# TRAVERSING BINARY TREES SIMPLY AND CHEAPLY

Joseph M. MORRIS

*Department of Computer Science *, Trinity College, Dublin 2, Ireland and Department of Mathematics, Technische Hogeschool, Eindhoven, The Netherlands*

Received 17 August 1979; revised version received 3 October 1979

Binary trees, tree traversal

## 1. Introduction

We describe an algorithm, and variations on it, for traversing a binary tree simply and efficiently. The algorithm is simple in its operations and, arguably, in its ease of comprehension. It is also efficient, taking time proportional to the number of nodes in the tree and requiring neither a run-time stack nor 'flag' bits in the nodes.

Binary trees and algorithms for their traversal are described in many textbooks, for example [1]. Many of these algorithms are designed to be economical in their use of additional storage, but that design goal has been notoriously hard to achieve in practice. Indeed Waite [2] has speculated: "the minimum information required by a non-recursive tree traversal that does not permanently alter the data structure seems to be three addresses plus one bit per level". The present algorithm uses no auxiliary storage other than two pointers.

## 2. Inorder traversal

In this section we discuss inorder traversal: traverse the left subtree; process the root; traverse the right subtree. We begin with some terminology.

Each node in a tree has a unique name (in machine terms, an address), the empty node having the name **nil**. A 'pointer' is a variable of type 'node name'.

* Current address.

Each node t has two fields of interest, the left and right pointer fields, which are denoted by t.l and t.r, respectively. A pointer field (or 'link') whose value is not **nil** corresponds to an 'edge' of the tree. We abbreviate 'the (sub)tree whose root is t' to '(sub)tree t', and 'binary tree' to 'tree'. In this section 'traversal' means 'inorder traversal'.

Suppose we are confronted with traversing (sub)-tree t. If t is **nil** we are done. Otherwise, if t.l is **nil** we can make progress by processing node t and proceeding to traverse subtree t.r. If t.l is not **nil**, however, we cannot make progress by processing node t. In this case we will seek salvation by looking for a transformation of the tree, postponing for the moment considerations of restoring its original status. We will attempt to make progress by decreasing – by 1 will be sufficient – the number of left edges (and, concomitantly, increasing by 1 the number of right edges), with the proviso that the modified tree has a similar inorder traversal. In short, we are processing with a loop whose monotonically decreasing function is the number of nodes still to be processed plus the number of left edges, and whose invariant is 'it remains to traverse tree t'.

Let tree t in Fig. 1 be the tree to be traversed. The triangles represent possibly empty subtrees. The circles represent distinct nodes except that nodes X2 and X4 (and their left subtrees) may coincide, in which case node X3 and its left subtree disappear. Tree t is transformed into tree t′ as follows: the 'rightmost' node of subtree t.l (whose right pointer is **nil**) acquires node t and subtree t.r as its right sub-

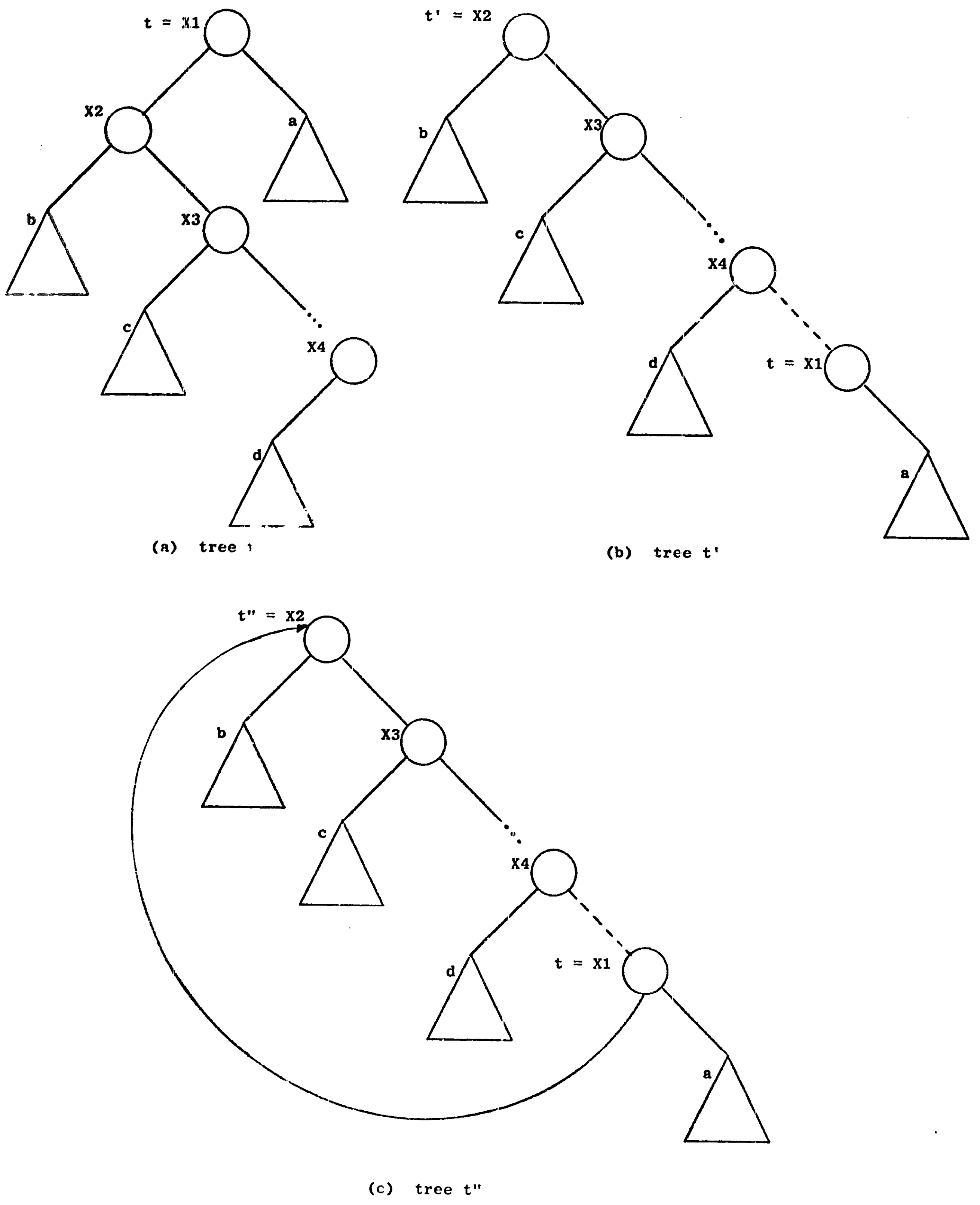
t = X1
X2
a
b
X3
c
X4
d
(a) tree t
t' = X2
b
X3
c
X4
d
t = X1
a
(b) tree t'
t" = X2
b
X3
c
X4
d
t = X1
a
(c) tree t"

Fig. 1.

tree, and node t acquires an empty left subtree. The dashed line in tree t′ represents the added (right) edge. Not only has tree t′ one less left edge, but (we submit without further argument) its traversal is the same as that of tree t.

The preceding analysis leads immediately to a traversal algorithm, but one that destroys the original structure of the tree. In order to restore the original tree we need a history of the transformations, necessitating a stack – but that is a path along which we do not want to travel. Instead, let us investigate the possibility of not deleting the left edge of node t during the transformation; tree t is transformed into tree (graph) t″ instead of t′. This gives us a hope of restoring the original tree because the task of restoration is now just one of deleting information from the structure – specifically, the added links. We will see this hope evolve into a certainty.

Assuming that we transform tree t into tree t″, when processing t″ further we must be able to detect in some manner that the left edge t.l should, for purposes of further traversal, be treated as though it were **nil**. Instead of actually marking node t (e.g., using a bit in the node), we prefer to use some predicate, marked(t), equivalent to 'node t is marked'. Now, note that a transformation introduces a marked node and a cycle that contains the node. Moreover, further traversal, we maintain, will not disturb the cycle. Relating the cycle to the marked node we can define marked(t) as

'node t is right-reachable from node t.l',

where 'right-reachable' means that the path is via right links only. Hence we see that transformations t → t″ (instead of t → t′) can be used to effect an inorder traversal, as long as 'marked' nodes are treated as having **nil** left subtrees – and this can be tested using predicate marked(t).

To complete the analysis we consider how to restore the original tree. The simple answer is to remove the added edges – by 'unmarking' the marked nodes! Subsequent to the transformation that marked it, a node will be visited and, being marked, it will be processed and unmarked, after which the algorithm will proceed with the traversal of the right subtree of the node.

Let us gather together the arguments into an informal invariant. The invariant is established by t := T, where T is the root of the tree to be traversed. The invariant is (i) and (ii) and (iii) and (iv) and (v) as follows:

(i) A node m is 'marked' if it is on a cycle such that m is the first and last node on the cycle, m.l is the second, and the path from m.l to m is via right links. The penultimate node k on each such cycle (i.e., k.r = m) has a **nil** right pointer field in the original tree. Otherwise, all links contain their original values.

(ii) All cycles are edge-disjoint; no edge is part of two different cycles. This follows from the fact that in each cycle m, m.l, ..., k, m all links except k.r (= m) contain their initial values.

(iii) All marked nodes are right-reachable from t.

(iv) Node t is a node of the original tree, and is the root of a tree whose marked nodes are considered to have empty left subtrees.

(v) An inorder traversal of tree t completes the inorder traversal of the original tree T.

The algorithm is:

```
t := T;
do t ≠ nil →
  if  t.l = nil → process(t); t := t.r
  □ t.l ≠ nil →
       p := t.l;
       do p.r ≠ nil and p.r ≠ t → p := p.r od
       {p = 'rightmost' node of the original subtree t.l};
        if p.r  = nil → {not marked(t)}
              p.r := t {marked(t)}; t := t.l
        □ p.r = t → {marked(t)}
              process(t); p.r := nil {not marked(t)};
              t := t.r
        fi
  fi
od.
```

We now explain why the algorithm is linear in the number n of nodes of the tree. First, we have already given the variant, monotonically decreasing function for the main loop, which shows that the loop body can be executed no more than

n + (the number of left edges) $<$ 2n

times. The problem is to show that, in total, execution of the inner loop **do** p.r ≠ **nil and** p.r ≠ t → p := p.r **od** is proportional to n. Each execution of the loop traverses the edge of a path (x1, x2, x3, ..., x4) of the *original* tree T as in Fig. 1(a). Each such path is traversed exactly twice by an execution of the loop – once to build a cycle (Fig. 1(c)) and once to destroy it. Since each edge can be in at most *one* such path, the total time spent in executing the loop is no worse than proportional to 2 ∗ (No. of edges of T), which is $\leqslant 2n$.

## 3. Variations on the theme

Preorder traversal is: process the root; traverse the left subtree in preorder; traverse the right subtree in preorder. It differs but little from inorder traversal: a node is processed before proceeding to the left subtree, instead of before proceeding to the right subtree as with inorder traversal. So the above algorithm is readily converted to a preorder traversal algorithm by processing an unmarked rather than a marked node.

The third principal traversal order is postorder: traverse the left subtree in postorder; traverse the right subtree in postorder; process the root. In general, postorder traversal is more difficult because each node may have to be visited up to three times. The above algorithm provides the basis for a postorder traversal, but it is tricky. We leave it to the reader, with the following hints. Introduce a header node whose left subtree is the tree to be traversed and whose right subtree is **nil**. Initialise t with the name of the header node. The traversal proceeds as for inorder traversal except that the only processing of nodes is: if t is marked, then process nodes t.l, t.l.r, t.l.r.r, ... of the original tree in reverse order.

## Acknowledgment

I am grateful to W.H.J. Feijen whose unfailing encouragement saw me through several attempts to describe this algorithm, and to E.W. Dijkstra who by his patient and inspiring criticism of a previous attempt made me try harder. David Gries, in a criticism that went way beyond the call of duty, contributed enormously to the final draft. I also enjoyed the criticism of A.J. Martin, R.W. Bulterman, A.J.M. van Gasteren, Heikki Saikkonen and J.G. Byrne.

## References

[1] D.E. Knuth, The Art of Computer Programming, Vol. 1: Fundamental Algorithms (Addison-Wesley, Reading, MA, 2nd ed., 1973).

[2] W.M. Waite, Comput. J. 20 (1) (1977) 92.