Information Processing Letters 23 (1986) 209–214
North-Holland

8 November 1986

ELIMINATING THE FLAG IN THREADED BINARY SEARCH TREES

Dan GORDON *

Department of Computer Science, University of Cincinnati, Cincinnati, OH 45221, U.S.A.

Communicated by David Gries
Received 29 May 1985
Revised 6 September 1985

Keywords: Binary search tree, binary tree, multiprocessor environment, shared memory, threaded binary tree

1. Introduction

Right in-threaded binary trees, introduced by Perlis and Thornton [3], offer the advantage of inorder traversal without a stack (see also [1,5]). In such trees, the right pointer field of a node with no right son contains a pointer to the node's inorder successor (the pointer is then called a *thread*). In order to distinguish between a thread and a right-son pointer, a 1-bit Boolean flag is used. In many applications, the flag can be worked into the node's existing fields (e.g., as the sign bit of an always-positive field), but in others the flag must be added as an extra field. On most computers, this requires a byte, or even a larger word, and the attractiveness of the scheme diminishes —especially in applications where space is at a premium.

Morris [2] and Robson [4] proposed solutions for inorder traversal without flags, back-pointers or stack (see also [1, p. 281]). The underlying principle in both is that certain pointers are modified during traversal to keep track of the path. This solution is unacceptable in applications where read-only access is allowed. Such a restriction can occur with read-only memory or in cases where a single data set is shared by several processors that may access it simultaneously.

In this paper we demonstrate how threading can be used without flags in a *binary search tree* [1,5]. We prove a basic property of binary trees that allows us to distinguish efficiently between a thread and a right-son pointer during searching or insertion. The same property also yields a method for identifying a thread during inorder and preorder traversal of binary search trees with constant extra space.

We also consider doubly-threaded binary search trees [1,5], which, in addition to right threads, also contain left threads, which point at a node's inorder predecessor when the node has no left son. Traversing a path from the root to some node, for example for searching and insertion, can be done on such trees in a manner similar to regular (right in-) threaded trees. For inorder or preorder traversal, the above-mentioned method for identifying a thread does not work, but we prove that inorder traversal is possible without distinguishing between a thread and a right-son pointer.

Our algorithms are simple modifications of the regular ones for threaded trees and exhibit the same time complexity. There is no modification of pointers (or any other fields) in any of the algorithms, so they are suitable for shared memory in a multiprocessing environment.

2. An elementary property of binary trees

Assume that every node P has two fields LEFT and RIGHT pointing at P's left and right sons

* Present address: Department of Computer Science, School of Mathematical Sciences, Tel-Aviv University, Tel-Aviv 69978, Israel (on leave from the University of Haifa).

0020-0190/86/$3.50 © 1986, Elsevier Science Publishers B.V. (North-Holland)

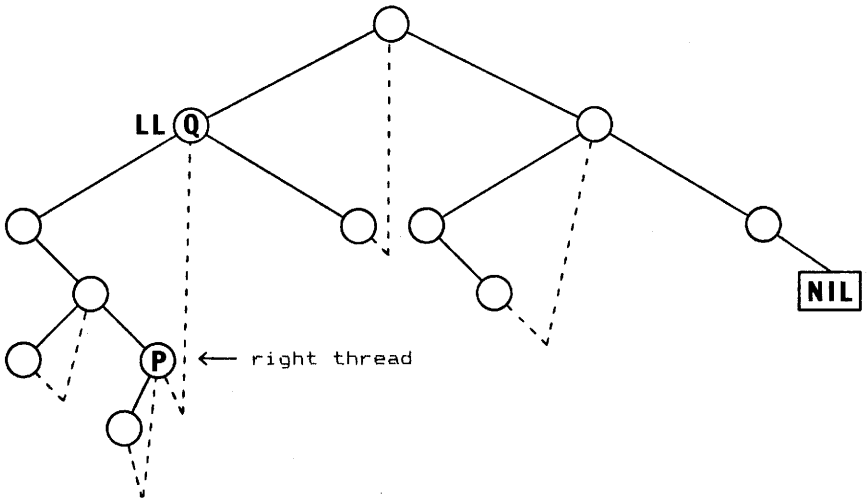

Fig. 1. A threaded binary tree. If node P has no right son, its inorder successor Q is the last node on the path from the root to P at which the left branch is taken.

respectively, if they exist. We say that the left branch is taken at a node N_i of a path $(N_1, N_2, \ldots, N_n)$ if $N_{i+1} = N_i$.LEFT; otherwise, the right branch is taken. Our technique is based on the following result, which is illustrated in Fig. 1.

Theorem 2.1. *Let node* P *of a binary tree have no right son. Node* Q *is* P*'s inorder successor iff* P *is in* Q*'s left subtree and, on the path from the root to* P, Q *is the last node at which the left branch is taken.*

Proof. Let $<$ denote the inorder relation among nodes. Assume that Q is P's successor. Since $P < Q$, one and only one of the following cases is true:

(1) P is in Q's left subtree.

(2) Q is in P's right subtree.

(3) For some third node R, P is in R's left subtree and Q is in R's right subtree.

To see why (3) must hold if (1) and (2) are false, take R to be the lowest-level common ancestor of P and Q.

Because P's right subtree is empty, (2) is impossible. Case (3) is also impossible because otherwise we would have $P < R < Q$, contradicting the assumption that Q is P's inorder successor. This leaves (1) as the only possibility.

From (1) it follows that Q is on the path from the root to P and that the left branch is taken at Q. Assume now that the left branch is taken at a third node R on the path from Q to P. Then, $P < R < Q$, again contradicting the assumption that Q is P's successor. This completes the proof in one direction.

Assume now that P is in Q's left subtree and that Q is the last node at which the left branch is taken on the path from the root to P. Then $P < Q$, so P is not the last inorder node. Let R be P's successor. By what has been proved above, R is the last node on the path from the root to P at which the left branch was taken. Since there is only one path from the root to any node in a binary tree, $R = Q$. □

3. Searching and insertion

Assume that we have a right in-threaded binary search tree, i.e., if node P does not have a right son, then P.RIGHT is P's inorder successor or NIL if P is the last inorder node (as in Pascal, NIL is a null pointer value).

Searching for a node or inserting a node in a binary search tree calls for traversing a path from the root to some place in the tree. This can be done using Theorem 2.1 in the following manner.

At any point in the traversal a pointer variable LL (Last Left) points at the last node at which the left branch was taken. Thus, initially LL = NIL, and taking a left branch at node Q calls for the assignment LL := Q. This is illustrated in Fig. 1.

Assume now that in the traversal a node P has been reached and let Q = P.RIGHT. If Q = NIL, then P is the last inorder node. Otherwise, we determine if Q is P's inorder successor or right son by comparing Q with LL. If Q = LL, then, by Theorem 2.1, Q is P's inorder successor (see Fig. 1); otherwise, Q is P's right son. All other operations for searching and insertion are done in the usual way for threaded binary trees.

4. Inorder and preorder traversal

For these traversals we need the fact that the tree is a binary search tree: each node has some field KEY and a linear order < is defined between KEY's so that, for any nodes P and Q, P.KEY < Q.KEY iff P precedes Q in inorder.

Traversal in inorder or preorder is performed as usual for threaded trees [1,5]; it is only necessary to be able to determine, for any node P, whether P.RIGHT is NIL, P's right son, or P's inorder successor. To do this requires the fact that the tree is a binary search tree. Let Q = P.RIGHT and, provided Q ≠ NIL, let R = Q.LEFT. Our technique is based on examining R.

If Q = NIL, then P has no right son and no inorder successor. Otherwise, we are left with four mutually exclusive, exhaustive cases (see Fig. 2):

(a) R = NIL (see Fig. 2(a)): Q has no left son, so, by Theorem 2.1, Q is not P's inorder successor. Hence, Q is P's right son.

(b) R ≠ NIL and R = P (see Fig. 2(b)): Q is not P's right son, because then P would be its own proper descendant, so Q is P's inorder successor.

(c) R ≠ NIL, R ≠ P, and R.KEY < P.KEY (see Fig. 2(c)): Q is P's inorder successor, as is proved below.

(d) R ≠ NIL, R ≠ P, and R.KEY > P.KEY (see Fig. 2(d)): Q is P's right son, as is proved below.

Proof of the claims in (c) and (d). Assume R ≠ NIL and R ≠ P. If Q is P's successor (but not its right son, see Fig. 2(c)), then, by Theorem 2.1, the path from the root to P turns left at Q, so it passes through R. The path turns right at R because Q is the last node at which it turns left. Hence, P is in R's right subtree, and, by the binary search property, R.KEY < P.KEY. If Q is P's right son (see Fig. 2(d)), then all KEY values in the subtree rooted at Q are greater than P.KEY; in particular, R.KEY > P.KEY. The claims made in (c) and (d) now immediately follow. □

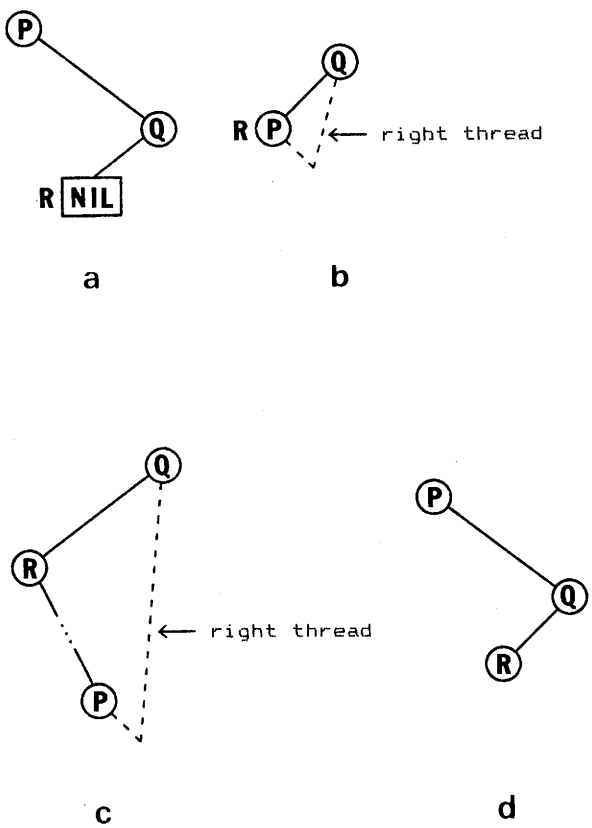

Fig. 2. Determining whether a node P's right pointer is a thread during traversal: Q = P.RIGHT, R = Q.LEFT. (a) R = NIL—Q is P's right son. (b) R = P—Q is P's inorder successor. (c) R.KEY < P.KEY—Q is P's inorder successor. (d) R.KEY > P.KEY—Q is P's right son.

Hence, by examining R and P only we can determine whether P.RIGHT is a thread. Cases (b) and (c) can be combined into R ≠ NIL and R.KEY ≤ P.KEY, so just one extra test may be required to identify a thread compared to the use of a flag.

5. Doubly threaded binary search trees

In these trees (see [1,5]), the right pointers are used in the same way as in right in-threaded trees. The left pointer of a node P is used in a symmetrical manner: P.LEFT is P's left son if it exists; otherwise, P.LEFT is P's inorder predecessor (or NIL if P is the first inorder node).

In a manner similar to Theorem 2.1, we can prove the equivalent statement for inorder predecessors: if P is not the first inorder node and has no left son, then Q is P's inorder predessor iff P is in Q's right subtree and, on the path from the root to P, Q is the last node at which the path turns right.

Searching and insertion can be done by extending our technique so that left and right branches are handled in a symmetric manner: besides variable LL (see Section 3), we maintain a variable LR (for Last Right), which always contains the last node from which the right branch was taken. LL helps identify a right thread, and LR a left thread.

For inorder or preorder traversal, the previous technique for identifying a right thread does not work, because the left pointer is not NIL when there is no left son (except for the leftmost node of the tree). However, identifying a thread is unnecessary for inorder traversal. This is shown by the following theorem, which forms the basis for our inorder traversal algorithm.

Theorem 5.1. *Let* P *be a node (not the last in inorder) in a doubly threaded binary search tree and* L_0 = P.RIGHT. *Then there exist nodes* $L_0, L_1, \ldots, L_k$, $k \geqslant 1$, *such that*

(1) $L_{i+1} = L_i$.LEFT *for* $0 \leqslant i < k$,
(2) L_i.KEY > P.KEY *for* $0 \leqslant i < k$,
(3) L_k.KEY $\leqslant$ P.KEY, *and*
(4) L_{k-1} *is* P's *inorder successor.*

Proof. L_0.KEY > P.KEY because L_0 is P's right son or inorder successor. Let $L_1 = L_0$.LEFT, and consider the following four cases, illustrated in Fig. 3:

(a) L_1 = P and L_0 is a thread (see Fig. 3(a)): then L_1.KEY = P.KEY and the statement holds with $k = 1$.

(b) L_1 = P and L_0 is P's right son (see Fig. 3(b)): again, L_1.KEY = P.KEY and the statement holds with $k = 1$; note that L_0 is also P's inorder successor in this case.

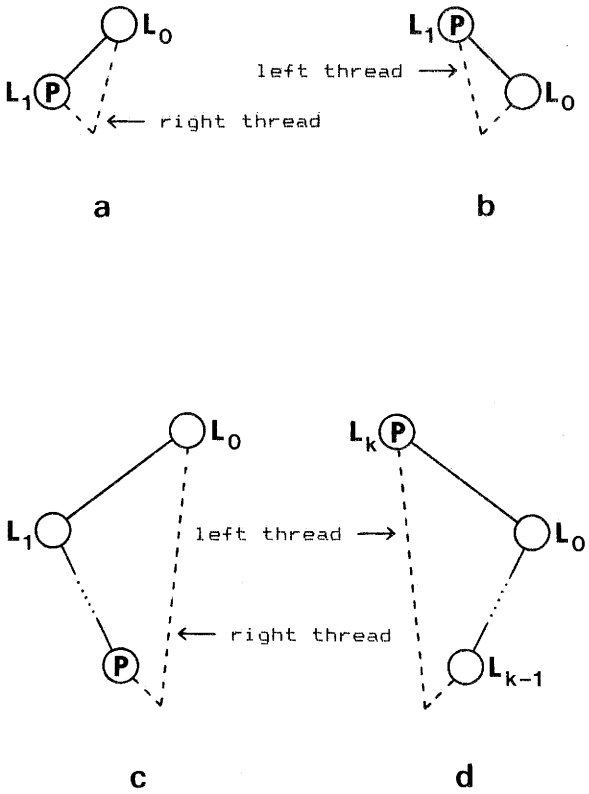

Fig. 3. P's inorder successor in a doubly-threaded tree is reached by moving right to L_0 and then left to $L_1, \ldots, L_k$ until L_k.KEY $\leqslant$ P.KEY; the successor is L_{k-1}. (a) L_1 = P and L_0 is a thread—k = 1. (b) L_1 = P and L_0 is P's right son—k = 1. (c) $L_1 \neq$ P and P.RIGHT is a thread—k = 1. (d) $L_1 \neq$ P and L_0 is P's right son—k is the number of nodes in the left-most branch rooted in L_0.

(c) $L_1 \neq$ P and P.RIGHT is a thread (see Fig. 3(c)): then, by the argument used in the proof of cases (c) and (d) of Section 4, L_1.KEY < P.KEY and the statement again holds with $k = 1$.

(d) $L_1 \neq$ P and L_0 is P's right son (see Fig. 3(d)): take k to be the number of nodes in the longest leftmost path of the subtree rooted at L_0, and define $L_2, \ldots, L_k$ according to (1); $L_0, \ldots, L_{k-1}$ form that entire branch, they are in P's right subtree and so their KEY's are > P.KEY; furthermore, L_{k-1} is P's successor and so its left

```
Inorder traversal of a nonempty doubly threaded binary tree
"Set P to the first inorder node:"
  P := Root of tree;
  while P.LEFT ≠ NIL do P := P.LEFT;

VISIT(P); L := P.RIGHT;

{invariant: node P is the last node visited in inorder and L is the corresponding value L0 of Theorem 5.1 (each iteration
   visits one node)}
while L ≠ NIL do
  begin "Change P to its inorder successor, using Theorem 5.1:"
          while L.LEFT.KEY > P.KEY do L := L.LEFT;
          {L is node Lk-1 of Theorem 5.1}
          P := L;
        "Visit node P and reestablish the invariant:"
          VISIT(P); L := P.RIGHT
  end
```

Fig. 4.

pointer is a thread pointing at P; therefore, $L_k = P$ and L_k.KEY = P.KEY. □

Our inorder traversal algorithm starts from the leftmost node, follows the right pointer and then a succession of left pointers until it reaches the L_k of Theorem 5.1. L_{k-1} is then the next inorder node, and the algorithm continues until the right pointer is NIL. The algorithm is presented in Fig. 4. VISIT(P) is assumed to be some procedure to be performed on each node P in inorder.

The correctness of the algorithm is a straightforward corollary of Theorem 5.1. Notice that when executing L := P.RIGHT, we may be moving to the right son or to the inorder successor, and the algorithm is oblivious to the difference (i.e., the same code is executed in either case).

6. Concluding remarks

Our traversal algorithms are simple variations of the regular algorithms for threaded trees with flags. The only differences are the following: For right in-threaded trees, we differ in our method of identifying a thread, but require only one extra test (which is not always evaluated). For doubly threaded trees, our traversal algorithm only differs in the need to go one extra node to the left (to L_k of Theorem 5.1) and compare its KEY value to the last KEY value visited. Furthermore, each thread and each real edge of the tree is traversed exactly once by the algorithm, and there are $2n - 2$ such threads and edges, where n is the number of nodes. It follows that the time complexity of our traversal algorithms is of the same order as that of the regular algorithms, which is O(n).

Our technique for insertion or searching, in both types of threaded trees, is very simple, but may take a little longer than the one for flagged threads, due to maintaining LL (and LR). Overall, our algorithms are slightly slower than the ones for threaded trees with flags. For any planned implementation, this must be weighted against the saving achieved by eliminating the flags. Examples of such considerations are (i) restricted memory, and (ii) a paging environment, where the saved space could result in fewer page fetches and hence an even improved throughput.

Acknowledgment

The author is most grateful to David Gries, whose suggestions and comments on a previous version of this paper have largely improved its presentation.

References

[1] E. Horowitz and S. Sahni, Fundamentals of Data Structures (Computer Science Press, Potomac, MD, 1976).
[2] J.M. Morris, Traversing binary trees simply and cheaply, Inform. Process. Lett. 9 (5) (1979) 197–200.
[3] A. Perlis and C. Thornton, Symbol manipulation by threaded trees, Comm. ACM 4 (3) (1960) 195–204.
[4] J. Robson, An improved algorithm for traversing binary trees without auxiliary stack, Inform. Process. Lett. 2 (1) (1973) 12–14.
[5] A.M. Tanenbaum and M.J. Augenstein, Data Structures Using Pascal (Prentice-Hall, Englewood Cliffs, NJ, 1981).