CONSTRUCTING A BINARY TREE EFFICIENTLY FROM ITS TRAVERSALS

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF TAMPERE

REPORT A-1998-5

UNIVERSITY OF TAMPERE
DEPARTMENT OF COMPUTER SCIENCE
SERIES OF PUBLICATIONS A
A-1998-5, APRIL 1998

CONSTRUCTING A BINARY TREE EFFICIENTLY FROM ITS TRAVERSALS

ERKKI MÄKINEN

University of Tampere
Department of Computer Science
P.O.Box 607
FIN-33101 Tampere, Finland

ISBN 951-44-4351-9
ISSN 0783-6910

Constructing a binary tree efficiently from its traversals

Erkki Mäkinen

Department of Computer Science, University of Tampere, P.O. Box 607, FIN-33101 Tampere, Finland

Abstract

In this note we streamline an earlier algorithm for constructing a binary tree from its inorder and preorder traversals. The new algorithm is conceptually simpler than the earlier algorithms and its time complexity has a smaller constant factor.

Keywords: Algorithms; Binary trees; Tree traversals.

1 Introduction

Given the preorder and inorder traversals (or postorder and inorder traversals) of the nodes of a binary tree, the binary tree structure can be constructed. In the matter of fact, the construction is possible in linear time [1,4]. Recently, Xiang and Ushijima [5] have determined the constant factors of these algorithms by counting the numbers of comparision operations needed. They found out that the linear coefficient of Mäkinen's algorithm [4] is 3 in the best case and 5 in the worst case, and that the corresponding coefficients in the algorithm of Andersson and Carlsson [1] are 4 and 7. Moreover, Xiang and Ushijima presented a new algorithm with linear coefficient 3 both in the best and in the worst case.

In this note we streamline our earlier algorithm [4], and show that the resulting algorithm can be implemented to run with linear coefficient 2 in the best case and 3 in the worst case.

We use the standard tree terminology [3]. In order to simplify our presentation, we first recall a binary tree coding system introduced by Johnsen [2]. A unique coding system for binary trees can be established by inserting the nodes into the tree in preorder and recording the positions to which the insertions are made. As an example, consider the tree in Figure 1. There are three possibilities to insert the next node in preorder. These are numbered from left to right by 0, 1, and 2. Johnsen [2] uses the phrase *trailing leaf* for

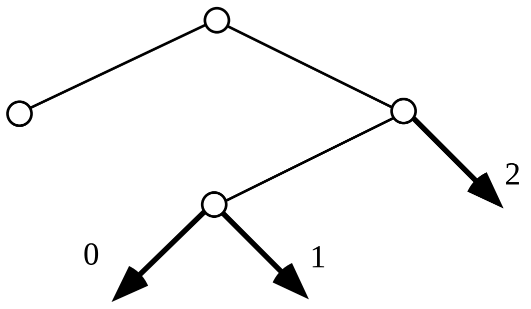

Fig. 1. A sample binary tree with three trailing leaves.

the positions where the next insertion can take place. So far, the insertions in the sample tree are done into the positions 0, 2, and 0. The next insertion will augment the code word to be (0,2,0,0), (0,2,0,1), or (0,2,0,2) depending on our choice for the place of the next insertion. Notice that a binary tree with n nodes has a code word with $n-1$ code items.

Johnsen has proved that for each item x_i in a code word $(x_1, x_2, \ldots, x_{n-1})$ we have $0 \leq x_i \leq i - (x_1 + x_2 + \ldots + x_{i-1})$. This implies the following fact.

Proposition 1 *If* $(x_1, x_2, \ldots, x_{n-1})$ *is a code word of a binary tree with* n *nodes in Johnsen's coding scheme, then* $\sum_{i=1}^{n-1} x_i \leq n-1$.

2 The algorithm

We suppose that trees are labeled with symbols from an ordered alphabet. Moreover, we suppose that inorder sequences always obey this order, i.e., if $I_1 I_2 \ldots I_n$ is an inorder sequence, then we have $I_1 < I_2 < \ldots < I_n$ according to the order defined in the alphabet. These assumptions make it possible to compare labels when the tree is constructed by studying the preorder sequence. Notice that these assumptions can be done without loss of generality. Namely, if the inorder sequences do not obey the order in the alphabet, we can store the inorder positions in a table and compare the table entries. Instead of using such a table we simply directly compare the labels.

As can be expected, our algorithm needs a stack. It is advantageous to use a bottom marker in the stack. We define the bottom marker X to be the last element in the order relation of the label alphabet used.

As an example, consider the tree shown in Figure 2. Its nodes are labeled in alphabetical order according to inorder. The corresponding preorder sequence is $DACBFEG$. The algorithm begins with creating the root labeled with D. The lable D is also pushed to the stack. The second label in the preorder sequence is A. Since $A < D$, where D is the top element of the stack, we

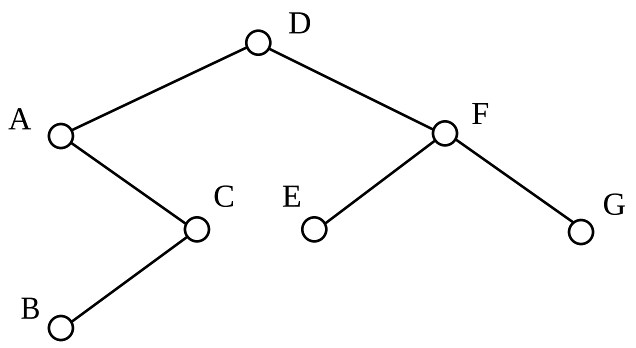

Fig. 2. A sample binary tree with nodes alphabetically labeled in inorder.

know that the left child of the root should be labeled with A. Moreover, A is pushed to the stack. Next, we notice that $A < C$ but $C < D$. Now, A is popped from the stack, and C is pushed to its place. C is the right child of A. Using the terminology of Johnsen's coding scheme, popping an element from the stack means that the leftmost trailing leaf does not actually exist in the tree. Or equivalently, the corresponding code item is greater that 0. Next, we compare B and the top element C. Since $B < C$, we insert B as the left child of C. Now the stack contains $XDCB$ (, where the top of the stack is on the right). The next label in the preorder sequence is F. We have $B < F$, $C < F$, $D < F$, and $F < X$. It follows, that B, C, and D are popped from the stack, and F is pushed to the stack. We now know that the left subtree of the root is completed. Notice that each label comparision was made for checking to which trailing leaf the next node can be inserted. The whole construction is completed by noticing that E is the label of the left child of F, and that G is the lable of the corresponding right child.

Notice that after pushing an element to the stack we create a left child, and after popping an element from the stack a right child is created. More importantly, notice the interesting fact that the number of label comparisions needed to find the correct place for a node, and the corresponding code item in Johnsen's coding scheme are in a very close connection with each others. The code word for the tree in Figure 2 is (0,1,0,3,0,2). It is easy to check that the number of label comparisions done for placing a node is always one greater than the corresponding code item. For example, when the code item is 0, the only comparision is done between the next label in the preorder sequence and the top of the stack. Before formulating this as a theorem, we give the algorithm itself in a more formal notation.

Algorithm 1 (Binary Tree Construction)
{ *the preorder sequence is indexed as* $preorder[i]$*, where* $1 \leq i \leq n$*;*
the phrase 'current node' refers to the node labeled with the top element of the stack }

```
begin
 initialize the stack with the bottom marker $X$;
 create the root and label it with $preorder[1]$;
 $index := 2$;
while ($index \leq n$) do
 if $preorder[index] < top(stack)$
   then begin
     create the left child of the current node and label it with preorder[index];
     push $preorder[index]$ to the stack;
     index:= index + 1
   end
   else begin
     pop top(stack);
     while $preorder[index] < top(stack)$ do pop top(stack);
     create the right child of the previous current node
       and label it with $preorder[index]$;
     push $preorder[index]$ to the stack;
     index:= index + 1
   end
end {Algorithm}
```

Consider now again the number of label comparisions done by the algorithm. The best case occurs when the tree to be constructed is a left list, i.e. a tree containing the root and left children only. The corresponding code word is $(0, 0, \ldots, 0)$, and we need only $n - 1$ label comparisions. On the other hand, the worst occurs when the sum of the code items in the code word is $n - 1$. Then we need $2n - 2$ label comparisions.

In order to make the number of comparisions comparable with those given by Xiang and Ushijima [5], we also count comparisions other than label comparisions made by the algorithm. The algorithm has a while-loop condition ($index \leq n$), which causes $n - 1$ comparisions when constructing a tree with n nodes. Hence, by Proposition 1 and the discussion above, we have

Theorem 2 *When constructing a binary tree with* n *nodes, Algorithm 1 makes* $n - 1$ *label comparisions in the best case and* $2n - 2$ *label comparisions in the worst case. The total number of comparisions is* $2n - 2$ *in the best case and* $3n - 3$ *in the worst case.*

3 Conclusions

Our algorithm requires fewer comparisions than the previously known algorithms for constructing a binary tree from its traversals. Moreover, our algorithm is conceptually simple imitating a known binary tree coding scheme.

References

[1] Arne Andersson and Svante Carlsson, Construction of a tree from its traversals in optimal time and space. *Inform. Process. Lett.* **34** (1990), 21–25.

[2] Ben Johnsen, Generating binary trees with uniform probability. *BIT* **31** (1991), 15–31.

[3] Donald E. Knuth, *The Art of Computer Programming. Vol. 1, Fundamental Algorithms. Third Edition.* Addison-Wesley, 1997.

[4] Erkki Mäkinen, Constructing a binary tree from its traversals. *BIT* **29** (1989), 572–575.

[5] Limin Xiang and Kazuo Ushijima, On constructing a bianry tree from its traversals. Manuscript, 1998.