INFORMATION PROCESSING LETTERS 2 (1973) 12–14. NORTH-HOLLAND PUBLISHING COMPANY

AN IMPROVED ALGORITHM FOR TRAVERSING BINARY TREES WITHOUT AUXILIARY STACK

J.M. ROBSON

Department of Comp. Studies, University of Lancaster, Bailrigg, Lancaster LA1 4YN, England

Received 14 March 1973

trees binary trees tree traversal algorithms traversals without stacks

Knuth [1] describes an algorithm for traversing the nodes of a binary tree in preorder, symmetric order or endorder without using an auxiliary stack. The motivation for doing this lies in the fact that if a stack is used, the number of items on it may be as great as the number of nodes in the tree. Knuth's algorithm uses a standard representation of the tree in which each node includes two links to its sons and also a one bit field for use in the traversal process. In non-terminal nodes the links and the extra bit are altered during the traversal but the links are restored to their original values.

Siklossy [2] has shown that this altering of nodes can be avoided by using a special representation of the tree which again needs two link fields and an extra bit, and rules out sharing of subtrees either within one tree or between trees. He asks whether the use of the extra bit can be avoided possibly at the expense of the speed or of the read-only properties of his method. This paper answers that question affirmatively by giving an algorithm similar to Knuth's which uses Knuth's representation without the extra bit and which modifies the link fields of terminal nodes as well as those of non-terminals.

The extra bits required by each of the previous algorithms are used when ascending through the tree to determine whether the return was from a left or right subtree. The idea behind the present method is to decide this by means of a linked stack kept in the link fields of terminal nodes which have already been visited. The elements of this stack are pointers to those nodes which have a non-nil left subtree which has already been visited and a non-nil right subtree which is currently being visited. When during the ascent of any branch of the three a node is found with two non-nil subtrees, the ascent was from the left unless the node is pointed to by the top of the stack.

The algorithm as shown in fig. 1 applies the procedure *previsit*, *symvisit* and *endvisit* to each node of the tree in preorder, symmetric order and endorder respectively. (The procedures used for test purposes merely list the information fields in three columns; in practise one would probably be the required action and the other two would be absent.) The way in which control passes between the sections labelled *up* and *down* makes it clear that *avail* always points to a new terminal node when one is needed for the stack. The program is written in ALGOL68-R [3] and has been tested on the Lancaster University ICL 1905F computer.

References

[1] D.E. Knuth, Fundamental Algorithms (Addison–Wesley, 1968), p. 562.

[2] L. Siklossy, Fast and Read-only Algorithms for Traversing Trees without an Auxiliary Stack, Information Processing Letters, 1 (1972) 149–152.

[3] P.M. Woodward and S.G. Bond, ALGOL 68-R Users Guide.

```
BEGIN MODE TREE = STRUCT (REF TREE llink, rlink, INT info);
OP == (REF TREE x, y) BOOL : (info OF x IS info OF y);
C
do x and y refer to the same tree??
C
   REF TREE nil = NIL      ;
   PROC previsit = (TREE t):(print          ((info OF t, newline)));
   PROC symvisit = (TREE t):(print(("    ", info OF t, newline)));
   PROC endvisit = (TREE t):(print(("    ", info OF t, newline)));
PROC scan = (REF TREE t):
   BEGIN
   REF TREE temp, stack, avail, p := t, q := t, top := nil;


   down:
      IF (llink OF p IS nil) AND (rlink OF p IS nil)
         THEN
C
this terminal node will be the next one used for stack
C
         avail := p;
         previsit(p); symvisit(p); endvisit(p)
      ELSF llink OF p IS nil
         THEN
         previsit(p); symvisit(p);
         temp := rlink OF p; rlink OF p := q;
         q := p; p := temp;
         GOTO down
      ELSE
         previsit(p);
         temp := llink OF p; llink OF p := q;
         q := p; p := temp;
         GOTO down
      FI;
      up:
         IF p IS t THEN SKIP
         ELSF llink OF q IS nil
C
if either link is nil there is no doubt about whether return was from left or right subtree
C
         THEN
         endvisit(q);
```

```
        temp := rlink OF q; rlink OF q := p;
        p := q; q := temp;
        GOTO up
        ELSF rlink OF q IS nil
        THEN
        symvisit(q); endvisit(q);
        temp := llink OF q; llink OF q := p;
        p := q; q := temp;
      GOTO up
    ELSF q = top
        THEN
C
q is on stack so return is from right subtree: pop stack and keep on up
C
        endvisit(q);
        temp := stack;
        top := rlink OF stack;
        stack := llink OF stack;
        llink OF temp := rlink OF temp := nil;
      temp := llink OF q; llink OF q := rlink OF q;
        rlink OF q := p; p := q; q := temp;
        GOTO up
      ELSE
C
having returned from left subtree, pack this node and scan right subtree
C
        symvisit(q);
        llink OF avail := stack;
        rlink OF avail := top;
        stack := avail;
        top := q;
        temp := rlink OF q; rlink OF q := p; p := temp;
        GOTO down
      FI
    END;
SKIP
C
dummy block body
C
END
```

Fig. 1