International Journal of Foundations of Computer Science Vol. 1 No. 4 (1990), 425-447
© World Scientific Publishing Company

TREE COMPRESSION AND OPTIMIZATION WITH APPLICATIONS

Dedicated to the memory of Markku Tamminen (1945-1989)

JYRKI KATAJAINEN
Department of Computer Science, University of Turku,
Lemminkäisenk. 14, SF-20520 Turku, Finland

and

ERKKI MÄKINEN
Department of Computer Science, University of Tampere
P.O. Box 607, SF-33101 Tampere, Finland

Received 20 October 1989
Revised 1 March 1990

Different methods for compressing trees are surveyed and developed. Tree compression can be seen as a trade-off problem between time and space in which we can choose different strategies depending on whether we prefer better compression results or more efficient operations in the compressed structure. Of special interest is the case where space can be saved while preserving the functionality of the operations; this is called data optimization. The general compression scheme employed here consists of separate linearization of the tree structure and the data stored in the tree. Also some applications of the tree compression methods are explored. These include the syntax-directed compression of program files, the compression of pixel trees, trie compaction and dictionaries maintained as implicit data structures.

Keywords: data optimization, linearization of tree structure, linearization of data, syntax-directed compression, pixel tree, trie, implicit dictionary.

1. Introduction

Data compression is a standard operation for which there are well-known utilities, e.g. *compact* and *compress* in UNIX[a]. Traditionally, input consists of a stream of tokens which can be bits, characters or words of fixed or varying length. Concatenation of consecutive tokens is the structure in the input. However, the data to be compressed is often stored within a structure of some special form, e.g. list, tree,

[a] UNIX is trademark of AT&T Bell Laboratories.

graph, etc. The plain data is usually worthless or at least of little value outside its context. Thus, besides the data itself, we must also store and compress if possible the structure in which the data is stored. In this paper we suppose that the data structure to be compressed and then manipulated is a tree. For a general survey of data compression the reader is referred to the recent work by Lelewer and Hirschberg [1].

Trees are widely used structures for maintaining data. They are also used as auxiliary data structures when compressing data (see e.g. [2,3]). Often it is also necessary to reduce the space needed for storing a tree itself. In this paper different tree compression methods are surveyed and developed. Trees are regarded both as the object and medium of data compression. We mainly concentrate on binary trees but also make some remarks on k-ary trees; these (ordered, rooted) trees cover most cases of practical importance.

Let us now formulate the *tree compression problem* abstractly. Given a tree, the task is to map it as compactly as possible to memory, which is seen as a string of bits, a set of memory locations (words), or a set of memory blocks (pages). The range of the mapping depends on the application in question. In traditional tree compression the only operations performed are encoding of a tree to a bit string and decoding the bit string back to a tree.

The benifits gained through data compression of large-to-very-large trees are obvious since compression reduces storage and data transfer requirements. On the other hand, there are some severe disadvantages of tree compression. Above all, compression makes all the normal tree operations (children, parent, search, delete, insert, etc.) more expensive. In most compression methods there is no other way to perform these operations other than to decode the compressed tree, carry out the operation, and encode the tree again!

In the *tree optimization problem*, a term adopted from the work of Jacobson [4], the task is to maintain the functionality of a tree in the compressed form. That is, we want to perform some tree operations as efficiently as done in the uncompressed case (where an operation is a simple matter of pointer manipulation). We are mainly concerned with applications where the trees are manipulated in the internal memory of a computer. So, the mappings are to bit strings or memory words only. For applications concerning external memories, see for example [5,6].

The compression and optimization of trees is usually performed in two phases: the compression of the structure (*linearization of the tree structure*) and the compression of the data stored in it (*linearization of data*). Our general policy is to handle the data and the structure separately. This enables us to compress the plain data by using any of the known methods and independently find an efficient coding method for the tree structure irrespective of the form and contents of the data items stored in the nodes. We shall not consider normal data compression methods but assume that the reader is familiar with e.g. arithmetic coding [7] and Huffman coding [2]; for a recent textbook on data compression, see [8]. We want to emphasize that the separation of the data from the structure will not always give an optimal compression result (see the applications in sections 6.1 and 8.2) and may not even be possible in some cases.

Appropriate linearization methods for binary trees are presented in chapters 2 and 3. We shall see that only about 2n bits are needed for the structure of any tree on n nodes; this result is asymptotically optimal. In chapter 4 we attempt to find the information theoretic optimum. The idea is to represent the structure of a tree by a single natural number (called the *rank* of the tree) from the interval 1,…, B_n, where B_n stands for to nth Catalan number giving the number of different binary trees on n nodes. Chapter 5 deals with the encoding of k-ary trees.

Typical application areas of the tree compression methods include the representation of graphics as pixel trees [9-12] and program files as syntax trees [13,14]. Both applications are of practical importance, and the tree methods support excellent compression results. These applications are examined in greater detail in chapter 6.

Chapter 7 contains a sketch of the basic ideas presented in [4] showing that tree traversal is possible in asymptotically optimal space.

One can separate the concepts of *concrete* and *abstract optimization*. In concrete optimization the data structure to be compressed is given while in abstract optimization only the desired operations are defined. As data optimization applications we study trie compaction [15-17] and the design of implicit data structures [18,19] in chapter 8. Trie compaction is an example of concrete optimization where the purpose of the data optimization is to implement the search operation within the same time bound as for the uncompressed trie. The trie compaction problem contains several NP-complete sub-problems. We consider heuristics for solving one of them. On the other hand, the construction of an implicit dictionary is an example of abstract optimization. Given only a constant amount of extra space, the goal is to perform the operations search, insert, and deletion as efficiently as possible.

The paper closes with some concluding remarks in chapter 9.

2. Encoding with Fixed Length Codewords

Several encoding methods for binary trees are presented in the literature; see [20] for a treatment of general tree types and [21] for a survey on binary trees. When forming a one-to-one correspondence between binary trees and integers, these methods use different kinds of number sequences as intermediate representations. The present chapter and the next are devoted to encoding methods whose intermediate phases have reasonable space requirements. We also suggest that these intermediate phases can be used as compressed representations for tree structures. In fact, there are several methods for forming a one-to-one correspondence between trees and integers which are not presented in this paper because they employ space intensive intermediate representations, e.g. permutations (see [21]).

We start by introducing an encoding method presented by Zaks [22]. Consider the tree in figure 1a. Label all the nodes by 1 and all the missing subtrees by 0 as in figure 1b. We obtain the codeword, called *Zaks' sequence*, by reading the labels in preorder. (Visit first the root, then recursively traverse the left subtree in preorder, and then the right subtree in preorder.) Hence, Zaks' sequence related to the tree in figure 1a is 111100100100111001000.

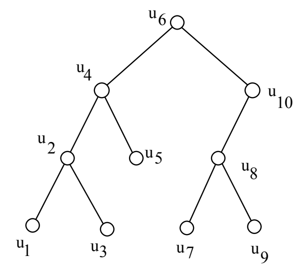
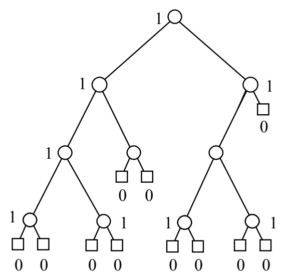


Fig. 1a. A binary tree.



Fib. 1b. The numbering of nodes and leaves related to Zaks' sequences.

We have the following characterization for feasible Zaks' sequences. A bit string is a Zaks' sequence if and only if the following three conditions hold

i) the string begins with 1,

ii) the number of 0's is one greater than the number of 1's,

iii) no proper prefix of the string has the property 2).

The length of a Zaks' sequence is 2n + 1 for a tree with n nodes [22].

The *children pattern sequence* is a codeword closely related to Zaks' sequence. In the children pattern method we label the nodes of a binary tree by 00, 01, 10 or 11 depending on whether the node has no children, only the right child, only the left child or two children, respectively. The codeword is obtained by reading the labels in preorder as in the Zaks' method. The children pattern sequence of the tree in figure 1a is 11111100000010110000. Generally, the codeword obtained for a binary tree on n nodes has length 2n.

Yet another method for representing a binary tree with 2n bits is to use *balanced parentheses*. A pair of parentheses corresponds to the root and the children are recursively represented in the same way inside the parentheses.

To decode a tree structure from a given Zaks', children pattern, or balanced paranthesis sequence is a relatively straightforward task and for this reason we exclude it from our treatment.

In the above methods we traverse the trees in preorder. Lee et al. [23] have used level-by-level order, i.e. first the root, then the children of the root from left to right, then their children from left to right, and so on. The length of these *level-to-level sequences* naturally equals the length of the codewords obtained by the the methods described above. Lee et al. [23] have found the codewords so obtained useful for some special purposes. Moreover, in chapter 7 we shall see that the level-by-level sequence allows tree traversal in compressed trees.

We end this chapter by considering three types of binary trees which form a hierarchy of the number of bits needed to represent their structure by using encodings like Zaks' sequence.

A binary tree is said to be *regular* if each node has either two children or no children at all. Consider now the children pattern sequence of a regular binary tree. We do not need labels 01 and 10. Thus, we may label a node having two children with label 1 and a node having no children with label 0. It follows that the children pattern sequence of a regular binary tree has only n bits. It is in fact easy to prove that there are exactly as many regular binary trees on 2n + 1 nodes as there are arbitrary binary trees on n nodes. Hence, the space requirement of n bits is asymptotically optimal (cf. chapter 4).

The (almost) complete tree structure is an example of an even more drastic instance of the above phenomenon. A binary tree is said to be *complete* if all the internal nodes have two children and all the paths from leaves to the root are of equal length. In such a tree the number of nodes is of the form $2^n - 1$. A binary tree is *almost complete* if it can be made complete by inserting leaves to the right-hand side end of the bottom level. The only thing needed to describe the shape of the tree is the number of nodes! Hence, $\lceil \log n \rceil$ bits are needed when representing the shape of an almost complete tree on n nodes. This property is extensively used in implicit data structures (cf. section 8.2).

One criteria for comparing different methods is how easy it is to detect from the encoded string whether or not there are any regularities in the tree. In Zaks' sequence the code information related to a node is in different parts of the string, while in the children pattern method the two bits describing a node are always together. Suppose we are compressing a regular tree without knowing its degree of regularity. By using the children pattern method together with arithmetic coding we obtain a compression result much better than 2 bits per node provided that the compression model reads the string as a sequence 2-bit integers. Even better compression gain is obtained when the children patterns are output in the level-by-level order and the tree to be compressed is almost complete. The resulting string has a long sequence of 1's followed by another sequence of 0's.

3. Encoding with Varying Length Codewords

In this chapter we introduce two encoding methods based on rotations. (For the various ways of using rotations in maintaining data structures, see e.g. [24].) These methods assign an integer to each node of the tree in question and the codeword is obtained by traversing the tree in symmetric order (traverse first the left subtree in symmetric order, then visit the root, and then traverse the right subtree in symmetric order). The number of bits needed in the resulting codeword varies because the size of the integers assigned to the nodes depends on the shape of the tree. Sometimes less than 2n bits are needed for representing a given tree's shape.

A rotation is an operation which transforms one tree into another as follows. If node u is the left child of v then a *right rotation* makes v the new right child of u, the old right child of u (if any) becomes the new left child of v, and the old parent of v (if any) becomes the new parent of u. A *left rotation* transforms the tree obtained back to the original one (see figure 2).

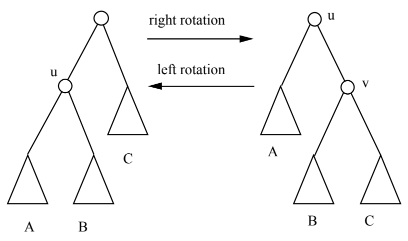


Fig. 2. Right and left rotations.

The rotation based coding methods fix some tree shape and then count the rotations needed for transforming a given tree into the fixed tree by using some standard order of rotations. A natural choice for the fixed tree shape is the degenerated tree consisting of left children only. Such a tree is called a *left list*.

In Zerling's [25] method we make left rotations with respect to the edge joining the current root and its right child until the greatest node in symmetric order is on the root. The code item x_{n-1} related to the greatest node u_{n-1} is the number of rotations done. Naturally, it is possible that $u_{n-1} = 0$. The same procedure is then repeated in the left subtree of the tree so far obtained; i.e. the root node is discounted and left rotations are done with respect to the edge joining the new root and its right child. This continues until the code item x_1 is set to have value 0 or 1, depending on whether we must rotate when there are two nodes left. The codeword representing a tree on n nodes has n - 1 code items. Zerling's method gives the codeword (1,0,1,0,1,1,0,2,1) for the tree in figure 1a.

In order to uniquely decode the tree structure we naturally have to know how many bits are needed for representing each code item. The sum of the code items of a tree on n nodes always lies between 0 and n-1 [25]. In a codeword $(x_1,x_2,\ldots,x_{n-1})$ define

$$T_i = \sum_{j=1}^{i-1} x_j, \text{for } i = 1,\ldots, n\text{-}1.$$

Now, the maximum value M_p for the code item x_p, p > 1, is $M_p = p - T_{p-1}$. The code item x_1 is 0 or 1. Hence, by setting $T_0 = 0$ we have defined M_p for all p = 1,..., n - 1. By using the values M_p we can calculate the number of bits needed for representing each code item.

As an example, consider binary trees on 10 nodes. In this case all codewords contain 9 code items. If the codeword to be stored is (1,1,1,1,1,1,1,1,1) we have $M_p = 1$, p = 1,..., 9. Thus, it suffices to reserve one bit per node. An opposite case appears if the codeword is (0,0,0,0,0,0,0,0). We have $M_p = p$, $p = 1,..., 9$, and we have to reserve one bit for x_1, two bits for x_2 and x_3, three bits for $x_4,..., x_7$, and four bits for x_8 and x_9. Totally, we need 25 bits. In general, we may need as many as

$$\sum_{i=1}^{n-1} \lceil \log i \rceil$$

bits. Hence, it is advantageous to have x_i as close M_i as possible.

A unique code system is also obtained when rotations are done on the edges joining the next node to be "lifted up" (the greatest node in symmetric order in the current tree) and its parent. The code item x_{n-1} for the greatest node u_{n-1} has the same value as in Zerling's method. When making rotations on the root of the tree Zerling's method transforms some smaller node to the left arm (i.e. the path from the root following the left child pointers), while the latter method does not change the distance of the smaller nodes from the left arm. As a consequence, this method has the property that for each node u_i, the distance from the left arm equals the code item x_i, when the distance $d(u_i)$ from the left arm is defined by

$$d(u_i) = \begin{cases} 0, \text{ if } u_i \text{ is on the left arm,} \\ d(p(u_i)), \text{ if } u_i \text{ is a left child,} \\ d(p(u_i)) + 1, \text{ if } u_i \text{ is a right child,} \end{cases}$$

where $p(u_i)$ stands for the parent of u_i [26].

This method gives the codeword (0,1,0,1,0,1,1,2,1) for the tree of figure 1a.

A feasible codeword $(x_1,x_2,...,x_{n-1})$ always has $x_1 = 0$ or $x_1 = 1$ and x_i is in $[0, x_{i-1} + 1]$, for $i = 2,..., n - 1$ [26]. Hence, the bits needed for the code item x_i depend on the value of x_{i-1}. The minimum number of bits is for the codeword (0,0,...,0) and maximum number for the codeword $(1, 2,\ldots, n - 2, n - 1)$. Note that this gives the same upper bound for the number of bits needed as Zerling's method.

Although rotation based encoding methods generally consume more space than Zaks' sequences there are also special cases in which they out perform Zaks' sequence as we have seen in the above examples.

4. Seeking after the Information Theoretic Optimum

In the two previous chapters we have presented different ways for representing tree structures as number sequences. In this chapter we discuss the possibility of representing tree structures as single integers.

Each of the methods presented in chapters 2 and 3 has a corresponding *ranking algorithm* which returns a unique integer from the interval $1,\ldots, B_n$ related to the tree in question. The number of binary trees on n nodes is

$$B = \frac{1}{n+1}\binom{2n}{n} [27, \text{section } 2.3.4.4.].$$

We now need $\lceil \log_2 B_n \rceil$ bits for representing any binary tree on n nodes. Since B_n is less than 2^{2n}, we save some space compared to the bit sequence methods. Clearly, it is impossible to differ B_n trees by using less bits than it is needed for the first B_n integers. Hence, this method reaches the information theoretic optimum with respect to the number of bits needed.

The ranking algorithms of the different encoding methods are basically similar and we present here only one example. We start with the Zaks' sequence 11110010010011100100 of the tree in figure 1a. The *z-sequence* of a Zaks' sequence contains the bit positions set to 1. In our example, we have z = 1, 2, 3, 4, 7, 10, 13, 14, 15, 18. The z-sequence is repeatedly reduced by omitting the largest i such that $z_i = i$ and subtracting 2 from all the z-values which lie to the right of the omitted one until the z-value to be omitted is the largest in the sequence. Hence, in our example the z-sequence is first reduced to the form z = 1, 2, 3, 5, 8, 11, 12, 13, 16, since 4 is the omitted value. If the z-sequence has length n and the omitted item is in the p^{th} index, we add $a_{n,p}$ to the sum index(z). In our example, the first term in the sum is $a_{10,4}$. The $a_{n,p}$-values are obtained as follows ($0 \le p \le n-1$)

$a_{n,n-1} = 1$, for all n,

$a_{n,0} = (2n)!/n!(n+1)!$, for all n,

$a_{n,p} = a_{n,p+1} + a_{n-1,p-1}$, otherwise.

When index(z) is calculated, the rank r(T) of the tree is obtained as

$$r(T) = B_n - \text{index}(z) + 1.$$

We have index(11110010010011100100) = $a_{10,4} + a_{9,3} + a_{8,3} + a7{,}2 + a_{6,2} + a_{5,1} + a_{4,3} + a_{3,2} + 1$ = 1638 + 1001 + 275 + 165 + 48 + 28 + 1 + 1 + 1 = 3158. The rank of the tree in figure 1a is r(T) = Bn - index(z) + 1 = 16796 - 3158 + 1 = 13637. Hence, instead of the 20 bits of the Zaks' sequence, we now need only $\lceil \log_2 13637 \rceil$ = 14 bits.

In order to decode the integer back to the tree, we naturally need an unranking algorithm which returns the Zaks' sequence related to the integer in question [22].

5. Encoding of Multiway Trees

In the preceding chapters we have considered the encoding of binary trees only. This chapter shows how these binary tree encodings can be used when encoding k-ary trees. We use the phrase multiway tree for ordered k-ary trees having an arbitrary degree k.

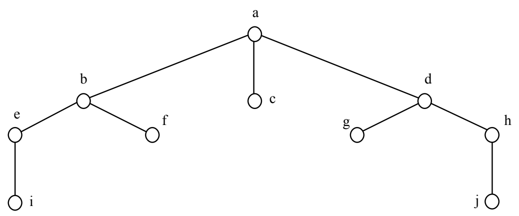


Fig. 3a. A multiway tree.

From a given multiway tree we can construct the corresponding child-sibling binary tree representation as follows [28]. Let x be a node in the multiway tree with t children (from left to right) x_1, x_2,..., x_t. In the corresponding binary tree the node containing x has only two pointers. The left pointer accesses the first child x_1, and the right pointer accesses x's first sibling, if any. Similarly, the left pointer of x_1 accesses its first child, and the right pointer of x_1 accesses x_2. The child-sibling binary tree representation of the multiway tree of figure 3a is shown in figure 3b. It is clear that this transformation preserves the uniqueness of tree shapes. Furthermore, the binary representation of a given multiway tree and its decoding back into a multiway tree can be produced in linear time. It follows that the shape of a multiway tree can be represented as efficiently as the structure of a binary tree on the same number of nodes.

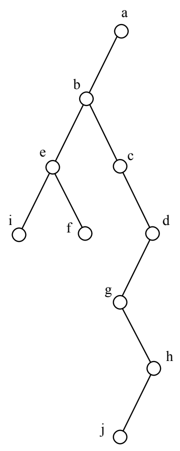


Fig. 3b. The child-sibling binary tree representation of the multiway tree in figure 3a.

Another way of achieving this is the arity method: mark each node by its arity and read the labels in some fixed order. If these arities are encoded by fixed length codewords the total number of bits needed for a tree on n nodes is $n\lceil \log_2 d \rceil$, where d is the greatest arity in the tree. A more efficient way of doing this, called the unary degree method, is described in [4,20]. In this method we encode each arity d by the string 1^d0 (d 1-bits followed by a 0-bit). Hence, instead of fixed length codewords we use varying length ones. This reduces the total number of bits down to 2n - 1 because for each node one 1-bit and one 0-bit is needed, except for the root which cannot be a child of any other node.

A multiway tree having degree k is *regular* if each node has either k children or no children. We can easily prove that the number of bits needed for representing the structure of a regular multiway tree on n nodes is 2n/k, which is also asymptotically optimal. As in the case of regular binary trees, we can gain further savings by making use of the regularities in the k-ary tree encoding. This is possible if we use a compression method which adapts to the patterns appearing in the tree structure.

6. Data Compression Applications

This chapter introduces two common tree compression applications, syntax-directed compression of program files and the compression of pixel trees.

6.1 Syntax-Directed Compression of Program Files

In syntax-directed compression of program files trees are used as an intermediate step in the compression process: a linear structure is first transform to a tree and then back to a linear form again. The basic idea of syntax-directed compression is first to construct a *parse tree* for a source program, to label the nodes of the tree with *small integers*, and then to encode a linearized form of the tree as *compactly* as possible. Another way to say this is that the compressed form represents a *Szilard language* of the grammar generating the source text (for recent advances of the theory of Szilard languages, see [29]). This idea is an old one introduced as early as the beginning of seventies by several researchers (see [30,31]). Recently, many compression system implementations have been published (see for example [13,14,32-34]) and two Ph.D. theses, [35,36], have been written on the subject.

The best results obtained by using syntax-directed compression are reported by Cameron [13]. He has achieved an overall compaction of better than 85% on average. (Observe that his encoder uses some context-sensitive modelling techniques as well.) This clearly out performs the results of the UNIX utility *compact*, implementing adaptive Huffman encoding, which achieves a 55% saving on average.

The decomposition of a syntax-directed compression system is similar to that of a compiler (see [37]). First, the source text is scanned from left to right and grouped into *tokens*. Tokens are sequences of characters which have a collective meaning, that is, *syntactic terminals* (keywords, operators, punctuation symbols) and *user terminals* (constants, identifiers). Second, the token list is parsed and simultaneously a parse tree is constructed. More precisely, the order is determined in which the productions have to be applied in order to regenerate the token list with the grammar in question. Third, the *symbol table* (gathered during the scanning phase) and the parse tree are bit-encoded.

Most systems assume that the input is always syntactically correct. This assumption is not restrictive because syntactically incorrect programs are rarely stored or transmitted. Moreover, if the program is produced by a syntax-directed editor, it cannot be incorrect. In addition, there should exist character-level subroutines for compressing comments in the input program.

To illustrate the compression process, let us consider a small example. We assume that the user terminals in our hypothetic programming language are IDENTIFIERs and INTEGERs. An identifier must begin with a letter which is followed by some (if any) letters or digits. An integer is a sequence of digits. Further, we assume that the programs to be compressed are generated by the following grammar. When describing the grammar, we use the following conventions. A *non-terminal* symbol is written with lower-case letters, a *syntactic terminal* of the language is delimited by apostrophe marks ('), and a *user terminal* is written with upper-case letters. The *start symbol* of the grammar is the non-terminal program.

P_1:	program	$\rightarrow$	heading ';' statements '.'
P_2:	heading	$\rightarrow$	'PROC' IDENTIFIER '(' identifiers ')'
P_3:	identifiers	$\rightarrow$	IDENTIFIER ';' identifiers
P_4:	identifiers	$\rightarrow$	IDENTIFIER
P_5:	statements	$\rightarrow$	statement ';' statements
P_6:	statements	$\rightarrow$	statement
P_7:	statement	$\rightarrow$	IDENTIFIER ':=' expression
P_8:	statement	$\rightarrow$	'IF' expression 'THEN' statement 'ELSE' statement
P_9:	statement	$\rightarrow$	'WHILE' expression 'DO' statement
P_{10}:	statement	$\rightarrow$	'RETURN' IDENTIFIER
P_{11}:	statement	$\rightarrow$	'(' statements ')'
P_{12}:	expression	$\rightarrow$	factor operand factor
P_{13}:	expression	$\rightarrow$	factor
P_{14}:	factor	$\rightarrow$	IDENTIFIER
P_{15}:	factor	$\rightarrow$	INTEGER
P_{16}:	operand	$\rightarrow$	'='
P_{17}:	operand	$\rightarrow$	'>'
P_{18}:	operand	$\rightarrow$	'+'
P_{19}:	operand	$\rightarrow$	'-'

In figure 4(a-c) we show how a program (computing the sum of the first n natural numbers) is encoded. The compressed form of the source program is a concatenation of the symbol table and the parse tree. The tree is linearized by outputting the nodes in preorder.

```
PROC sum(n);
result:= n;
WHILE n>1 DO
            (n:= n-1; result:= result+n);
RETURN result.
```

Figure 4a. A program to be compressed.

Symbol table: i_1: 1, i_2: n, i_3: result, i_4: sum

Parse tree
$p_1\ p_2\ i_4\ p_4\ i_2\ p_5\ p_7\ i_3\ p_{13}\ p_{14}\ i_2\ p_5\ p_9\ p_{12}\ p_{14}\ i_2\ p_{17}\ p_{15}\ i_1\ p_{11}\ p_5\ p_7\ i_2$
$p_{12}\ p_{14}\ i_2\ p_{19}\ p_{15}\ i_1\ p_6\ p_7\ i_3\ p_{12}\ p_{14}\ i_3\ p_{18}\ p_{14}\ i_2\ p_5\ p_6\ p_{10}\ i_3$

Figure 4b. The compressed form of the program in figure 4a.

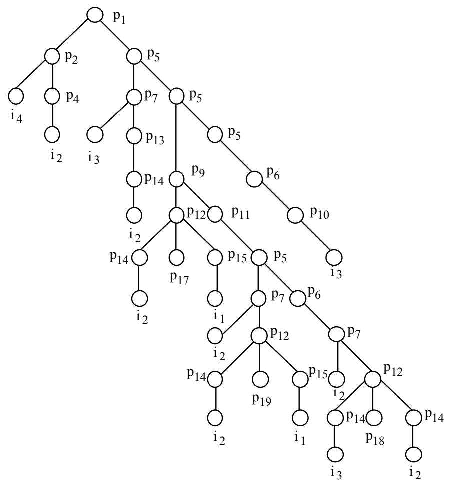


Figure 4c. The parse tree corresponding to the program in figure 4a.

The symbol table can be further compressed by using any traditional text-compression method. To bit-encode the parse tree one can use fixed-length codes. Each user terminal can be encoded by $\lceil \log_2 t \rceil$ bits, where t is the number of different user terminals. One of the simplest methods encodes the production label of a rule, whose left-hand side non-terminal occurs on the left-hand side of r rules by $\lceil \log_2 r \rceil$ bits that will uniquely specify the production which has been used in the substitution of the non-terminal. Note that if a non-terminal has only one derivation no code is needed. For example, in our example grammar the non-terminal statement has 5 derivations, i.e. three bits are then enough to encode the production rules. For more sophisticated variable-length coding schemes, the reader is referred to [13,32,34].

The decoding of the parse tree is basically a simple task achieved by the replacement of production labels. Moreover, the structure of the parse tree need not be stored separately since the arity of each production label is directly recovered from the grammar. The arity method is used here implicitly, not explicitly.

6.2 Binary Pixel Trees

Suppose a *picture* (or an *image*) consists of an array of $2^n \times 2^n$ pixels. Each pixel has a code (i.e. a *colour*) related to it. The problem is to store the picture efficiently. A good alternative to simple sequential storage is to use *pixel trees* which try to divide the picture into uniform areas, where adjacent pixels have the same colour, and to hierarchically organize these areas [9,10,12,38]. Pixel trees may be 4-ary or even 8-ary, but we restrict ourselves to binary pixel trees only.

The root of a binary pixel tree represents the whole picture. If all the pixels have the same colour we can assign the colour code to the root and the representation of the picture is complete. Otherwise, we divide the picture into two sub-pictures of equal size. The sub-pictures are represented by the two children of the root. Again, if in either of the sub-pictures all the pixels have the same colour we can assign the corresponding code to the node. This process is recursively repeated until all the sub-pictures consist of pixels of same colour. This recursive definition has the characteristic feature that a node in the tree always has two children or no children at all. Hence, all binary pixel trees are regular and their structure can be represented by using only one bit per node.

Figure 5a shows a simple picture and figure 5b the corresponding binary pixel tree. In this example we suppose that the picture is black-and-white, i.e. there are only two possible colours: black (code 0) and white (code 1).

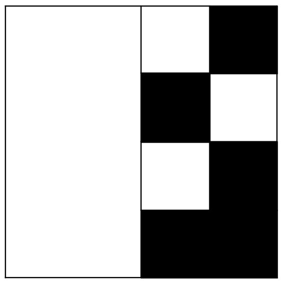
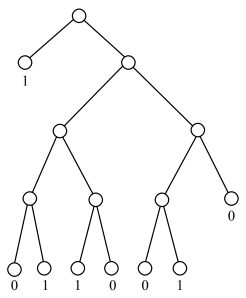


Fig. 5a. A black-and-white picture where the subpictures are divided by turns along the x- and y-axes.



Fib. 5b. The binary pixel tree corresponding the picture in figure 5a.

According to our general scheme the binary pixel tree in figure 5b is represented by two bit sequences. The first sequence represents the structure of the tree. Using 1 for the nodes with two children and 0 for the nodes with no children we obtain the sequence 101110010011000. The data of the tree is yielded by collecting the leaves in preorder producing the sequence 11001010. (A careful reader might have noticed that the resulting bit string is longer than the one obtained by simply reading the codes row by row. This shows that the compression of a picture via a pixel tree is not necessarily optimal. Indeed, the point of our treatment is to compress the tree which describes the picture rather than the picture itself.)

There are some application-dependent possibilities to further compress the data sequence. The size of the pixel array gives the maximal height to the pixel tree. Hence, by considering the depth of a node we can decide whether a given leaf is on the lowest possible level and thus representing a single pixel. On the other hand, we know that in a binary pixel tree a leaf does not have the same colour as its sibling [12]. We can represent the leaves appearing on the lowest possible level as a sequence of their own. This allows us to encode the possible pairs 01 and 10 of adjacent codes by a single bit each [12]. The tree in figure 5b has three pairs of leaves on the lowest level. We can encode these bit pairs in three bits producing a total number of five bits for the data sequence.

Kawaguchi and Endo [10] define a context-free grammar which generates feasible codewords of a binary pixel tree. The codewords represent both the structure of the tree and the data. In this respect their encoding method is less flexible than those following our general scheme.

Gargantini [39] has started the study of "linear quadtrees". In linear quadtrees pointers are eliminated by storing pixels (black only) by using an encoding which reflects the successive quadrant subdivisions. The encoding makes it possible to perform some basic quadtree operations, including the one of finding adjacent nodes, in logarithmic time. Hence, contrary to other applications discussed in this section, linear quadtrees are an example of data optimization. Linear quadtrees and other similar methods are surveyed in [40].

The use of arithmetic coding in pixel tree compression is studied by Langdon and Rissanen [11]. For the compression of 4-ary pixel trees, see e.g. [40-42] and the references therein.

7. Encodings Allowing Traversal Operations

In our treatment so far we have tried to reduce the number of bits needed in representing trees without paying attention to the accessibility of the resulting structure. It is clear that most of the methods discussed in previous chapters are unsuitable for applications where compressed trees are frequently accessed. An ideal compression method would simultaneous achieve minimization of bit usage while allowing efficient traversal in the compressed tree. By traversal we mean the ability to move from a node to its parent and children. Despite their practical importance, compressed trees allowing efficient traversal are mostly overlooked in the literature, Guy Jacobson's thesis [4] being an outstanding exception. In this chapter we survey some of his ideas.

A straightforward method for saving space and allowing efficient traversal in binary trees is to "thread" the tree so that isomorphic subtrees are stored only once. Note that this method is applicable only when we are interested in the structure of the tree and not in the data possibly stored in the nodes. There exists a linear time and space algorithm for finding all isomorphic subtrees of a given tree [43]. If trees are chosen at random under a uniform distribution the compression factor gained when merging subtrees grows slowly without bound. The growth rate is *O*(log n) [4].

Jacobson [4] also gives different kinds of bit string representations allowing efficient access. The basic form of these recursive representations is shown in figure 6. The header block contains information about the sizes of the left and right subtrees. When knowing the sizes of the subtrees we can continue the traversal at an appropriate position in the string. To make this step possible we insist that all n-node trees always occupies exactly the same number of bits in the string. The encodings of the subtrees recursively have their own header blocks. The empty tree is represented by the empty string.

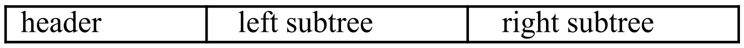


Fig. 6. A recursive string representation of a binary tree.

The basic form shown in figure 6 can be improved by storing the smaller subtree first and having only its size in the header block. In this version we need an additional bit to indicate whether or not the left subtree is the smaller one. Natural numbers standing for the sizes of the subtrees must be encoded by prefix codes (called universal codes in [8]). The sum of the sizes can be represented in amortized constant space for each node of the tree to be compressed. More effort and more bits per node are needed if we want to traverse a tree upwards (from a child to its parent); for details see [4].

Traversal is also possible in a method based on the level-by-level representation mentioned in chapter 2. Recall that the level-to-level sequence is obtained by first labeling the nodes and the missing subtrees by 1's and 0's, respectively, and then reading the labels in the level-by-level order, i.e. first the root, then the children of the root from left to right, and so on. In this sequence the left and right children of the node corresponding to the m^{th} 1-bit are in the positions indicated by the $2m^{th}$ and $2m+1^{st}$ 1-bits. In order to allow fast access to the position of the i^{th} 1-bit, and hence to make the tree traversal more efficient, Jacobson [4] uses directories inspired by the work of Elias [44]. We omit the details concerning the use of the directories, however the results show that it is possible to construct a structure which uses $2n+o(n)$ space and in which a tree traversal requires $O(\log n)$ bit-accesses, or $O(1)$ time if $O(\log n)$ consecutive bits can be manipulated at unit cost.

8. Data Optimization Applications

This chapter discusses two tree compression applications where data optimization is possible. In tries we can perform efficient searches in the compressed structure and in implicit dictionaries we perform operations insert, delete, and search in optimum space.

8.1 Trie Compression

A *trie* is a data structure for representing sets of character strings. The characters are also called *attributes*. Suppose the strings are over an alphabet of k elements. Then a trie is a k-ary tree where each path from the root to a node on level i corresponds to the set of keys in the represented set that start with the same sequence of characters. Each trie node must contain an array with k elements in order to identify the (possibly) k off-spring of the node. To search for a string $A = a_1a_2...a_n$ in a trie we follow the following procedure: In the root examine the array element corresponding to the character a_1. It can be (1) null, (2) a pointer to an auxiliary table containing the strings currently represented by the trie, or (3) a pointer to another node in the trie. Case (1) implies that the string searched is not present in the structure. Case (2) leads to a comparison of A against a string in the trie. A mismatch implies that A is not present. Finally, case (3) means that there is more than one string in the trie starting with a_1. In this case we must continue our search from the node pointed to and repeat the above procedure with the character a_2. This procedure terminates at the latest when the last character a_n in the string is handled. For further details concerning tries, see any standard text on data structures and algorithms, e.g. [24,45].

The main disadvantage of tries is their high space requirement. A straightforward method for saving space is to replace the arrays in the nodes by linked lists which contain an element for each child of the node in question. As a consequence, time spent in selecting the appropriate child of a node is no longer constant: in the worst case we have to examine the entire list whose length is bounded by k. The search time can be reduced by storing the nodes of each list in a binary tree.

Another solution for the trade-off between time and space in tries is given by Ramesh et al. [46]. In their solution the trie does not directly specify individual keys but a set of keys in which an additional binary search is performed. Hence, the purpose of the trie is to reduce the set of possible keys making binary searches more efficient. The size of the sets bounds the time needed for binary searching. One of the problems studied by Ramesh et al. [46] is to minimize the number of sets needed when the maximal size of the sets is given. Another problem is to minimize the worst-case binary search time under the restriction that the number of sets does not exceed a given bound. Both of these problems can be solved in polynomial time, and hence their work gives an alternative for implementing tries.

Compact representations for tries in the connection with predictive text compression are discussed by Teuhola and Raita [47] (see also [48]). In their application tries include all the substrings of length at most k (k is a constant) that exist in a text and, moreover, it is not important to support efficient access in the compressed trie. These properties can be utilized in order to further compress the trie.

In the case that the fast retrieval time of the array implementation should not be sacrificed and the tries to be compressed are static, a traditional space saving method is to find the optimal order for testing the attributes. Instead of proceeding from left to right we can proceed in the opposite direction or in fact any order we wish. The number of nodes, and thus the space needed for the trie, depends on the order in which we test the attributes. The problem of finding the optimal order is studied by Comer and Sethi [16] and elaborated further by Comer [49-51].

One can also consider the given set of node arrays as a sparse matrix to be compressed. Several heuristics for this situation can be found from [15] (see also [52-53]). In what follows we describe the method of Tarjan and Yao [17]. A data structure based on these ideas is presented also in [54].

Sparse matrices can be compressed into one-dimensional arrays such that several positions of the original matrix are represented by a single position in the resulting array provided that at most one of the original positions in question has a non-null content. If the non-null elements are conveniently distributed a simple heuristic is quaranteed to perform well. For presenting this result we need some notation. Let A be a matrix with m non-null elements and let m(t), $t \geq 0$, denote the total number of non-null elements in the rows of A which contain $t + 1$ or more non-null elements. The matrix A is said to have the *harmonic decay property* if $m(t) \leq m/(t + 1)$. If A has the harmonic decay property, at least half of its non-null elements must be in rows with only a single non-null element. Moreover, no row can have more than $\sqrt{n}$ non-null elements.

The following heuristic (often called *Ziegler's first-fit method*) gives a good compression result for matrices having the harmonic decay property:

i) Sort the rows in non-increasing order according to the number of non-null elements in each row.

ii) Using the order of rows fixed in step 1 determine a minimal row displacement rd(i) for each row i such there are no collisions with the previously placed rows.

Each row displacement r(i) now satisfies $0 \leq \mathrm{rd(i)} \leq \mathrm{m}$ [17,24].

An obvious disadvantage of the method is that not all sets of node arrays form a matrix having the harmonic decay property. If we have a matrix not fulfilling this property we must define a set of appropriate column displacements which increase the number of rows but, at the same time, create extra null elements among the original rows. Tarjan and Yao [17] provided a method for choosing the column displacements such that the resulting matrix will contain $O(\mathrm{n \log \log n})$ rows, where n is the number of rows (node arrays or non-null elements in our application) in the matrix to be compressed. When the resulting matrix fulfilling the harmonic decay property is compressed by the first-fit method, a structure with space requirement $O(\mathrm{n \log \log n})$ is obtained. A further reduction to $O(\mathrm{n})$ is still possible by observing that at most n of the rows contain non-null elements and by packing several small numbers (indicating the row displacements) into one storage location [17,24].

8.2 Implicit Dictionary

The design of implicit dictionaries is an example of data optimization where the goal is to manipulate some data items in the first n locations of an array as efficiently as possible without using more than a constant amount of extra space for pointers, counters, etc. In this section we survey various ways of implementing an implicit dictionary, i.e. a data structure supporting the operations insert, delete and search.

The implicit dictionary problem was first studied by Munro and Suwada [18]. They presented an implementation that guarantees $O(\log \mathrm{n})$ search time and $O(\mathrm{n}^{1/2}\log \mathrm{n})$ insert/delete time or alternatively $O(\mathrm{n}^{1/3}\log \mathrm{n})$ for all operations. These results were later improved by Frederickson [55] and Munro [19]. Frederickson was able to reduce the update times to $O(2^{\sqrt{2/\log \mathrm{n}}} \log^{3/2} \mathrm{n})$ while still maintaining the $O(\log \mathrm{n})$ bound for searches. Munro's implicit dictionary supports any operation in $O(\log^2 \mathrm{n})$ time.

In the static case the problem can be easily solved by a sorted array. Searches are then done in $O(\log \mathrm{n})$ time by a binary search although updates require linear time. However, the static structure can be converted into a semi-dynamic structure supporting efficient searches and insertions by using the general *binary transform* proposed by Bentley and Saxe [56]. The idea is to maintain a set of sorted arrays (*segments*) each whose size is a power of two. These are then stored sequentially according to size such that the largest segment is leftmost in the array. The binary representation of n determines the segment sizes as follows. If the i^{th} bit of n is one, we have a segment of size 2^{i} in the structure. For example, when $\mathrm{n} = 29 = 11101_2$ the structure contains segments of size 16, 8, 4, and 1.

Binary searches are made in each of these segments. The first and last items within a segment are readily computed from n. In addition, there will be at most $\lceil \log_2 n \rceil$ segments, and hence the overall search cost is $O(\log^2 n)$.

A new element is inserted as a rightmost one-item segment. If its neighbouring segment is also a one-item segment, these two are merged resulting in a two-item segment. This process is repeated for the rightmost segment until no segment pairs of equal size exist. The merge operation is performed by using some in-place merging algorithm (see for example [57]). In the worst case a single insertion might take time proportional to n. However, the cost of inserting N items is $O(N \log N)$. This is because one item can take part in at most $\lceil \log_2 N \rceil$ merges. Thus, insertions are done in $O(\log n)$ amortized time. Munro and Poblete [58] have improved this data structure such that $O(\log n)$ worst-case performance is guaranteed for insertions. The crucial idea is not to complete merges at once but distribute the work for several insertions.

Let us next consider how polylog performance is obtained for all dictionary operations. Our discussion follows the solution given by Munro [19]. The heart of the construction is a balanced search tree (for example an AVL-tree) which is implemented for sorted segments of k items. The balanced search tree keeps segments in sorted order. In order to facilitate updates for single items, an overflow area of size less than k is to be maintained between any pair of nodes. Munro [19] has proposed a way for handling the overflows so that $O(k + \log n)$ time is enough for any operation. Moreover, this can be done such that only at most k of the segments are non-full. Thus, the structure uses $n + O(k^2)$ storage locations.

How the above structure is converted to a fully implicit one? Basically, there are three issues to be addressed. First, non-full segments are compacted and maintained at the end of the array. Since a single update may involve shifting all the items in this area, the insertion and deletion costs are increased by an additive factor $O(k^2)$.

Second, the children, balancing, and overflow information of a node is encoded in the relative order of the data items. A sorted segment of k data items gives an extra storage of $\lfloor k/2 \rfloor$ bits: the order of an item pair a_{2i-1}, a_{2i} ($i=1, 2, \ldots, \lfloor k/2 \rfloor$) indicates, say, a 0-bit if $a_{2i-1} < a_{2i}$, and a 1-bit if $a_{2i-1} > a_{2i}$. For example, the bit sequence 11101 is embedded into the sequence (1, 4, 5, 8, 11, 17, 20, 22, 29, 31) by rearranging it to the form (4, 1, 8, 5, 17, 11, 20, 22, 31, 29). Observe, that in spite of the local rearrangement, the data items are still easy to manipulate in sorted order. By choosing $k = c \log_2 n + d$ (for some suitable constants c and d), pointers and counters can be encoded, and the time complexity of the dictionary operations is $O(\log^2 n)$. Note that the encoding trick is used in several other algorithms (see for example [58,59]).

Third, we must know n beforehand in order to be able to fix the proper value for the parameter k. The growth and shrinkage of n is handled by maintaining $O(\log \log n)$ separate structures of sizes 2^{2^i} ($i = 1, 2, \ldots, \lfloor \log \log n \rfloor$) plus one more of the appropriate size. The sizes are again recovered directly from n. Due to the double exponential growth of the sizes, the cost of a search in each of these structures is dominated by that of the largest. Insertions are made on the last structure, and a deletion is accomplished by moving an item from the last structure into the one in which the item to be deleted is found.

As a final comment on implicit dictionaries, we like to emphasize that the encoding of the pointers and counters is possible only when the data items are distinct. Note however that the method based on the binary transform works for any set of items. It is still an open problem whether the complexity bound $O(n^{1/2})$ for any dictionary opreration [18] can be improved in the case that equal data items are allowed.

9. Concluding Remarks

We have seen that 2n bits are enough for storing the structure of a binary tree on n nodes. Even better results can be obtained if there are regularities in the structure. If we know that a tree is regular we can encode its structure using only n bits. By using encoding methods that physically keep the information related to a node together (such as the children pattern sequence), in conjunction with an appropriate compression method, we can go below 2 bits per node without knowing a priori the degree of regularities in the structure. Jacobson [4] has shown that data optimization is possible in binary trees: the neighbouring nodes are found in constant time even in the storage space of $2n + o(n)$ bits.

Natural generalizations of tree compression and optimization have corresponding considerations for graphs. Turan [60] has shown that a planar graph having n nodes can be encoded by 12n bits. Data optimization in graphs is studied by Kannan et al. [61] and Jacobson [4]. For dense general graphs we cannot really hope for better solutions than adjacency lists [62].

We conclude this paper by listing some research problems. Dynamization is an important feature in all our example applications; it can be considered as a special case of data optimization where the operations of interest are insertion and deletion. It would be profitable if we could efficiently perform these operations in compressed structures. Aside from the applications mentioned in chapters 6 and 8, dynamization would be useful in the compression of structured text databases (for a related work see [63]).

Our formulation of the tree compression problem states that we should always map a tree structure "as compactly as possible". In the case of tries we found that finding the optimal solution can be NP-complete. Research remains to be done on approximation algorithms related to trie compaction and similar applications.

An open problem of theoretical interest is the one of finding more efficient implicit dictionaries. For instance, is it possible to perform the search operations in time $O(\log n)$ while keeping the polylog complexity for the update operations? Another question is to examine whether it is possible to construct an implicit dictionary in which equal keys are allowed and the product of search time (t_s) and update time (t_u) is less than linear, i.e. $t_s \times t_u < n$.

Acknowledgements

We would like to thank Robert D. Cameron, Matti Jakobsson, Heikki Mannila, Timo Raita, Markku Tamminen, Jukka Teuhola, and Richard Williams who read earlier versions of this paper and supplied many helpful comments. This work was supported by the Academy of Finland.

References

1. D.A. Lelewer and D.S. Hirschberg, "Data compression", *ACM Comput. Surv.* **19** (1987) 261-296.
2. D.A. Huffmann, "A method for construction of minimum redundancy codes", *Proc. of the IRE* **40** (1952) 1098-1101.
3. D.W. Jones, "Application of splay trees to data compression", *Comm. ACM* **31** (1988) 996-1007.
4. G. Jacobson, Succinct static data structures. Report CMU-CS-89-112, Computer Science Department, Carnegie Mellon University.
5. R. Bayer and E. McCreight, "Organization and maintenance of large ordered indexes", *Acta Inform.* **1** (1972) 173-189.
6. R. Bayer and K. Unterauer, "Prefix B-trees", *ACM Trans. Database Syst.* **2** (1977) 11-26.
7. I.H. Witten, R.M. Neal and J.G. Cleary, "Arithmetic coding for data compression", *Commun. ACM* **30** (1987) 520-540. (see also *Commun. ACM* **31** (1987) 1139-1145.)
8. J.A. Storer, *Data Compression: Methods and Theory* (Computer Science Press, Rockville, 1988).
9. Y. Cohen, M.S. Landy, and M. Pavel, "Hierarchical coding of binary images", *IEEE Trans. Pattern Anal. Mach. Intell.* **PAMI-7** (1985) 284-298.
10. E. Kawaguchi and T. Endo, "On a method of binary-picture representation and its application to data compression", *IEEE Trans. Pattern Anal. Mach. Intell.* **PAMI-2** (1980) 27-35.
11. G.G. Langdon and J. Rissanen, "Compression of black-white images with arithmetic coding", *IEEE Trans. Commun.* **COM-29** (1981) 858-867.
12. M. Tamminen, "Encoding pixel trees", *Comput. Vision Graph. Image Process.* **28** (1984) 44-57.
13. R.D. Cameron, "Source encoding using syntactic information source models", *IEEE Trans. Inform. Theory* **IT-34** (1988) 843-850.
14. J.Katajainen, M. Penttonen and J. Teuhola, "Syntax-directed compression of programs", *Softw. Pract. Exper.* **16** (1986) 269-276.
15. M. Al-Suwaiyel and E. Horowitz, "Algorithms for trie compaction", *ACM Trans. Database Syst.* **9** (1984),243-263.

16. D. Comer and R. Sethi, “Complexity of trie index construction”, *J. ACM* **24** (1977) 428-440.
17. R.E. Tarjan and A. C.-C. Yao, “Storing a sparse table”, *Comm. ACM* **22** (1979) 606-611.
18. J.I. Munro and H. Suwada, “Implicit data structures for fast search and update”, *J. Comput. Syst. Sci.* **21** (1980) 236-250.
19. J.I Munro, “An implicit data structure supporting insertion, deletion, and search in O($\log^2$ n) time”, *J. Comput. Syst. Sci.* **33**(1986) 66-74.
20. R.C. Read, “The coding of various kinds of unlabeled tree”, in *Graph Theory and Computing*, ed. R.C. Read (Academic Press, New York, 1972) pp. 153-183.
21. E. Mäkinen, “A survey on binary tree coding”, to appear in: *Comput. J.*
22. S. Zaks, “Lexicographic generation of ordered trees”, *Theoret. Comput. Sci.* **10** (1980) 63-82.
23. C.C. Lee, D.T. Lee, D.T. and C.K. Wong, “Generating binary trees of bounded height”, *Acta Inf.* **23** (1986) 529-544.
24. K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*, (Springer, Berlin, 1984).
25. D. Zerling, “Generating binary trees using rotations”, *J. ACM* **32** (1985) 694-701.
26. E. Mäkinen, “Left distance binary tree representations”, *BIT* **27** (1987) 163-169.
27. D.E. Knuth, *The Art of Computer Programming. Vol. I, Fundamental Algorithms* (Addison-Wesley, Reading, 1968).
28. J.T. Stasko and J.S. Vitter, “Pairing heaps: experiments and analysis”, *Comm. ACM* **30** (1987) 234-249.
29. E. Mäkinen, *On context-free derivations*, Acta Universitatis Tamperensis **A198**, University of Tampere, Finland.
30. S.A. Hutchins, “Data compression in context-free languages”, in *Information Processing 71*, ed. C.V. Freiman (North-Holland, Amsterdam) pp. 104-109.
31. R.A. Thompson and T.L. Booth, “Encoding of probabilistic context-free languages”, in *Theory of Machines and Computations*, eds. Z. Kohavi and A. Paz (Academic Press, New York) pp. 169-186.
32. J.F. Contla, “Compact coding of syntactically correct source programs”, *Softw. Pract. Exper.* **15** (1985) 625-636.
33. J. Katajainen, M. Penttonen, and J. Teuhola, “A Prolog prototype of a syntax-directed compression system”, Report D31, Department of Computer Science, University of Turku (1988).
34. R.G. Stone, “On the choice of grammar and parser for the compact analytical encoding of programs”, Comput. J. **29** (1986) 307-314.
35. J.F. Contla, “Compact coding methods for syntax-tables and source programs”, Ph.D. thesis, Reading University, 1981.
36. A.M.M. Al-Hussaini, “File compression using probabilistic grammars and LR parsing”, Ph.D. thesis, Loughborough University, 1983.
37. A.V. Aho, R. Sethi and J.D. Ullman, *Compilers - Principles, Techniques, and Tools*, (Addison-Wesley, Reading, 1986).

38. K. Knowlton, “Progressive transmission of grey scale and B/W pictures by simple, efficient, and loseless encoding schemes”, in Proc. IEEE **68**, pp. 885-896.
39. I. Gargantini, “An effective way to represent quadtrees”, *Comm. ACM* **25** (1982) 905-910.
40. H. Samet, “The quadtree and related hierarchical data structures”, *ACM Comput. Surv.* **16** (1984) 187-260.
41. H. Samet, H. “Data structures for quadtree approximation and compression”, *Comm. ACM* **28** (1985) 973-993.
42. C. Anedda and L. Felician, “P-compressed quadtrees for image storing”, *Comput. J.* **31** (1988) 353-357.
43. E. Mäkinen, “A linear time and space algorithm for finding isomorphic subtrees of a binary tree”, Report A-1990-5, Dept. of Computer Science, University of Tampere.
44. P. Elias, “Efficient storage and retrieval by content and address of static files”, *J. ACM* **21** (1974), 246-260.
45. G. Gonnet, *Handbook of Algorithms and Data Structures* (Addison-Wesley, Reading, 1984).
46. R. Ramesh, A.J.G. Babu and J.P. Kincaid, Variable-depth trie index optimization: theory and experimental results, *ACM Trans. Database Syst.* **14** (1989) 41-74.
47. J. Teuhola and T. Raita, Text compression using prediction, in *Proc. ACM 1986 Conference on Research and Development in Information Retrieval*, pp. 97-102.
48. T. Raita, “Topics in text compression”, Ph.D. Thesis, University of Turku, 1987.
49. D. Comer, “The difficulty of optimum index selection”, *ACM Trans. Database Syst.* **3** (1978) 440-445.
50. D. Comer, “Heuristics for trie index minimization”, *ACM Trans. Database Syst.* **4** (1979) 383-395.
51. D. Comer, “Analysis of a heuristic for full trie minimization”, *ACM Trans. Database Syst.* **6** (1981) 513-537.
52. J. Dill, “Optimal trie compaction is NP-complete”, Report 87-814, Dept. of Computer Science, Cornell University.
53. J. Katajainen and E. Mäkinen, “A note on the complexity of trie compaction”, *EATCS Bulletin* **41** (1990) 212-216.
54. A. Pastirik, “Searching on adaptive trie in practically constant time”, *Computers and Artificial Intelligence* **4** (1985)109-114.
55. G.N. Frederickson, “Implicit data structures for the dictionary problem”, *J. ACM* **30** (1983) 80-94.
56. J.L. Bentley and J.B. Saxe, “Decomposable searching problems I. Static-to-dynamic transformation”, *J. Algorithms* **1** (1980) 301-358.
57. J. Salowe and W. Steiger, “Simplified stable merging tasks”, *J. Algorithms* **8** (1987) 557-571.
58. J.I. Munro and P. Poblete, “Searchability in merging and implicit data structures”, *BIT* **27** (1987) 324-329.

59. T. Lai and D. Wood, “Implicit selection”, in: *Lecture Notes in Computer Science* 318 (Springer-Verlag, Berlin, 1988), pp. 14-23.
60. G. Turan, “On the succinct representation of graphs”, *Discrete Appl. Math.* **8** (1984) 289-294.
61. S. Kannan, M. Naor and S. Rudich, “Implicit representations of graphs”, in: *Proc. of the 20th ACM Symposium on Theory of Computer Science 1988*, pp. 334-343.
62. A. Itai and M. Rodeh, “Representation of graphs”, *Acta Inform*. **17** (1982) 215-220.
63. G. Gonnet and F.W. Tompa, “Mind your grammar: A new approach to modelling text”, in: *Proc. of the 13th Int. Conf. on Very Large Data Bases* (1987), pp. 339-346.