MRC Technical Summary Report #1926

A NOTE ON ENUMERATING BINARY TREES

Marvin Solomon and Raphael A. Finkel

Mathematics Research Center
University of Wisconsin—Madison
610 Walnut Street
Madison, Wisconsin 53706

February 1979

Received February 5, 1979

Approved for public release
Distribution unlimited

Sponsored by
U. S. Army Research Office
P.O. Box 12211
Research Triangle Park
North Carolina 27709
A NOTE ON ENUMERATING BINARY TREES

Gary Knott has presented algorithms for computing a bijection between the set of binary trees on n nodes and an initial segment of the positive integers. Rotem and Varol presented a more complicated algorithm that computes a different bijection, claiming that their algorithm is more efficient and has advantages if a sequence of several consecutive trees is required. We present a modification of Knott's algorithm that is simpler than Knott's and as efficient as Rotem and Varol's. We also give a new linear-time algorithm for transforming a tree into its successor in the natural ordering of binary trees.

Keywords and phrases: binary trees, permutations, generators, enumeration, combinatorics, stack-sortable permutations

AMS(MOS) Subject Classification: 05C30, 68A10, 05A10, 68A20

Work Unit Numbers 4 and 8: Probability, Statistics & Combinatorics and Computer Science

Sponsored by the United States Army under Contract No. DAAG29-75-C-0024. Partial support by the Research Committee of the Graduate School of the University of Wisconsin-Madison.
Significance and Explanation

We illustrate the mathematical problem discussed in this paper in terms of one possible application, the storage and retrieval of information in digital computers by techniques that depend on a series of yes/no questions. The binary tree is a structure that makes it easy to put information into, and take information out of, a data base, using such a sequence of questions.

Given a certain amount of information, there will be many different ways of representing this in the form of binary trees. The procedure described in this report gives all possible ways of storing this information in binary trees. Given one representation, it is possible to find the next one in line. Any representation in the list can be generated without going through all of them.

The responsibility for the wording and views expressed in this descriptive summary lies with MRC, and not with the authors of this report.
A NOTE ON ENUMERATING BINARY TREES

Marvin Solomon and Raphael A. Finkel

Introduction

Gary Knott has published algorithms for computing a bijection \( \text{Rank} \) from the set of binary trees with \( n \) nodes to an initial segment of the integers and for computing \( \text{Rank}^{-1} \) [1]. His method for computing \( \text{Rank}^{-1} \) involves generating certain permutations of \( [1, 2, \ldots, n] \) called \textit{tree permutations}, which are in one to one correspondence with the binary trees and from which the binary trees may be easily constructed. Rotem and Varol propose an alternative technique for generating the trees [2]. Instead of constructing the tree permutations (which they call \textit{stack-sortable permutations}) directly, they construct the \textit{inversion tables} of these permutations, which are sequences of non-negative integers called \textit{ballot sequences}. They show that the ballot sequences may be generated in lexicographic order and present a clever and efficient algorithm for converting the ballot sequences into trees. In this note, we present modifications of Knott's algorithms that map directly between trees and integers. In addition, we correct some misconceptions put forth by Rotem and Varol, and present a new algorithm that transforms a given tree to the next one in sequence.

In comparing their technique to Knott's, Rotem and Varol concede that their method does not generate the trees in the "natural" order, but claim two advantages: (1) They say their technique is more efficient, since it is "well-known" that the mapping of permutations to trees requires \( O(n^2) \) operations in the worst case for trees with \( n \) nodes. In contrast, their method requires only \( O(n \log n) \) operations to create an \( n \)-node tree, pro-

Sponsored by the United States Army under Contract No. DAAG29-75-C-0024. Partial support by the Research Committee of the Graduate School of the University of Wisconsin-Madison.
vided a table of \( n^2 \) values has been precomputed. They also imply a similar advantage for their calculation of Rank. (2) They point out that a ballot sequence may be efficiently generated from the previous one and converted to a tree, yielding a technique for generating a sequence of trees. They claim that using Knott's method, "to generate \( k \) trees corresponding to consecutive permutations would require the transformation of \( k \) indices, since there is no simple way of deriving stack-sortable permutations in their order corresponding to the natural order of trees" [1, p. 404].

With regard to the first claim, we point out that whereas the mapping of an arbitrary permutation to the corresponding tree may take \( O(n^2) \) operations, a tree permutation may be converted to a tree in \( O(n) \) operations, due to its special properties. Rather than prove this result here, however, we present a modification of Knott's algorithm that translates directly from indices to trees.

With regard to the generation of sequences of trees, we show how to transform a tree to the next one in the natural order by an algorithm that works directly on the tree and requires \( O(n) \) operations.

Definitions

A binary tree \( T \) is either a null tree or consists of a node called the root and two binary trees denoted \( \text{Left}(T) \) and \( \text{Right}(T) \). In the former case, the size of \( T \) is zero; in the latter case \( \text{Size}(T) = 1 + \text{Size}(%& & T) + \text{Size}(\text{Right}(T)) \). We
will often identify a tree with its root.

Define a relation on trees by $T_1 \prec T_2$ if and only if one of the following conditions holds:

- $\text{Size}(T_1) < \text{Size}(T_2)$
- $\text{Size}(T_1) = \text{Size}(T_2)$ and $\text{Left}(T_1) \prec \text{Left}(T_2)$
- $\text{Size}(T_1) = \text{Size}(T_2)$ and $\text{Left}(T_1) = \text{Left}(T_2)$ and $\text{Right}(T_1) \prec \text{Right}(T_2)$.

This ordering is called the natural ordering of trees. Let $\{T_1, T_2, \ldots, T_B\}$ be the sequence of all trees of size $n$, ordered by the natural ordering. It is well known that $B_n$ is the $n$th Catalan number \cite{3}: $B_n = \binom{2n}{n} \frac{1}{n+1}$. Define $\text{Rank}(T_i) = i$. Let $\text{First}(n)$ denote the tree $T_1$, depicted in Figure 1(a). The predicate $\text{Last}(T)$ is true if and only if $T = T_B$, depicted in Figure 1(b).

The Algorithms

In the Pascal \cite{4} program presented in the appendix, a tree $T$ is represented by a pointer to a structure containing the two trees $\text{Left}(T)$ and $\text{Right}(T)$ as well as $\text{Size}(T)$. The size is provided for efficient implementation of Rank and Next.

The main program computes a table of Catalan numbers using the recurrence cited by Knott: $B_n = 4B_{n-1} - \frac{n-1}{n+1}$. It also computes a table of other values which are used to speed up the calculation of $\text{Rank}$ and $\text{RankInverse}$ as described below.

The procedure $\text{Next}$ attempts to transform $T$ to the successor of $T$ in the natural ordering. $\text{Result}$ is set to $\text{true}$ if $\text{Next}$ succeeds or to $\text{false}$ if $\text{Last}(T)$ is true. The method comes
directly from the definition of the natural order. The successor of $T$ may be formed by attempting first to transform $T$'s right subtree to its successor. If $T$'s right subtree has no successor, then it is reset to the first tree of its size and the left subtree is transformed to its successor. If both the subtrees are the last trees of their sizes, then one node is moved from the right to the left and both subtrees are re-initialized.

Let $T$ be a tree and let $n = \text{Size}(T)$. A top-level call of Next gives rise to at most one recursive call for each node of $T$, plus at most one call for each of the $n - 1$ null trees at the leaves. The amount of work in one call to Next, exclusive of embedded calls to Next and to First is bounded by a constant. Each call to First allocates a node of the new tree, so the total work involved in calls to First is bounded by $n$. Thus Next($T$) requires at most $O(n)$ time, where $n = \text{Size}(T)$.

The procedure RankInverse constructs the tree whose rank is $i$ by essentially the same counting argument as the one used by Knott. Let $G_{kn}$ denote the number of trees with $n$ nodes whose left subtree has $k$ nodes. As Knott points out,

$$G_{kn} = B_{k-1}B_{n-k}.$$ Then the size of the left subtree of $T_i$ is the largest integer $r$ such that $S_{rn} = \sum_{k=0}^{r} G_{kn} < i$. As Rotem and Varol point out, this value may be calculated quickly by precomputing some values of $S_{rn}$ and using binary search to find the largest $S_{rn}$ less than $i$. The complexity of finding the root is then $O(\log r) \leq O(\log n)$. The rest of RankInverse calculates the ranks of the left and right subtrees as $\lceil (i - S_{rn})/k \rceil + 1$ and $(i - S_{rn}) \mod k + 1$, respectively, where $k$ is the size of the tree.
right subtree. Therefore, the time devoted to one call of 
\textit{RankInverse} can be bounded by $O(\log n)$, and since each call of 
\textit{RankInverse} generates one node of the resulting tree, the total 
time is $O(n \log n)$.

The procedure \textit{Rank}(T) works by counting the number of trees 
preceeding T in the natural ordering. It divides them into three 
classes: those $T'$ for which $\text{Size}(\text{Left}(T')) < \text{Size}(\text{Left}(T))$, 
those for which $\text{Size}(\text{Left}(T')) = \text{Size}(\text{Right}(T))$ but 
$\text{Left}(T') \neq \text{Left}(T)$, and those for which $\text{Left}(T') = \text{Left}(T)$ but 
$\text{Right}(T') \neq \text{Right}(T)$. The size of the first class is just $S_{rn}$, 
where $r$ is the inorder number of the root, and the other sizes 
can be calculated in a constant amount of time, exclusive of re-
cursive calls. The procedure \textit{Rank} is called once at each node of 
the tree. Hence, if the $S_{rn}$ numbers are precomputed, the running 
time of \textit{Rank} is bounded by $O(n)$.

\section*{Conclusions}

We have presented straightforward and efficient algorithms 
for computing the rank of a tree in the natural ordering of 
binary trees of a given size and for constructing the tree with a 
given rank. We have also presented a new linear algorithm that 
transforms a tree to its successor in the natural ordering.

\section*{References}

\textit{ACM} 20, 2 (Feb 1977), 113-115.


---

(a)  
(b)

**Figure 1**  
The first and last trees with n nodes
Appendix

type tree = ↑ node;
    node =
    record
        left, right : tree;
        size : integer; (* number of nodes in the tree *)
    end;

var  n, k : integer;
    B : array [0 .. MAX] of integer; (* Catalan numbers *)
    S : array [0 .. MAX, 0 .. MAX] of integer;
        (* Sums of products of Catalan numbers *)

procedure Initialize;
(* Precompute the tables B and S *)
begin
    B[0] := 1;
    for n := 1 to MAX do
    begin
        B[n] := 4 * B[n-1] - (5 * B[n-1]) div (n + 1);
        S[0,n] := 0;
        for k := 0 to n - 1 do
            S[k+1,n] := S[k,n] + B[k] * B[n-k-1];
    end (* procedure Initialize *);
end (* First *)

function First(n : integer) : tree;
(* Generate the first tree having n nodes *)
var Result : tree;
begin
    if n = 0 then First := nil
    else begin
        new(Result);
        with Result↑ do
        begin
            size := n;
            left := nil;
            right := First(n - 1);
        end;
        First := Result;
    end;
end (* First *);
procedure Next(T : tree; var Result : Boolean);
(* Change T to the next (in the canonical ordering) tree
   if possible. Report success or failure in Result. *)
label 99;
var
   Ok : Boolean;
   Rsize : integer;
begin
   if T = nil then Result := false
   else with T do
     begin
       Next(right, Ok);
       if right = nil then Rsize := 0 else Rsize := right↑.size;
       if not Ok then
         begin
           Next(left, Ok);
           if not Ok then
             begin
               Rsize := Rsize - 1;
               if Rsize < 0 then
                 begin
                   Result := false;
                   goto 99; (* return *)
                 end;
               left := First(size - Rsize - 1);
               end;
             right := First(Rsize);
           end;
         Result := true;
       end;
     end;
99:
end (* Next *);

function Rank(T : tree) : integer;
var
   Lsize, Lrank, Rsize, Rrank : integer;
begin
   if T = nil then Rank := 1
   else with T do
     begin
       if left = nil then Lsize := 0 else Lsize := left↑.size;
       Lrank := Rank(left);
       if right = nil then Rsize := 0 else Rsize := right↑.size;
       Rrank := Rank(right);
       Rank := B[Rsize] * (Lrank - 1)
            + Rrank
            + S[Lsize, size];
     end;
end (* Rank *);
function RankInverse(i, n : integer) : tree;
(* Return the tree whose rank is i among those with n nodes *)
var
  Low, High, Center : integer; (* For binary search *)
  Lsize, Rsize : integer;
  Result : tree;
begin
  if n = 0 then RankInverse := nil
  else begin
    (* Set High = max { k - S[k,n] < i } using binary search. *)
    Low := 0;
    High := n - 1;
    repeat
      Center := (Low + High) div 2;
      if i > S(Center, n) then Low := Center + 1
      else High := Center - 1;
    until Low > High;
    Lsize := High;
    Rsize := n - Lsize - 1;
    i := i - S(Lsize, n) - 1;
    new(Result);
    with Result do
      begin
        left := RankInverse(i div B[Rsize] + 1, Lsize);
        right := RankInverse(i mod B[Rsize] + 1, Rsize);
        size := n;
      end;
    RankInverse := Result;
  end;
end (* RankInverse *);
A NOTE ON ENUMERATING BINARY TREES

Gary Knott has presented algorithms for computing a bijection between the set of binary trees on n nodes and an initial segment of the positive integers. Rotem and Varol—presented a more complicated algorithm, that computes a different bijection, claiming that their algorithm is more efficient and has advantages if a sequence of several consecutive trees is required. We present a modification of Knott's algorithm that is simpler than Knott's and as efficient as Rotem and Varol's. We also give a new linear-time algorithm for transforming a tree into its successor in the natural ordering of binary trees.