Appl. Math. Lett. Vol. 2, No. 1, pp. 79–82, 1989
Printed in Great Britain

0893-9659/89 $3.00 + 0.00

Efficient Reconstruction of Binary Trees from Their Traversals

R. D. CAMERON, B. K. BHATTACHARYA and E. A. T. MERKS

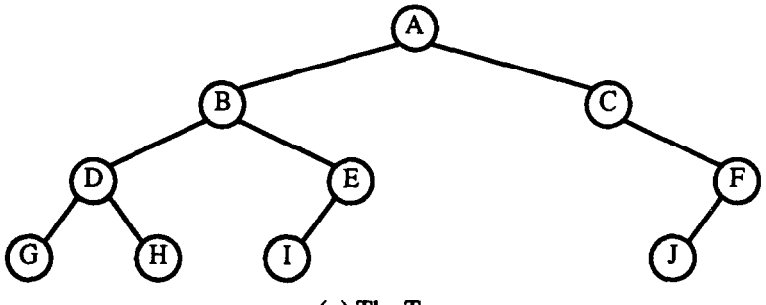
School of Computing Science
Simon Fraser University

Abstract. Given the n nodes of a binary tree in both inorder and preorder sequence, the tree can be uniquely identified. An efficient algorithm for reconstructing such trees from their sequences is presented, using O (n) time and O (h) intermediate storage, where h is the height of the tree being reconstructed.

1. INTRODUCTION

Given the nodes of an n-node binary tree in both inorder and preorder, the original tree may be reconstructed in a recursive fashion as described by Knuth [4] or using recently reported algorithms. Burgdorff et al. [1] employ *inorder-preorder sequences* [3] to achieve an iterative O (n^2) algorithm which they claim is an improvement over the classical O (n^2) recursive algorithm. Chen et al. [2] report two algorithms also using inorder-preorder sequences; one uses a sort resulting in an algorithm requiring O ($n \log n$) time and O (n) space while the second uses a hashing technique yielding O (n) time in the expected case, but requiring a large amount of intermediate storage for both the inorder-preorder sequences and the hash table. In this paper, we present an efficient algorithm which directly rebuilds the tree without intermediate inorder-preorder sequences. The algorithm employs a straightforward recursive approach requiring O (n) time and O (h) intermediate storage where h is the height of the tree which is being reconstructed.

Figure 1 illustrates the problem.



(a) The Tree

G	D	H	B	I	E	A	C	J	F

(b) The Inorder Sequence

A	B	D	G	H	E	I	C	F	J

(c) The Preorder Sequence

Figure 1: An Example Tree and Its Sequences

For the binary tree shown in Figure 1a, an inorder traversal generates the node sequence shown in Figure 1b. Similarly, a preorder traversal generates the node sequence of Figure 1c. In these figures, nodes are represented using distinct alphabetic labels; any other unique labelling scheme will do. The problem, then, is to reconstruct the original tree given the inorder and preorder sequences of node labels.

2. DUAL-ORDER TRAVERSAL

Our algorithm employs a combined preorder and inorder traversal. When a tree node is to be traversed, it is visited both before its left subtree is traversed (preorder) and then again after the left subtree is traversed and before the right subtree is traversed (inorder). Such an algorithm is said to traverse trees in *dual order* [5]. Note that all dual-order traversal algorithms will visit nodes for preorder and inorder processing in exactly the same order.

In fact, the original generation of the inorder and preorder sequences can be achieved using a dual-order traversal algorithm as shown in the Modula procedure of Figure 2.

```
PROCEDURE GenerateSequences (x : Tree);
BEGIN
  EmitLabel(Pre, LabelOf(x));
  IF LeftTree(x) ≠ NIL THEN GenerateSequences(LeftTree(x)) END;
  EmitLabel(In, LabelOf(x));
  IF RightTree(x) ≠ NIL THEN GenerateSequences(RightTree(x)) END
END GenerateSequences;
```

Figure 2: Sequence Generation Using Preorder-Inorder Traversal

In this figure, trees are represented as values of some data type `Tree`; the special value `NIL` denotes the empty tree. The function `LabelOf` determines the unique label of a tree node, while the functions `LeftTree` and `RightTree` return its left and right subtrees, respectively. The variables `Pre` and `In` represent the preorder and inorder sequences being generated, respectively; the procedure `EmitLabel` appends a node label to one of the sequences. Note that this algorithm is a straightforward combination of the separate preorder and inorder traversals for generating the sequences.

Tree reconstruction using dual-order traversal can be achieved by making use of two properties about the order in which tree nodes are visited for preorder and inorder processing. When the dual-order traversal algorithm is reversed from a tree-traversal to a tree-building algorithm, these properties provide the information necessary to decide when the left and/or right subtrees of a given tree node are empty. Knowing this, the tree reconstruction is straightforward. These properties are presented as the following lemmas.

LEMMA 1. After a node is visited for preorder processing during a dual-order traversal, it is also the next node visited for inorder processing if and only if the left subtree of that node is empty.

PROOF: The proof proceeds by demonstrating that this property holds for the dual-order algorithm of Figure 2; this establishes it for all dual-order algorithms. After a label is emitted to the `Pre` sequence, it will be emitted to the `In` sequence immediately unless the intervening conditional recursive call is made. If the left subtree is empty, the recursive call is not made and so the label for this node is the next one emitted to the `In` sequence. If the left subtree is non-empty, a recursive call to `GenerateSequences` is made. This call will see the label for at least one other node emitted to the `In` sequence before the node of interest can be emitted. □

DEFINITION. In a dual-order traversal, a node is said to be *pending* if it has been visited for preorder processing, but has not yet been visited for inorder processing. If at least one node is pending, the one that has most recently been visited for preorder processing is said to be the *currently pending* node.

LEMMA 2. If a node is pending during the dual-order traversal of a node x, then after x

has been visited for inorder processing, the next node to be visited for inorder processing will be the currently pending node if and only if the right subtree of node x is empty.

PROOF: Consider Figure 2. Assume that a node is pending. When a node is visited for inorder processing, it remains to traverse the right subtree for that node.

1. If the right subtree is empty, then the traversal of this node is complete and control returns to the point at which the traversal of this node was recursively invoked. This may be either a recursive call for traversing a left subtree or one for traversing a right subtree. As long as a return is made to a recursive call for a right subtree, the node being returned to is not pending (i.e., its label has been emitted to both the `Pre` and `In` sequences) and control will again be immediately returned. Since a node is pending, control must eventually return to the point after a recursive call for a left subtree. The first such return from a left recursion will be for the currently pending node. The next action at this point is to emit the label of the pending node to the `In` sequence.

2. If the right subtree of the original node is non-empty, then a recursive call is made to traverse that subtree. This will see the label for at least one node emitted to the `In` sequence. Thus, if the right subtree is non-empty, the next label to be emitted to the `In` sequence will be that for some subnode in the right subtree, which cannot be the currently pending node. □

3. THE TREE RECONSTRUCTION ALGORITHM

Using the properties established by Lemmas 1 and 2, the tree reconstruction algorithm can be formulated as shown in Figure 3.

```
PROCEDURE RebuildTree (VAR In, Pre : LabelSequenceType) : Tree;

  PROCEDURE RebuildWithPendingNode (pending : LabelType) : Tree;

    VAR TheLabel : LabelType; LeftTree, RightTree : Tree;
  BEGIN
    TheLabel := First(Pre);
    Advance(Pre);
    IF TheLabel = First(In) THEN LeftTree := NIL
     ELSE LeftTree := RebuildWithPendingNode(TheLabel)
     END;
    Advance(In);
    IF First(In) = pending THEN RightTree := NIL
     ELSE RightTree := RebuildWithPendingNode(pending)
     END;
    RETURN BuildTreeNode(TheLabel, LeftTree, RightTree)
  END RebuildWithPendingNode;

  VAR TheLabel : LabelType; LeftTree, RightTree : Tree;
BEGIN
  TheLabel := First(Pre);
  Advance(Pre);
  IF TheLabel = First(In) THEN LeftTree := NIL
   ELSE LeftTree := RebuildWithPendingNode(TheLabel)
   END;
  Advance(In);
  IF EmptyQ(In) THEN RightTree := NIL
   ELSE RightTree := RebuildTree(In, Pre)
   END;
  RETURN BuildTreeNode(TheLabel, LeftTree, RightTree)
END RebuildTree;
```

Figure 3: The Tree Reconstruction Algorithm

In this figure, the `In` and `Pre` sequences are each processed in sequential order. The `First`

function returns the first node in a sequence, while the `Advance` procedure removes the first element so that the previously second element (if there is one) now becomes the first element. The `EmptyQ` function is used to determine whether all the elements of a sequence have been removed or not. The node labels stored in these sequences are assumed to be of some data type `LabelType`. The only new tree processing function introduced is `BuildTreeNode` which constructs a tree node given its label and its left and right subtrees.

The main procedure in Figure 3 is used to start the reconstruction process and also to continue the process when there is no currently pending node. Whenever there is a currently pending node, the auxiliary procedure is used. Note that the auxiliary procedure accepts as a parameter the label of the currently pending node, this will be `TheLabel` of the current node when a left subtree is being constructed, otherwise it will be the `pending` label previously passed in. Note how the properties established in Lemmas 1 and 2 above are used in determining whether the left and/or right subtrees of a node should be empty. It can be established that this algorithm is a proper dual-order traversal by straightforward structural induction on binary trees.

It is easy to see that this algorithm requires O (n) time and O (h) intermediate storage. Each of the basic operations (equality (=) on labels, `First`, `Advance`, and `EmptyQ` on label sequences, and `BuildTreeNode` on trees) can easily be implemented as constant time operations on appropriate data structures. It thus costs only a constant amount of time to process a node (i.e., to visit it for preorder and inorder processing and to construct it given its subtrees). Since each tree node is processed exactly once, the total time required is O (n). The only intermediate storage required by the algorithm is the stack space required for the local variables of each recursive activation. Since the maximum number of activations that can be present at any one time is h, the height of the tree, the intermediate storage requirement is O (h).

REFERENCES

[1] H. A. Burgdorff, S. Jajodia, F. N. Springsteel and Y. Zalcstein, *Alternative methods for the reconstruction of trees from their traversals*, BIT **27** (2), 1987, 134-140.

[2] G.-H. Chen, M. S. Yu and L. T. Liu, *Two algorithms for constructing a binary tree from its traversals*, Information Processing Letters **28** (6), 1988, 297-299.

[3] T. Hikita, *Listing and counting subtrees of equal size of a binary tree*, Information Processing Letters **17** (2), 1983, 225-229.

[4] D. E. Knuth, "The Art of Computer Programming, Volume 1 - Fundamental Algorithms", Addison-Wesley, Reading, Mass., 1973, pages 329 and 560.

[5] E. M. Reingold and W. J. Hansen, "Data Structures in Pascal", Little, Brown and Company, Boston, 1986, page 229.