A Bounded-Space Tree Traversal Algorithm

D.S. Hirschberg† and S.S. Seiden†

Abstract

An algorithm for traversing binary trees in linear time using constant extra space is presented. The algorithm offers advantages to both Robson traversal and Lindstrom scanning. Under certain conditions, the algorithm can be applied to the marking of cyclic list structures. The algorithm can be generalized to handle N-trees and N-lists.

Keywords

Data structures

Introduction

Algorithms to traverse trees are in the tool chest of every good programmer. Tree traversals are used in many diverse applications, from searching to artificial intelligence. It is therefore important to be able to traverse trees in a time- and space-efficient manner. We present an algorithm which is efficient in both these considerations. The algorithm visits all nodes of an n-node tree in $O(n)$ time using $O(1)$ extra storage. We assume that the tree is represented as a collection of nodes, each of which contains some fixed number of pointers to their children, with no spare bits. Additionally, under certain conditions, we can use a variation of the algorithm to mark cyclic lists.

Background

There has been a long progression of algorithms devised to traverse trees. Perhaps the simplest of these is the basic recursive method taught to many first-year students. This method is fairly time-efficient but quite space-wasteful. The use of an explicit stack in traversal results in a slightly more space-efficient algorithm because there is no need to save local variables and return addresses. However, both of these algorithms require, in the worst case, $O(n)$ extra storage. More space-efficient algorithms are often required. Improvements on the basic traversal method rely on one of two methods: pointer inversion and threading.

The pointer inversion method is due to Schorr and Waite [8] (Knuth [2] and Standish [10] also

---

† Department of Information and Computer Science, University of California, Irvine, CA 92717.

credit Deutsch with independent discovery). This method uses the fields of nodes previously traversed to store a stack in the tree itself. The Schorr-Waite-Deutsch algorithm, as originally developed for cyclic lists, requires an extra two bits per node for traversal, but an adaptation to the case of trees makes one of these bits unnecessary. Wegbreit [11] uses a bit stack external to the list structure which requires $O(\log n)$ bits on average and $O(n)$ bits in the worst case. A further improvement was offered by Lindstrom [3], in the form of Lindstrom scanning, an algorithm which requires $O(1)$ extra storage. However, Lindstrom scanning is unable to perform the three standard tree traversals: preoder, inorder, and postorder. Lindstrom also devised a marking scheme for cyclic lists which uses no space outside of the mark bit [4]. However, this algorithm has a best case time of $O(n \log n)$ and a worst case time of $O(n2)$.

The basic threading method is due to Perlis and Thorton [6]. Their method, however, requires an extra bit for each pointer, making it an $O(n)$-space algorithm. An improvement on the basic threading method is Robson traversal [7], which threads the stack through the leftmost leaf nodes. However, both of these algorithms require that the leaf nodes have empty fields available. In some cases those fields may not be available. For example, expression trees (commonly used in interpreters and compilers) have interior nodes which represent operators, and leaves which represent variables or constants. The leaves are often stored in symbol or constant tables, which have no empty fields for threading.

Table 1 provides a summary comparison of tree traversal algorithms.

Algorithm	leaf pointer fields not required	worst space (bits)	uses node space	shared subtrees	cyclic lists	notes
Schorr-Waite	X	n	X	X	X	
Wegbreit	X	n		X	X	
Lindstrom scan	X	–				cannot do pre/in/post-order
Lindstrom mark	X	–	X	X	X	$O(n2)$ worst time
Perlis-Thorton		n	X			
Robson		–				
Sikl`ossy	X	n	X			read-only
Algorithm A	X	–				nodes may change location
Algorithm B	X	–		X	X	unordered structures only
recursive	X	$O(n)$wds		X	X	
explicit stack	X	n wds		X	X	

Note that algorithms for cyclic lists require an additional mark bit per node.

Table 1. Algorithms for tree traversal

About the Algorithm

Unlike Robson's traversal, the algorithm presented does not require leaves to have two empty pointer fields; therefore, it will work for trees where the leaves are stored differently than interior nodes. Unlike Lindstrom scanning, the algorithm knows whether it is visiting a node for the first, second, or third time. Therefore, it can be used for preoder, inorder, and postorder traversals. Furthermore, under special conditions, the algorithm also can be applied to cyclic lists. However, it should be noted that outside pointers to nodes of the tree other than the root may become obsolete.

Our algorithm might be useful in the following situation. Suppose we have a program which uses a binary tree structure to store information. Leaf nodes have a structure differing from that of internal nodes; they have no left and right pointer fields. There are no extra bits in the nodes which might be utilized for traversal. If we run out of memory, and there is no memory that can be reclaimed, then we wish to print out the information in the tree in both inorder and preorder, so that we might reconstruct the tree.

The algorithm is based on the fact that the children of any node of a tree can be rearranged to be ordered according to their addresses. This is true for many (though not all!) programming languages and architectures. If a node has pointers, L and R, to its two children and L is a higher-valued address than is R, the node can be ordered as follows. Exchange the pointers, L and R, and then exchange the contents of the nodes pointed to by L and R. The result is that L points to a different node than before but the contents of the node pointed to by the new L will be identical to the contents of the node pointed to the old L. Similarly for R. The resulting tree is semantically identical to the original (in structure and data content), and differs only in the location of some tree nodes. This is possible, but not trivial, in cyclic lists or trees which have shared subtrees, because several pointers to any node may exist. The algorithm orders the children, so that the address of the left child is always less than the address of the right child. When a node in the tree has only one child, the algorithm works the same as does normal pointer inversion. However, the algorithm always traverses the left subtree of nodes with two children. When the right subtree needs to be traversed, the left and right subtrees are exchanged, and the left is traversed. This is the key part of the algorithm. Because the pointer to the parent is always stored in the left pointer field, when returning to a node with two non-empty children the algorithm always knows which pointer is to the parent. The algorithm merely has to return the children to their proper orientation. A special pointer value, Lroot, is required to indicate the parent of the root.

More abstractly, the algorithm rearranges the non-empty children of any node so that they are ordered by their addresses. The first non-empty child pointer is traversed using the standard pointer inversion technique. Each time the algorithm visits a particular node, it permutes the non-empty child pointers cyclically, repeatedly traversing the first until the children are returned to the original configuration. It then returns to the parent.

This procedure of *child ordering* (where the memory addresses of nodes are changed) permutes values (in this case, addresses) to encode bits and is illustrative of *implicit* data structures [1],[5]. Child ordering can be applied to ordered trees and may invalidate outside pointers. We will also show a variation of this procedure using *pointer ordering* in which the memory addresses of nodes remain unchanged but the order of the contents is changed. This variation can be applied

to structures with shared subtrees and to cyclic lists, but requires the assumption that children are unordered.

A non-leaf node, *x*, is presumed to have fields left and right which (at least initially) contain pointers to the left and right sons of *x*. isleaf(*x*) evaluates to true iff *x* is a leaf node. Note that isleaf(Lroot) is False. We list some of the ways in which isleaf could be implemented.
1. check whether both pointer fields of *x* evaluate to *nil* (in this case leaves have the same structure as do non-leaf nodes).
2. check whether a tag field is set (in this case all nodes have an extra one-bit tag field).
3. check whether a data field contained within the node has value satisfying a known constraint.
4. check whether the node's address is within a predetermined range.

The algorithm uses the following variables: *c* points to the current node, *p* points to the parent of the current node, and *t* is a temporary.

ALGORITHM A

1. (Initialize)
    if root = *nil* **then goto** Step 9 **fi**
    c = root
    p ← Lroot
2. (Order children)
    if left(*c*) ≠ *nil* **and** right(*c*) ≠ *nil*
        and ¬isleaf(left(*c*)) **and** ¬isleaf(right(*c*))
        and right(*c*) < left(*c*) **then**
        exchange left(*c*) and right(*c*)
        exchange the contents of left(*c*) and the contents of right(*c*)
    fi
    preorder visit
    if left(*c*) = *nil* **then goto** Step 4 **fi**
    if isleaf(left(*c*)) **then**
        visit left(*c*)
        goto Step 4
    fi
3. (Traverse left)
    t ← left(*c*)
    left(*c*) ← *p*
    p ← *c*
    c ← *t*
    goto Step 2
4. (Finished traversing left subtree)
    inorder visit
    if right(*c*) = *nil* **then goto** Step 6 **fi**
    if isleaf(right(*c*)) **then**
        visit right(*c*)
        goto Step 6

fi

5. (Traverse right)

if left(*c*) = *nil* **or** isleaf(left(*c*)) **then**

$t \leftarrow$ right(*c*)

right(*c*) $\leftarrow$ *p*

$p \leftarrow c$

$c \leftarrow t$

goto Step 2

else

exchange left(*c*) and right(*c*)

goto Step 3

fi

6. (Finished traversing right subtree)

postorder visit

if *p* = Lroot **then goto** Step 9

if left(*p*) = *nil* **or** isleaf(left(*p*)) **then**

7. (There had been no left subtree)

$t \leftarrow c$

$c \leftarrow p$

$p \leftarrow$ right(*c*)

right(*c*) $\leftarrow t$

goto Step 6

fi

8. (There had been a left subtree)

$t \leftarrow c$

$c \leftarrow p$

$p \leftarrow$ left(*c*)

left(*c*) $\leftarrow t$

if right(*c*) = *nil* **or** isleaf(right(*c*)) **or** $t <$ right(*c*) **then**

goto Step 4

else

exchange left(*c*) and right(*c*)

goto Step 6

fi

9. (Done!)

The programmer is free to perform whatever operation he wants (as long as it does not alter the tree) during the preorder, inorder, and postorder visits.

Analysis

We note that the step “Order children" does not alter the tree in such a way that the semantics of the tree are changed. If *x* is a node in a tree, then the only pointer to the left child of *x* is left(*x*).

Likewise for the the right child. Therefore, exchanging the left and right pointers, and then their contents, does not change the meaning of the tree.

We use LEFT(x) to denote the address of the left son of node x in the original tree as adjusted by the “Order children" fragment. We use RIGHT(x) and PARENT(x) analogously. We define a tree to be trivial if it is empty or consists of a single leaf node.

We show that the algorithm is correct by induction on the size of the tree. If the given tree is empty, this is detected in Step 1, and the algorithm terminates. Otherwise, for any given node x in the tree, we show that if the traversal of the subtrees headed by the children of x is correct, then the traversal of the subtree headed by x is correct. By this it is meant that the following events occur in the following order.

(i) we reach the point marked *preorder visit*, with c pointing to x and p pointing to the parent of x.
(ii) the subtree headed by left(x) is traversed.
(iii) we reach the point marked *inorder visit*, with c and p as above.
(iv) the subtree headed by right(x) is traversed.
(v) we reach the point marked *postorder visit*, with c and p as above.

We consider the sequence of events that follow after the preorder visit in Step 2 of a node x. Each of the subtrees headed by x’s two children can be either trivial or non-trivial, thus producing four possible situations.

If x has two trivial subtrees, then Step 4 is executed, and the inorder visit performed. Finally Step 6 is executed, and the postorder visit is performed. All values are correct.

If x has a non-trivial left subtree and a trivial right subtree then Step 3 is performed. This rotates the values of p, c, and left(x). The old values were: p=PARENT(x), c=x, left(x)=LEFT(x). The new values are: p=x, c=LEFT(x), left(x)=PARENT(x). Assuming the traversal of the left subtree is correct, Step 6 will be eventually reached. Since p now contains the value x, left(p) contains the value PARENT(x) which is non-trivial. Accordingly, Step 7 will be bypassed and Step 8 will be performed, rotating the values of p, c, and left(x) back to their original locations. Since the right(x) is trivial, we go to Step 4. The inorder visit is performed and then the postorder visit is performed in Step 6. The values of p and c have been restored, as well as the values of left(x) and right(x).

If *x* has a trivial left subtree and a non-trivial right subtree then, since left(*x*) is trivial, we proceed to Step 4 where the inorder visit is performed and then to Step 5. Since the left subtree is trivial, the first section of Step 5 is performed. This rotates the values of *p*, *c*, and right(*x*). The old values were: *p*=PARENT(*x*), *c*=*x*, right(*x*)=RIGHT(*x*). The new values are: *p*=*x*, *c*=RIGHT(*x*), right(*x*)=PARENT(*x*). Assuming the traversal of the right subtree is correct, Step 6 will be eventually reached. Since *p* now contains the value *x*, left(*p*) contains the value LEFT(*x*) which is trivial. Therefore, Step 7 will be performed, rotating the values of *p*, *c*, and right(*x*) back to their original locations, and then we proceed to Step 6. Step 6 performs the postorder visit. Once again, all values are correct.

If *x* has two non-trivial subtrees then Step 3 is performed. This rotates the values of *p*, *c*, and left(*x*). The old values were: *p*=PARENT(*x*), *c*=*x*, left(*x*)=LEFT(*x*). The new values are: *p*=*x*, *c*=LEFT(*x*), left(*x*)=PARENT(*x*). Assuming the traversal of the left subtree is correct, Step 6 will be eventually reached. Since *p* now contains the value *x*, left(*p*) contains the value PARENT(*x*) which is non-trivial. Accordingly, Step 7 will be bypassed and Step 8 will be performed, rotating the values of *p*, *c*, and left(*x*) back to their original locations. Since the neither left(*x*) or right(*x*) are trivial and the address of left(*x*) is less than the address of right(*x*) we know that the right subtree has yet to be traversed and proceed to Step 4. The inorder visit is performed. The values of left(*x*) and right(*x*) are exchanged in Step 5. Step 3 is then performed. This rotates the values of *p*, *c*, and left(*x*). The old values were: *p*=PARENT(*x*), *c*=*x*, left(*x*)=RIGHT(*x*). The new values are: *p*=*x*, *c*=RIGHT(*x*), left(*x*)=PARENT(*x*). Assuming the traversal of the left subtree (which is really the right subtree) is correct, Step 6 will be eventually reached. Since *p* now contains the value *x*, left(*p*) contains the value PARENT(*x*) which is non-trivial. Accordingly, Step 7 will be bypassed and Step 8 will be performed, rotating the values of *p*, *c*, and left(*x*) back. The old values were: *p*=*x*, *c*=RIGHT(*x*), left(*x*)=PARENT(*x*). The new values are: *p*=PARENT(*x*), *c*=*x*, left(*x*)=RIGHT(*x*). Since neither left(*x*) nor right(*x*) are trivial and the address of left(*x*) is greater than the address of right(*x*) we re-exchange left(*x*) and right(*x*), returning them to their original locations, and perform the postorder visit. All values are once again correct, and the proof is concluded.

Potential Variations

Algorithm A, in a slightly different form, can be used to mark cyclic list structures, if one is willing to forego having a predefined order on a node's children. That is, despite the fact that the pointer fields within a node are ordered (there is a first pointer and a second, etc.), the fields may

be permuted, destroying the information inherent in their positioning.

ALGORITHM B

B1. (Initialize)
    if $c = nil$ **then goto** Step B6 **fi**
    $p \leftarrow nil$
B2. mark(c)
    if left(c) = *nil* **or** marked(left(c)) **then goto** B4 **fi**
B3. (Traverse left)
    $t \leftarrow$ left(c)
    left(c) $\leftarrow p$
    $p \leftarrow c$
    $c \leftarrow t$
    goto Step B2
B4. (Traverse right)
    if right(c) = *nil* **or** marked(right(c)) **then goto** Step B5 **fi**
    exchange left(c) and right(c)
    goto Step B3
B5. (Traverse up)
    if $p = nil$ **then goto** Step B6 **fi**
    $t \leftarrow c$
    $c \leftarrow p$
    $p \leftarrow$ left(c)
    left(c) $\leftarrow t$
    goto Step B4
B6. (Done!)

It is easy to see that Algorithm B works correctly. We define a cycle-causing link to be a pointer from a node to one of its ancestors, or to itself. Without the cycle-causing links, any list is merely a tree (perhaps with shared subtrees). Algorithm B never traverses any cycle-causing links. This is because, at any point in the traversal, all the ancestors of the current node are marked, as is the current node itself. Algorithm B never revisits nodes which are marked. The fact that there may be shared subtrees makes no difference, because this algorithm does not reorder nodes. Since the pointer to the parent is always stored in the pointer to the left child, it is always possible to correctly restore the p pointer. This also makes the use of Lroot unnecessary. However, it is impossible to always correctly restore the original orientation of the children. The left and right children may have been exchanged, but this cannot be determined. The algorithm can be generalized to handle trees and lists with N children.

Marking algorithms are generally used for garbage collection. The best previous algorithm for

cyclic list structures using bounded space is Lindstrom marking [4]. In many situations, list structures may be used to represent graphs. For instance, interference graphs used for register allocation in compilers might be represented as lists. In such a situation the ordering of edges is unimportant. Our algorithm has the the advantage of linear time in such situations.

References

[1] Frederickson, G.N. "Implicit data structures for the dictionary problem," *Journal ACM* **30**, 1 (Jan. 1983), 80-94.

[2] Knuth D. E. *The Art of Computer Programming* Vol. 1, Addison-Wesley, Reading, Massachusetts, 1973, 634 pp.

[3] Lindstrom G. "Scanning list structures without stacks or tag bits," *Info. Proc. Letters* **2**, (1973) 47-51.

[4] Lindstrom G. "Copying list structures using bounded workspace," *Comm. ACM* **17**,4 (1974) 198-202.

[5] Munro, J. I. "An implicit data structure for the dictionary problem that runs in polylog time," *Proc. 25th Annual Symp. on Foundations of Computer Science*, 1984, 369-374.

[6] Perlis A. J., Thorton C. "Symbol manipulation by threaded lists," *Comm. ACM* **3**,4 (April 1960) 195-204.

[7] Robson J. M. "An improved algorithm for traversing binary trees without auxiliary stack," *Info. Proc. Letters* **2**, (1973) 12-14.

[8] Schorr H. and Waite W. M. "An efficient machine independent procedure for garbage collection in various list structures," *Comm. ACM* **10**,8 (Aug. 1967), 501-506.

[9] Siklòssy L. "Fast and read-only algorithms for traversing trees without an auxiliary stack," *IPL* **1**, (1972), 149-152.

[10] Standish T. A. *Data Structure Techniques*, Addison-Wesley, Reading, Massachusetts, 1980, 447pp.

[11] Wegbreit B. "A space-efficient list structure tracing algorithm," *IEEE Trans. Comp. C-21* **9**, (Sept. 1972), 1009-1010.