University of Wollongong
Research Online

Department of Computing Science Working Paper Series

Faculty of Engineering and Information Sciences

1982

Stack permutations and an order relation for binary trees

Reinhold Friedrich Hille
University of Wollongong

Recommended Citation

Hille, Reinhold Friedrich, Stack permutations and an order relation for binary trees, Department of Computing Science, University of Wollongong, Working Paper 82-8, 1982, 11p.
http://ro.uow.edu.au/compsciwp/25

Research Online is the open access institutional repository for the University of Wollongong. For further information contact the UOW Library: research-pubs@uow.edu.au

STACK PERMUTATIONS AND AN ORDER RELATION FOR BINARY TREES

R. Fritz Hille

Department of Computing Science
University of Wollongong

Preprint No. 82-8 Reprinted 9.4.84

P.O. Box 1144, WOLLONGONG N.S.W. 2500, AUSTRALIA
tel (042)-270-859
telex AA29022

STACK PERMUTATIONS AND AN ORDER RELATION FOR BINARY TREES.

by
R.F. Hille

Department of Computing Science
The University of Wollongong
P.O. Box 1144
Wollongong 2500
Australia

ABSTRACT

An isomorphism between stack permutations of a set of n elements and ordered binary trees with n vertices is presented, which allows the construction of simple linear time algorithms to compute a ranking function and its inverse for binary trees. No pre-processing of tables is required, as was the case with previously published methods.

Keywords and phrases: binary trees, encoding, lexicographic order, permutations, stack.

CR categories: 5.31, 5.32, 5.39.

STACK PERMUTATIONS AND AN ORDER RELATION FOR BINARY TREES.

by
R.F. Hille

Department of Computing Science
The University of Wollongong
P.O. Box 1144
Wollongong 2500
Australia

1. INTRODUCTION

Binary trees play an important role in computing, not only as data structures but also as analytical devices. We focus our attention on ordered binary trees, because the distinction between left and right subtree is naturally implied in the usual computer representations.

By a *stack permutation* of a set we mean a permutation that can be generated by passing the set through a stack once. They are the inverses of Knott's tree permutations [1].

He describes a numbering system for binary trees based on the following definition of an order relation:

Given two binary trees T1 and T2, we say T1 < T2 iff:

1. c(T1) < c(T2), or
2. c(T1) = c(T2) and l(T1) < l(T2), or
3. c(T1) = c(T2) and l(T1) = l(T2) and r(T1) < r(T2),

where c(T) is the tree's cardinality, i.e. the number of vertices, l(T) and r(T) are the left and right subtrees, respectively.

Knott uses his *tree permutations* to define a ranking function, which is a mapping from these permutations into the set of integers. His tree permutations are in fact the pre-order traversal sequences of binary trees, which are labeled such that their in-

order traversal sequences give the integers in ascending order. He defines them in the following way: a tree permutation $p = p_1\ p'\ p''$ has the property that p' and p'' are again tree permutations. p_1 is just the leftmost element of p. His ranking function is a 1-1 mapping from the set of these permutations, and hence from the set of ordered binary trees, onto the set $\{1,2,3,\ldots,B_n\}$ of integers, where B_n is the number of ordered binary trees with cardinality n [2, sec. 2.3.4.4]. Knott's algorithm for computing the ranking function, *rank*(T), runs in linear time and gives the relative addresses of the element (rank(l(T)), rank(r(T))) in a two dimensional array of size

$$[1 : B_{c(l(T))},\ 1 : B_{c(r(T))}],$$

where $B_{c(l(T))}$ stands for the number of trees with the cardinality of the left subtree and $B_{c(r(T))}$ for the number of trees with the cardinality of the right subtree. Hence, his algorithm requires pre-processing of a large table whose size grows exponentially with the cardinality of the trees.

His method for computing $rank^{-1}$ involves the generation of tree permutations from which the corresponding binary trees can easily be constructed. This algorithm also requires pre-processing of the table.

Rotem and Varol [3] use *ballot sequences* instead of tree permutations to generate all trees of given cardinality. These ballot sequences are the inversion tables of the tree permutations. The authors concede that their method does not produce the trees in lexicographical order, but claim that their algorithms are more efficient than those reported by Knott.

Solomon and Finkel [4] present an algorithm that transforms a tree into its successor by operating directly on the tree, and a modification of Knott's inverse ranking function. Their algorithms still require the preparation of a table of *Catalan Numbers*

$$B_n \doteq \frac{1}{1+n} \quad {}^{2n}C_n.$$

Their algorithm *Next*(T) requires calls to two routines that run in O(n). Their $Rank^{-1}$(T) runs in O(nlogn), whereas their *Rank*(T) requires O(n) operations. These algorithms still require the pre-processing of tables.

Proskurowski [5] constructs all ordered binary trees with n vertices by generating all *extended binary trees* with n+1 leaves. This is achieved by expanding certain leaves of the extensions of ordered binary trees with n-1 vertices. His order is derived from a way of deciding which leaf to expand next. It appears that his algorithm *expand* will always commence by expanding the smallest tree first and building the list from the beginning. His extended binary trees are characterised by sequences of binary digits. He generates the master list of trees in reverse order.

Rotem and Varol [3] claim: "... there is no simple way of deriving stack sortable permutations in their order corresponding to the natural order of trees."

We present a bijection mapping between stack permutations and ordered binary trees and derive from it a simple way to generate binary trees *in order* corresponding to the lexicographical order of the related stack permutations. Our algorithms *ENCODE* and *DECODE* transform a tree into a binary sequence or a sequence into a tree, respectively. The algorithm *NEXT* constructs the successor of a given sequence. These algorithms require no pre-processing of tables and their time complexity is linear in the number of tree nodes.

2. Stack Permutations and Binary Trees.

A stack permutation of the set S = {1,2,3,...,n} is a permutation that can be generated by passing this set through a stack once. The necessary string of additions and deletions represents a *legal* sequence of stack operations, meaning, that at any time the number of deletions does not excede the number of additions. We represent the stack

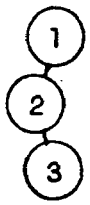
operations *add* and *delete* by the binary disgits 1 and 0, respectively. Consequently, we have an isomorphism between legal sequences of stack operations and our stack permutations. We can now construct a simple isomorphism between stack permutations and binary trees in the following way:

The digit 1 means to add the left child to the current node, whereas the subsequence 01 means to add the right child. A subsequence of k digits 0 preceding the next digit 1 means to ascend k-1 left links (return k-1 levels of recursion) before adding the right child to the node just reached.

For example, the sequence

1 1 0 1 0 0 0

means that we generate the permutation (2,3,1) from the set (1,2,3) and it represents the binary tree



The sequences have 2n+1 digits because every vertex forces a "1" and every open link forces a "0". A binary tree with n vertices has n+1 open links.

The trees are always labeled in such a way that their pre-order traversal sequence gives the integers in ascending order. The stack permutation generated is then given by the in-order traversal sequence of the tree.

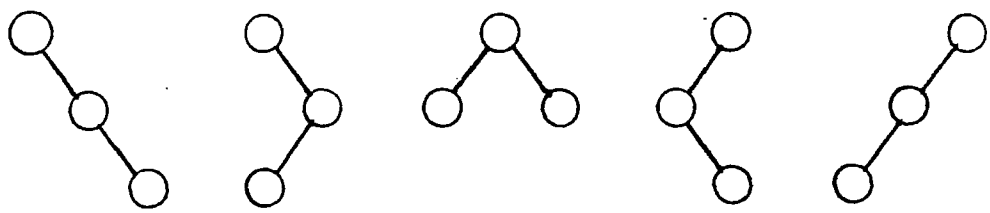
The proof that this is an isomorphism can easily be constructed directly by showing that there exists exactly one binary tree for every stack permutation, and that there exists exactly one stack permutation for each binary tree.

Given a legal sequence of stack operations, one can always construct at least one corresponding ordered binary tree according to the mapping. It is easily seen that

there cannot be another different binary tree corresponding to the sequence.

Conversely, given an ordered binary tree, one can traverse it in pre-order and build the sequence by adding a "1" every time a vertex is visited. When an empty pointer field is encountered, a "0" is added to the sequence. Hence, it is possible to construct at least one sequence for each binary tree. The fact that this sequene is unique follows from the uniqueness of the traversal order.

The order of the trees, and therefore also the order of the stack permutations, is the same as the lexicographic order of the binary sequences. Thus, the five binary trees with cardinality 3 are, in order:



and the corresponding binary sequences together with the stack permutations in the same ascending order are:

1010100	1 2 3
1011000	1 3 2
1100100	2 1 3
1101000	2 3 1
1110000	3 2 1

The last two digits are always "0" and may be omitted. We show them here for completeness.

It seems at first sight that the order of the trees produced here is the same as the order defined by Knott. However, since the stack permutations are the inverses of the tree pemutations, the order of the trees will differ in general from that given by Knott. This does not matter very much because the order is used here primarily for the purpose of avoiding isomorphism checking (see Read, R.C. [7]) when generating a list of trees.

If one wants to generate a random binary tree one may either use the algorithm of Arnold and Sleep [8] for the uniform random generation of balanced paranthesis strings or construct a mapping, and therefore an algorithm, from the integers into the set of bit strings defined above. This allows the use of a random number generator.

The successor of a given binary sequence can be constructed by locating the right-most digit "1" that has a left neighbour "0" and transposing the two. The tail of the sequence (to the right of the current position) must then be reversed to the firtst one of it's size, that is, all digits "1" must be put back at their original places. This can already be done during the search. The current sequence is the last one if the "1" located is the leftmost digit of the sequence.

3. The Algorithms

Given a binary tree with n vertices, and therefore a binary sequence with 2n+1 digits, we construct the successor in the following way:

```
procedure NEXT
begin
        begin at the right end of the sequence
        finished:=false
        while there are digits to the left and not finished do
        begin
                move left to the nearest "1"
                if the left neighbour is "0" then
                begin
                        exchange
                        finished:=true
                end
                else
                        move the "1" to its original place
        end
end NEXT
```

In the worst case the while loop runs n times, namely when the given sequence is already the last one. A small amount of book keeping is required, namely, how many

digits "1" are already in their original places. This determines the original place of the current "1" if it must be moved.

Our procedure ENCODE is the equivalent of Knott's ranking function. It produces a binary sequence which is interpreted as an integer for the purpose of determining its successor or comparing it with the code sequence of another binary tree. The algorithm ENCODE is essentially a pre-order traversal of the tree, adding the digits 1 or 0 to the sequence at the right moment.

```
procedure ENCODE(rootptr)
begin
        if not rootptr=nil  then
        begin
                add "1" to the sequence
                ENCODE(rootptr.left)
                ENCODE(rootptr.right)
        end
        else
                add "0" to the sequence
end ENCODE
```

The inverse of this algorithm simply scans the sequence from left to right and builds the tree according to the mapping described in section 2. This corresponds to a pre-order traversal of the tree and hence also runs in linear time because every vertex is visited exactly once.

```
procedure DECODE(rootptr)
begin
        rootptr:=get_node
        get_digit
        if digit=1 then
                DECODE(rootptr.left)
        if digit=0 and next_digit=1 then
                DECODE(rootptr.right)
        get_digit
end DECODE
```

4. A Classification of Tree Permutations

In his paper [1] Knott mentiones four classes of tree permutations, which he characterises as *hLR*, *hRL*, *LRh*, and *RLh*. Obviously, these correspond to the pre- and post-order traversal sequences of the trees associated with the permutations. He then states that the classes *LhR* and *RhL* are degenerate because they contain only one permutation.

The reason for this "degeneracy" is very simple. Knott's tree permutations are the pre-order traversal sequences of trees that are labeled in such a way that their in-order sequences are just the integers in ascending order. The class *LhR* of permutations is made up of the in-order traversal sequences of trees whose in-rder traversal sequence is (1,2,3,...,N). Hence the degeneracy.

Instead of the four classes of tree permutations mentioned by Knott we can define 24 classes. The trees can be labeled in 6 different ways, namely two for each of the three traversal sequences. In each case they can be traversed in four different ways, so that we obtain 24 classes of tree permutations. There exist various close relationships between these classes.

5. Conclusion

We have presented an algorithm for computing the value of a ranking function of a binary tree, the inverse, and an algorithm to compute the successor of a binary tree. The time complexity of these algorithms is O(n), where n is the number of vertices. No pre-processing of tables is required. Previously published algorithms for the same purpose required the pre-processing of tables, and computing the inverse of the ranking function required time O(nlogn).

REFERENCES

[1] Knott, G.D. "A numbering System for binary Trees", Comm ACM **20** (1977)113-115.

[2] Knuth, D.E. "The Art of Computer Programming", vols. 1 and 3, second edition, Addison-Wesley, 1973.

[3] Rotem, D. and Varol, Y.L. "Generation of Binary Trees from Ballot Sequences", J. ACM **25** (1978)396-404.

[4] Solomon, M. and Finkel, R.A. "A Note on Enumerating Binary Trees" J. ACM **27** (1980)3-5.

[5] Proskurowski, A. "On the Generation of Binary Trees", J. ACM **27** (1980)1-2.

[6] Rotem, D. "On a Correspondence between Binary Trees and a certain Type of Permutation", Info. Proc. Lett. **4** (1975)58-61.

[7] Read, R.C. "Every one a Winner" in *Algorithmic Aspects of Combinatorics*, Alpach, Hell, and Miller, Eds., North-Holland, Amsterdam, 1978, pp. 107-120.

[8] Arnold, D.P. and Sleep, M.R. "Uniform Random Generation of Balanced Paranthesis Strings" ACM Transactions on Programming Languages and Systems **2** (1980)122-8.