistical and syntactic pattern recognition, stochastic algorithms, and partitioning algorithms. He is the author of more than 285 refereed journal and conference publications, and is a Fellow of the IEEE and a Fellow of the IAPR. Dr Oommen is on the Editorial Board of the *IEEE Transactions on Systems*, *Man and Cybernetics* and *Pattern Recognition*.

DOI: 10.1002/dac