On Rotations and the Generation of Binary Trees

JOAN M. LUCAS *
Department of Computer Science
Rutgers University
New Brunswick, New Jersey 08903

D. ROELANTS VAN BARONAIGIEN † & FRANK RUSKEY ‡
Department of Computer Science
University of Victoria
Victoria, B.C. V8W 2Y2, Canada

Abstract

The *rotation graph*, G_n, has vertex set consisting of all binary trees with n nodes. Two vertices are connected by an edge if a single rotation will transform one tree into the other. We provide a simpler proof of a result of Lucas [7] that G_n contains a Hamilton path. Our proof deals directly with the pointer representation of the binary tree. This proof provides the basis of an algorithm for generating all binary trees that can be implemented to run on a pointer machine and to use only constant time between the output of successive trees.

Ranking and unranking algorithms are developed for the ordering of binary trees implied by the generation algorithm. These algorithms have time complexity $O(n^2)$ (arithmetic operations).

We also show strong relationships amongst various representations of binary trees and amongst binary tree generation algorithms that have recently appeared in the literature.

*Current address: SUNY Brockport, Mathematics & Computer Science Dept., Brockport, NY 14420

†Research supported by the Natural Sciences and Engineering Research Council of Canada under grant 89767.

‡Research supported by the Natural Sciences and Engineering Research Council of Canada under grant A3379.

1 Introduction

Many algorithms have been published for generating binary trees. In most of the algorithms, the trees are encoded as integer sequences and then those sequences are generated lexicographically. See, for example, Pallo [10], Ruskey and Hu [13], Roelants van Baronaigien and Ruskey [21], Zerling [23] and Zaks [22]. Recently, some attention has been given to the question of generating the trees (or their representations) in such a way that successively generated trees (or their representations) differ by a constant amount (Proskurowski and Ruskey [12], Lucas [7], Ruskey and Proskurowski [14], Roelants van Baronaigien [19]). Other than the algorithm of [7], existing published algorithms do not generate binary trees in their familiar computer representation using pointers to the left and right subtrees. Of course, the other algorithms could in principle be modified to produce the trees themselves, but usually at a cost of greatly complicating the algorithm and/or invalidating the run time analysis.

We follow Knuth [4] in defining a *binary tree* to be either empty or to consist of a root node and a left and a right subtree, both of which are binary trees. Nodes are drawn as circles in the figures and arbitrary subtrees are drawn as triangles. Binary trees are classified by their size; i.e., by the number of nodes in the tree. We shall assume that the nodes of any tree of size n are numbered from 1 to n according to an inorder traversal of the tree, and shall not distinguish between a node and its number. We define the *right path* of a binary tree to be the path in the tree from the root node to the largest node; i.e., the maximal path formed by following right child pointers from the root. The number of trees with n nodes is the well-known *Catalan number* $C_n = (2n)!/(n!(n+1)!)$.

A *rotation* in a binary tree is the familiar operation used in maintaining balanced binary search trees (Knuth [5]). Figure 1 shows a left and a right rotation at a node x. Note the asymmetry with respect to x. We also refer to these two rotations as rotations of the edge (w, x). We will develop an algorithm for generating all binary trees so that each successive tree differs from its predecessor by a single left or right rotation. Note that a rotation does not change the inorder numbering of the nodes.

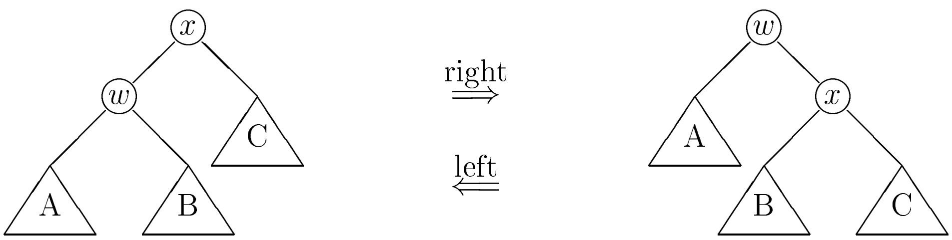


Figure 1: A left or right rotation at node x.

Our approach to the problem uses the paradigm of attaching alternating directions to elements or positions of the object in order to obtain "Gray codes" of combinatorial objects; see Proskurowski and Ruskey [12], Nijenhuis and Wilf [9], or Joichi, White, and Williamson [3] for examples.

The rotation graph G_n has vertex set consisting of all binary trees with n nodes. Two vertices are connected by an edge if a single left or right rotation will transform one tree into the other. This graph was used by Pallo [10], who showed that the directed version obtained by using only left rotations is a lattice. Further properties of this lattice are given in Pallo [11]. The problem of determining the diameter of G_n was considered by Sleator, Tarjan, and Thurston [15]. Lucas [7] showed that the rotation graph has a Hamilton path. In section 2 of this paper we develop a simpler and easier to understand proof of this result. Our algorithm does not require the maintenance of any sequence representation of the binary tree and can be implemented to run in constant time per tree on a pointer machine, neglecting the time needed to actually print the tree.

In [7] the more difficult result that the rotation graph has a Hamilton cycle is proven. It remains a challenge to provide a simpler proof that a cycle exists.

The *rank* of a binary tree is the number of trees that precede it in the list. By *unranking* we mean the process of taking a number and producing the binary tree with that number as its rank. In section 3 we present $O(n^2)$ ranking and unranking algorithms. It should be pointed out that in these algorithms we count such operations as the addition or comparison of two integers to be a unit cost operation. The numbers can be as large as the Catalan numbers, and hence require $\Theta(n)$ bits to represent them. Thus the ranking and unranking algorithms require $\Theta(n^3)$ bit operations.

Ranking algorithms are studied because they provide perfect hashing functions for the objects being ranked, and they often lead to interesting enumeration questions. Unranking algorithms are useful for producing a binary tree at random, and for generating binary trees in parallel. It is to be noted that $O(n)$ ranking algorithms for binary trees exist for other orderings [13], [22]. We believe that the complexity of ranking provides some complexity measure of the underlying ordering. Somehow the ordering of Lucas [7] and this paper is more complicated than the traditional lexicographic orderings. We do not have a lower bound for our ranking algorithm, but there is no obvious way to improve it. The problem of ranking and unranking was not considered in [7]. Some of the results of sections 2 and 3 were first reported in [20].

There are many different representations for binary trees. In Section 4 we discuss the representations used by Zerling [23], Zaks [22], Mäkinen [8] and Pallo [10], and we provide simple mappings between these various representations.

In section 5 we describe the generation algorithms of Lucas [7], Zerling [23], Pallo [10], and Mäkinen [8]. Having described the algorithms we then show that Algorithm `List` of Section 2 and the the above algorithms all have the same underlying recursive computation tree when that tree is regarded as a rooted, unordered tree. We further show that where the algorithms produce representations lexicographically ([8],[10],[23]), the recursion trees are identical. Similarly, the algorithms that produce successive trees that differ by a constant amount (`List`, [7]) have identical recursion trees.

2 The Generation Algorithm

A Hamilton path of G_n can be generated using a recursive strategy. We show how to construct a Hamilton path in G_n from a Hamilton path in G_{n-1}.

The proof uses the idea of an *induced subtree* of a binary tree. Let T be a binary tree of size n. We define $T[i..j]$ to be the tree formed by the nodes i, $i+1,\ldots,j$, where the parent of k, $i \leq k \leq j$, in $T[i..j]$ is defined to be the node l, $i \leq l \leq j$, such that l is the nearest proper ancestor of k in T. This is simply the tree formed by contracting any edge in T whose endpoints are not both in the range $[i..j]$. See Figure 2.

Consider the effect of a rotation of the edge (x, y) in T on the tree $T[i..j]$. If both x and y are in the range $[i..j]$ then this rotation has the effect of rotating the same edge in $T[i..j]$, otherwise $T[i..j]$ is unchanged. Notice that in the example of Figure 2 it is possible to rotate the edge (x, y) in $T[i..j]$ but not in T, because in T the nodes u and v are "in the way".

This definition has been used for several divide-and-conquer type arguments that have been applied to binary trees, see Tarjan [18], Lucas [6], Cole [2], or Sundar [17]. In the remainder of this paper we will only use induced subtrees with $i = 1$.

If one is given $T[1..n-1]$, then the only possible way to form a corresponding tree T is to place node n as the root of $T[1..n-1]$, as the right child of $n-1$ in $T[1..n-1]$, or "between" any two nodes along the right path of $T[1..n-1]$. See Figure 3.

From this observation we can derive the following result.

THEOREM 1 *The rotation graph G_n contains a Hamilton path.*

PROOF: The proof is by induction on n. Clearly there exists a Hamilton path in G_1. There is only one binary tree that contains a single node.

Let n be any integer greater than 1. By induction there exists a sequence $P_{n-1} = T_0, T_1, \ldots, T_k$ of the trees of size $n-1$ that forms a Hamilton path in G_{n-1}. Using P_{n-1}, we construct a sequence P_n of the trees of size n that forms a Hamilton path in G_n. The initial tree in this sequence is formed by placing node n as the right child of $n-1$ in T_0. We then perform precisely the same sequence of rotations used to generate P_{n-1}, but before performing each rotation of P_{n-1} we first rotate n as follows:

1. If n is not the root, then repeatedly rotate the edge $(parent(n), n)$ until n is the root. These are left rotations at n.

2. If n is the root, then repeatedly rotate the edge $(leftchild(n), n)$ until n has no left child. These are right rotations at n.

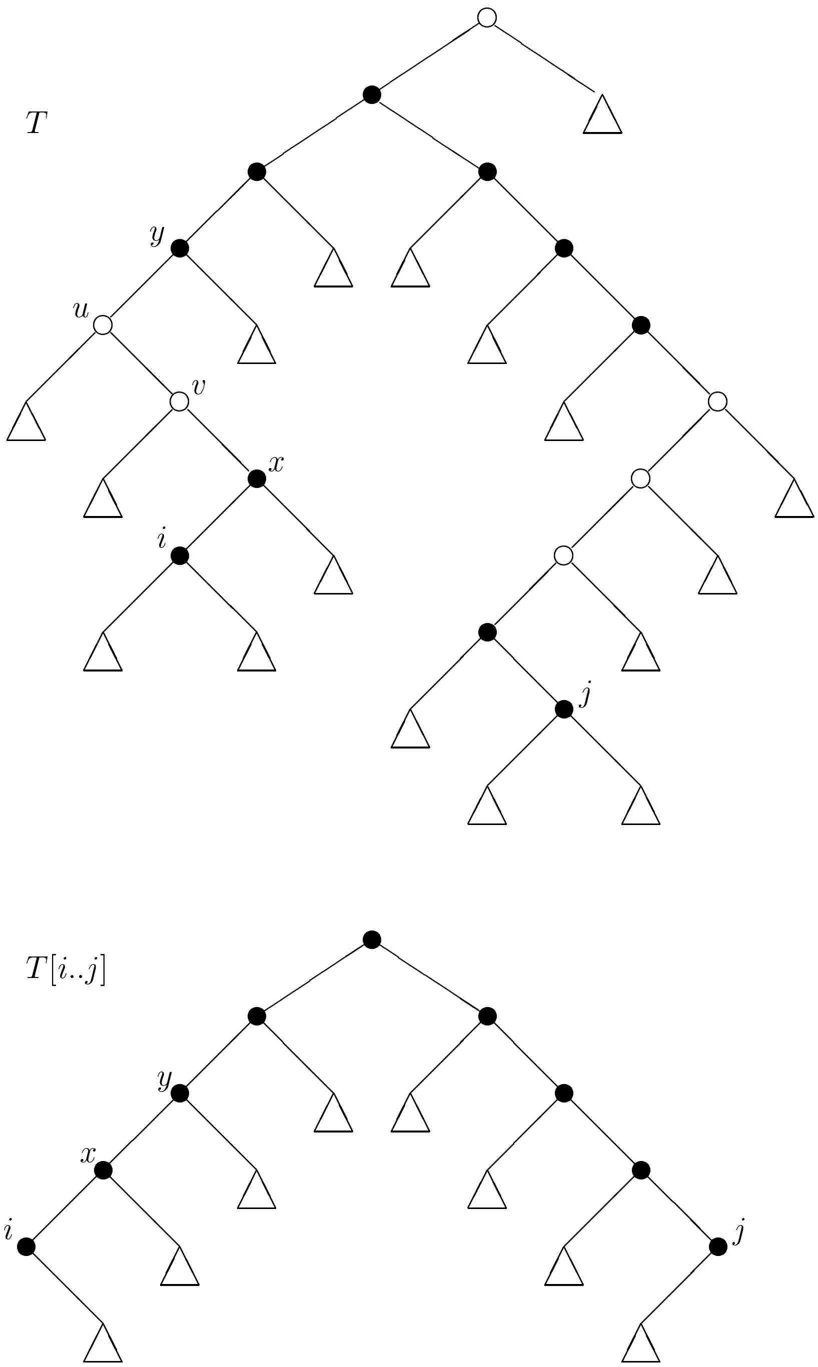
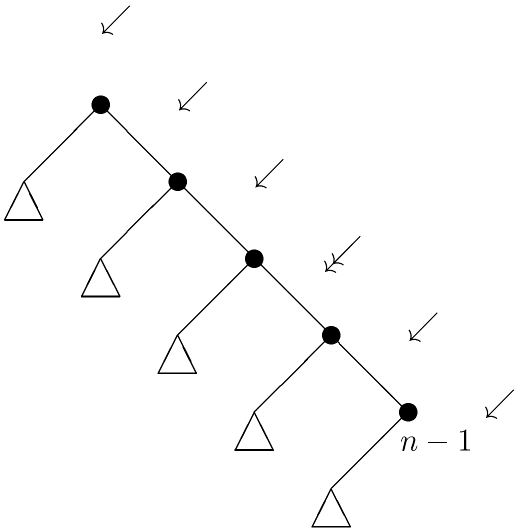
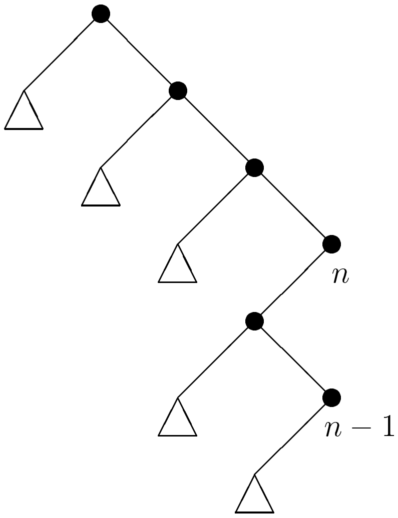


Figure 2: Example of induced tree.



(a) $T[1..n-1]$



(b) T

Figure 3: (a) Given $T[1..n-1]$ there are 6 places where node n could possibly be inserted to get T. (b) Node n has been inserted at the position of the double arrow.

Finally, after performing the last rotation of P_{n-1}, we repeat the step above.

Clearly every tree T of size n is generated, since every possible shape of the subtree $T[1..n-1]$ is generated, and for each such shape all the trees of size n whose induced subtree of the nodes 1 to $n-1$ has that shape have been generated.

Clearly it is possible to perform all of the rotations at n. Any rotation that does not contain n as one of the endpoints of the rotated edge can also be performed. Such a rotation is possible as long as n is not "in the way", which is true since n is either the root of the tree or the right child of $n-1$ whenever such a rotation of P_{n-1} is done. $\square$

This proof leads directly to an efficient algorithm for generating all binary trees by rotations. One such implementation is given in Figure 4. The initial tree is skewed to the left and the initial call is `List(t)`, where `t` points to the second node in inorder.

The running time of this algorithm is determined by the number of recursive calls to `List`, i.e., by the size of the recursion tree. From the proof above we know that the number of nodes in the recursion tree is the sum of the first n Catalan numbers, since the list P_n of all C_n binary trees with n nodes is created from the list P_{n-1} of all C_{n-1} binary trees with $n-1$ nodes, and so on. This sum is bounded above by $1.5C_n$ (for $n \geq 6$). This bound may be proven by induction from the recurrence relation $C_n = (4n-2)C_{n-1}/(n+1)$. Thus `List` runs in constant average time (i.e., $O(1)$ amortized time) per tree.

The algorithm to generate the sequence of trees can also be implemented on a pointer machine in $O(1)$ *worst case* time per rotation. In the remainder of this section we explain how to do this.

The algorithm requires each node to store the usual pointers to its parent, left child and right child in the tree, and in addition two more pointers and a *direction* bit. The direction bit for node i indicates whether i is currently rotating left on its way to becoming the root of $T[1..i]$ (i.e., direction(i) = up) or whether i is currently rotating right on its way to becoming the right child of $i-1$ (i.e., direction(i) = down).

We need only show that the next node to be rotated can be located in $O(1)$ time.

At any given time in the generation procedure each node i is currently in the process of rotating either up or down the right path of $T[1..i-1]$. We say i is *unfinished* if at the current time i has not yet completed its rotations along the right path of $T[1..i-1]$. Otherwise we say i is *finished*.

Initially all the nodes are unfinished except for node 1, and the initial tree is the one where each node, except node n, has a right subtree and no left subtree.

At any point in the generation procedure, if T is the current shape of the tree, then the next node that is to be rotated, node j say, is the largest (rightmost) unfinished node. All nodes greater than j are either proper ancestors of the root of $T[1..j]$ or in the right subtree of j.

After the rotation at node j, every node i, $i > j$ commences to rotate along the

```
{ Let t point to a binary tree T, where t^ is the k-th node in inorder. }
{ List(t) generates all binary trees with induced subtree T[1..k-1].    }
procedure List( t : ptr );
begin
   if t = nil then PrintTree
   else begin
      List( t^.sym_succ );
      if t^.left = nil then {increasing}
         while LeftRotationPossible( t ) do begin
            RotateLeft( t );
            List( t^.sym_succ );
         end
      else {decreasing}
         while RightRotationPossible( t ) do begin
            RotateRight( t );
            List( t^.sym_succ );
         end;
   end;
end {of List};
```

Figure 4: Pascal procedure to generate trees by rotations.

right path of the resulting tree $T'[1..i-1]$. But before and after each such rotation at i, all trees T'' whose subtree $T''[1..i]$ is identical to the current induced subtree of the nodes from 1 to i are generated.

Consider a rotation at node j at some point during the generation procedure. Then the state of every node i, $i < j$, remains unchanged but the state of every node i, $i > j$, changes from finished to unfinished. The node j, which was unfinished before this rotation, may be either finished or unfinished after this rotation.

Thus our algorithm can easily identify the next rotation to be performed by maintaining two lists, the list of finished nodes and the list of unfinished nodes. These two lists partition the nodes of the tree. On each list the nodes will appear in decreasing order. Each node i has a pointer *next* to the node following i on the list containing i. The *next* pointers are used for both the list of finished and unfinished nodes, since a node can be on only one list. This facilitates the updating of the lists needed after each rotation. We also maintain for each node i a pointer *sym_succ* to its inorder successor $i+1$.

The next node to be rotated is simply the first node on the list of unfinished nodes. These two lists of nodes can easily be updated in constant time after each rotation by making $O(1)$ pointer changes. The details are given in the code for this generation algorithm which appears in Appendix A.

3 Ranking and Unranking

Recall that the *rank* of a tree is the number of trees generated before it. An unranking algorithm takes a number r as input, and constructs the tree that has r as its rank.

To do the ranking and unranking it is necessary to know in which direction each node is currently rotating. When generating the trees of size n, the corresponding list of induced subtrees of the nodes 1 to $n-1$ is the same as the sequence of trees of size $n-1$ generated by the algorithm. The number of times the direction of node j changes is precisely the rank of the tree $T[1..j-1]$. If the rank is even then j is moving up (rotating left), otherwise it is moving down (rotating right).

It is also necessary to know the number of possible trees T of size n whose induced subtree $T[1..j]$ has some restricted shape.

DEFINITION 1 *We define $B(\tau, n, j, p)$ to be the number of trees T of size n whose induced subtree $T[1..j]$ is equal to the tree τ, where τ is a tree of size j whose right path contains exactly p nodes.*

Notice that the value of $B(\tau, n, j, p)$ is the same for all trees τ of size j whose right path has p nodes. Therefore we shall omit τ from the B notation in what follows.

These numbers are related to $T(n,k)$, the number of binary trees with n internal nodes where the left most leaf occurs on level k as used by Ruskey and Hu [13] (these are a shifted version of the Delannoy numbers). They satisfy the initial conditions $T(n,0)=0$, $T(n,n)=1$, and the following recurrence relation for $0<k<n$.

$$T(n,k) = T(n-1,k-1) + T(n,k+1) \tag{1}$$

The following explicit expression is also known [13].

$$T(n,k) = \frac{k}{2n-k}\binom{2n-k}{n-k}$$

LEMMA 1 *For all $1 \le j \le n$ and $1 \le p \le j$,*

$$B(n,j,p) = T(n-j+p+1, p+1)$$

PROOF: We will prove this by induction on decreasing values of j. Clearly when $j=n$ there is only one possible binary tree, i.e. $B(n,n,p)=1$, and $T(p+1,p+1)$ is also one.

Assume that $B(n,k,p) = T(n-k+p+1, p+1)$ for all values of k larger than j. The set of binary trees T where $T[1..j]$ is equal to some tree τ of size j where the right path of τ contains p nodes is the set of binary trees T' where $T'[1..j+1]$ is equal to a binary tree that can be formed by adding node $j+1$ to τ as the right child of j and then performing between 0 and p left rotations at $j+1$. There are $p+1$ possible shapes for $T'[1..j+1]$ given τ. Thus

$$B(n,j,p) = \sum_{i=1}^{p+1} B(n,j+1,i) \tag{2}$$

and so by the inductive assumption

$$B(n,j,p) \ = \ \sum_{i=1}^{p+1} T(n-j+i,i+1) \ = \ T(n-j+p+1,p+1).$$

The second equality follows by iterating (1). □

Define $R(T,j)$ to be the rank of the first tree T' in the generated sequence such that $T'[1..j] = T[1..j]$. Also let $r(T,j) = R(T,j) - R(T,j-1)$, and let p_j denote the number of nodes on the right path of $T[1..j]$. We can see that the rank of any tree T is given by the following equation.

$$rank(T) \ = \ R(T,n) \ = \ \sum_{j=2}^{n} r(T,j),$$

where the numbers $r(T,j)$ can be computed as follows.

$$r(T,j) = \begin{cases} \displaystyle\sum_{i=p_j+1}^{p_{j-1}+1} B(n,j,i) & \text{if } rank(T[1..j-1]) \text{ is even} \\ \\ \displaystyle\sum_{i=1}^{p_j-1} B(n,j,i) & \text{otherwise} \end{cases}$$

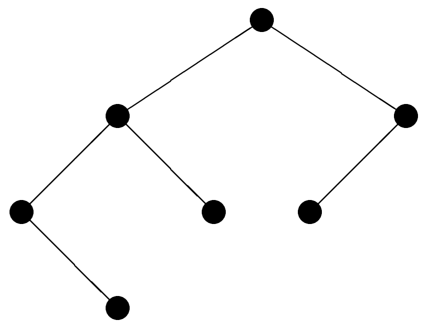
By (2) the above formula for $r(T,j)$ can be simplified as shown below.

$$r(T,j) = \begin{cases} B(n,j-1,p_{j-1}) - B(n,j-1,p_j-1) & \text{if } rank(T[1..j-1]) \text{ even} \\ B(n,j-1,p_j-2) & \text{otherwise} \end{cases}$$

As an example, consider the tree of Figure 5; the accompanying table gives the rank of each tree $T[1..j]$ and the values of $r(T,j)$.

The time complexity of the ranking algorithm is $O(n^2)$ (or $O(n^3)$ if bit operations are counted). This results from the need to compute the values $r(T[1..i],j)$, for all i, $2 \le i \le n$ and j, $1 \le j \le i$. We need all of these values because we must determine the rank, or rather the parity of the rank, of each tree $T[1..i]$.

We have recently learned that Bent [1] has found a way to determine the parity of $T(n,k)$ in $O(\log n)$ bit operations. He has used this result to give a ranking algorithm for the same ordering of trees as used in this paper that uses $\Theta(n^2 \log n)$ bit operations,



j	rank $(T[1..j-1])$	p_j	$r(T,j)$
1		1	
2	0	2	0
3	0	1	255
4	2	2	0
5	7	1	0
6	23	2	2
7	77	2	1
8	258		

Figure 5: Example tree for ranking.

so long as the n by n table of the $T(n,k)$ values has been precomputed. The same analysis applies to our ranking algorithm as well.

We present in Figure 6 an unranking algorithm. This algorithm builds the tree a node at a time. It has the same structure as the listing algorithm, except that a simple iteration replaces the recursion. As each node is added to the tree, either at the root or along the rightmost path, it is then rotated into its proper position. The ranks of the subtrees $T[1..i]$ are used to determine the direction of each node in T.

The time complexity of the unranking algorithm is dominated by the time taken to compute the rank of each tree $T[1..j]$, $1 \le j \le n$. All of these values are computed during the $O(n^2)$ time procedure call that computes $rank(T)$. Unfortunately, because of the need to know the parity of the rank of each subtree $T[1..i]$, we have been unable to reduce the running time of the ranking or unranking algorithms to less than $O(n^2)$ arithmetic operations.

4 Sequence correspondences

Almost all binary tree generation algorithms have produced integer or bit sequence representations of binary trees. Since all of the different representations are derived from some structural property of binary trees, there must be some natural correspondences between the different representations. In this section we discuss the *codeword* representation used by Zerling [23], Lucas [7] and Roelants van Baronaigien [19], the *weight sequence* used by Pallo [10] and Roelants van Baronaigien and Ruskey [20], the *bitstring* used by Zaks [22] and Proskurowski and Ruskey [14], and the *distance* representation used by Mäkinen [8]. Although these different representations differ greatly in how they are constructed, we show that there are simple order-preserving bijections between them.

We start by briefly describing each of the representations.

The *weight sequence* representation was introduced by Pallo [10]. The weight sequence $w_1, w_2, \ldots, w_n$ of a binary tree with n nodes is obtained by labeling each of the nodes of the tree with one plus the number of nodes in its left subtree. These labels are then listed in inorder as a sequence. For example, the tree of Figure 7 has

```
function Unrank ( rank : integer ) : BinaryTree;
var   tree : BinaryTree;
      j, p : integer;
begin
      tree := "create tree of one node";
      { add each of the nodes to the tree in turn }
      for j := 2 to n do begin
            if odd( Rank( tree ) ) then begin
                  "add node j as the root";
                  p := 1;
                  while rank > B(n, j, p) do begin
                        rank := rank − B(n, j, p);
                        RotateRight( j );
                        p := p + 1;
                  end;
            end else begin
                  "add node j as the right child of node j − 1";
                  p := "number of ancestors of j in the tree";
                  while rank > B(n, j, p) do begin
                        rank := rank − B(n, j, p);
                        RotateLeft( j );
                        p := p − 1;
                  end;
            end;
      end;
      Unrank := tree;
end {of Unrank};
```

Figure 6: The unranking procedure.

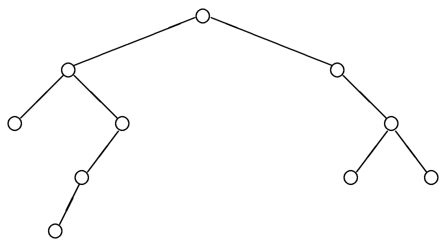


Figure 7: The tree with weight sequence 1,2,1,2,3,6,1,1,2,1.

weight sequence $1, 2, 1, 2, 3, 6, 1, 1, 2, 1$. The set of all length n weight sequences is denoted $\mathbf{W}(n)$.

The *distance* representation, introduced by Mäkinen [8], is a sequence $d_1, d_2, \ldots, d_n$, where d_i is the number of proper ancestors of i in $T[1..i]$. For example, the tree of Figure 7 has distance sequence $0, 0, 1, 1, 1, 0, 1, 2, 2, 3$. Any sequence $d_1, d_2, \ldots, d_n$ where $d_1 = 0$ and $0 \le d_i \le d_{i-1} + 1$ for i, $2 \le i \le n$ is a valid distance sequence. The set of all length n distance sequences is denoted $\mathbf{D}(n)$.

Clearly, any prefix $w_1, w_2, \ldots, w_k$ of the weight sequence for T is the weight sequence for the tree $T[1..k]$ because every node in $T[1..k]$ has the same left subtree in T and $T[1..k]$. Similarly, any prefix $d_1, d_2, \ldots, d_k$ of the distance sequence for T is the distance sequence for the tree $T[1..k]$.

The *codeword* representation was introduced by Zerling [23]. For convenience, and ease of understanding, we index the sequence in reverse order and add an additional element to the start of the sequence that is always zero. A codeword is a non-negative integer sequence $c_1, c_2, \ldots, c_n$ where (3) below holds.

$$\sum_{i=1}^{j} c_i \le j - 1 \qquad \text{for } j = 1, 2, \ldots, n. \tag{3}$$

The tree T that corresponds to $c_1, c_2, \ldots, c_n$ is formed by the following process. Start with the tree that consists of only a left path of n nodes. For each value of i from 2 to n perform c_i right rotations at the root of $T[1..i]$. (Note that $T[1..i]$ changes in this process.) The resulting tree is T. Zerling calls this process the "unfolding" of the binary tree from the left path. The codeword for the tree given in Figure 7 is 0011100103. The set of all length n codewords is denoted $\mathbf{C}(n)$.

The last representation to be discussed is the oldest. The *bitstring* representation is a binary sequence $b_1 b_2 \cdots b_{2n}$ containing n 1's and n 0's and such that every prefix contains at least as many 1's as 0's; i.e., where $2\sum_{i=1}^{j} b_i \ge i$, for $i = 1, 2, \ldots, 2n$. To construct a bitstring from a binary tree T, do a preorder traversal of $E(T)$, where $E(T)$ is the "extended" version of T with a leaf for every empty subtree of T. As the traversal is performed, write a 1 at each internal (non-leaf) node and a 0 at each leaf. The last bit is always a zero, and is omitted. For example, the tree in Figure 7 is represented by the bitstring 11100111000010110010. By $\mathbf{B}(n)$ we denote the set of all length $2n$ bitstring representations.

Table 1 lists the various representations for $n = 4$ in lexicographic or reverse lexicographic order.

The following theorem reveals close relationships between the various representations. These relationships are based solely on the properties of the sequences, and not of the corresponding trees.

Theorem 2 *There are simple $O(n)$ bijections between $\mathbf{B}(n)$, $\mathbf{C}(n)$, $\mathbf{D}(n)$, and $\mathbf{W}(n)$ that preserve or reverse lexicographic order.*

codewords Alg-Z $\{c_i\}_{i=1}^n$	weights Alg-P $\{w_i\}_{i=1}^n$	distance Alg-M $\{d_i\}_{i=1}^n$	bitstring Alg-RP $\{b_i\}_{i=1}^{2n}$
0000	1111	0123	11110000
0001	1112	0122	11101000
0002	1113	0121	11100100
0003	1114	0120	11100010
0010	1121	0112	11011000
0011	1123	0111	11010100
0012	1124	0110	11010010
0020	1131	0101	11001100
0021	1134	0100	11001010
0100	1211	0012	10111000
0101	1212	0011	10110100
0102	1214	0010	10110010
0110	1231	0001	10101100
0111	1234	0000	10101010

Table 1: The equivalent sequences for binary trees with 4 nodes

PROOF: We will show how to construct $\mathbf{B}(n)$, $\mathbf{D}(n)$, and $\mathbf{W}(n)$ from $\mathbf{C}(n)$. Refer to Table 1. Let $\mathbf{c} = c_1, c_2, \ldots, c_n$ be a codeword in $\mathbf{C}(n)$.

The corresponding bitstring sequence is $10^{c_2}10^{c_3}1 \cdots 0^{c_n}10^N$, where 0^i is a string of i zeroes and $N = n - \sum c_i$. Condition (3) guarantees that this is a valid bitstring. It should be clear that this correspondence reverses lexicographic order.

To construct the the distance sequence $d_1, d_2, \ldots d_n$ from $\mathbf{c}$, set $d_1 = 0$ and $d_i = 1 + d_{i-1} - c_i$, for $i = 2, 3, \ldots, n$. Again, it is clear that this correspondence reverses lexicographic order.

To construct the weight sequence $w_1, w_2, \ldots w_n$ from $\mathbf{c}$ we may use the following $O(n)$ algorithm. Recall that the update $w_i := w_i + w_{i-w_i}$ corresponds to a single left rotation at node i.

> **for** $i = 1, 2, \ldots, n$ **do** $w_i := 1$;
> **for** $i = 1, 2, \ldots, n$ **do**
>   **for** $j = 1, 2, \ldots, c_i$ **do** $w_i := w_i + w_{i-w_i}$;

This correspondence preserves lexicographic order. If the subsequence $c_1, c_2, \ldots c_{i-1}$ is the same as the subsequence $c'_1, c'_2, \ldots, c'_{i-1}$ but $c_i > c'_i$, then the corresponding sequences $w_1, w_2, \ldots, w_{i-1}$ and $w'_1, w'_2, \ldots w'_{i-1}$ are identical, but $w_i > w'_i$. $\square$

Having shown correspondences between the representations used by a number of

```
procedure GenCD(i)
    if i > n then "output tree sequence"
    else
        for k ∈ {0, 1, ..., 1 + d_{i-1}} (* or k ∈ {0, ..., i - 1 - Σ_{j=1}^{i-1} c_j} *) do
            d_i ← k (* or c_i ← k *);
            GenCD(i + 1)
```

Figure 8: Algorithms to generate $\mathbf{D}(n)$ (Alg-M) and $\mathbf{C}(n)$ (Alg-Z and Alg-L).

algorithms, it is only natural to consider correspondences between the algorithms that generate these representations.

5 Algorithm Correspondences

We now show that the underlying recursion trees for the algorithms presented here and in [23], [7], [8], [14] and [10] are isomorphic. We first present a brief description of each algorithm.

The algorithm of Zerling [23], which we refer to as Alg-Z, is a simple recursive algorithm which generates *codewords* in lexicographic order. A codeword is generated from position 1 to position n, where the maximum value in position i is $i-1-\sum_{j=1}^{i-1} c_j$. This maximum value can be maintained using $O(1)$ computation per recursive call.

The algorithm of Mäkinen [8], which we refer to as Alg-M, also generates sequences in lexicographic order. Alg-Z and Alg-M are very similar, and both are presented in Figure 8 (the comments should be shifted left to get Alg-Z). To generate the sequences in lexicographic order the elements of the **for** loop must be selected in increasing order. Initially, set $d_1 \leftarrow 0$ or $c_1 \leftarrow 0$ and then call GenCD(2).

The last of the three lexicographic generation algorithms is slightly more complicated. Pallo's algorithm [10], which we refer to as Alg-P, is given in Figure 9. Initially, set $w_1 \leftarrow 1$ and then call Lex_GenW(2).

The recursion tree for these three algorithms when $n = 4$ is given in Figure 10.

We also consider the underlying recursion trees of three algorithms which list binary trees in a "Gray Code" manner. The first of these algorithms is due to Lucas [7] and will be referred to as Alg-L. It has the same form as Alg-Z of Figure 8, except that the elements of the **for** loop are selected alternately in increasing and decreasing order at each level (value of i). This requires some additional storage, say a direction array indexed $2..n$. The algorithm of Roelants van Baronaigien and Ruskey [20] uses the same idea of alternating directions, except that it is based on Alg-P instead of Alg-Z. We refer to it as Alg-RR. The recursion trees for these two algorithms when $n = 4$ is given in Figure 11.

procedure Lex_GenW(i)

  if $i > n$ **then** "output tree sequence"

  else

    $w_i \leftarrow 1$; Lex_Gen(i+1);

    while $w_i \neq i$ **do**

      $w_i \leftarrow w_i + w_{i-w_i}$;

      Lex_GenW($i+1$);

Figure 9: Algorithm to generate $\mathbf{W}(n)$ (Alg-P).

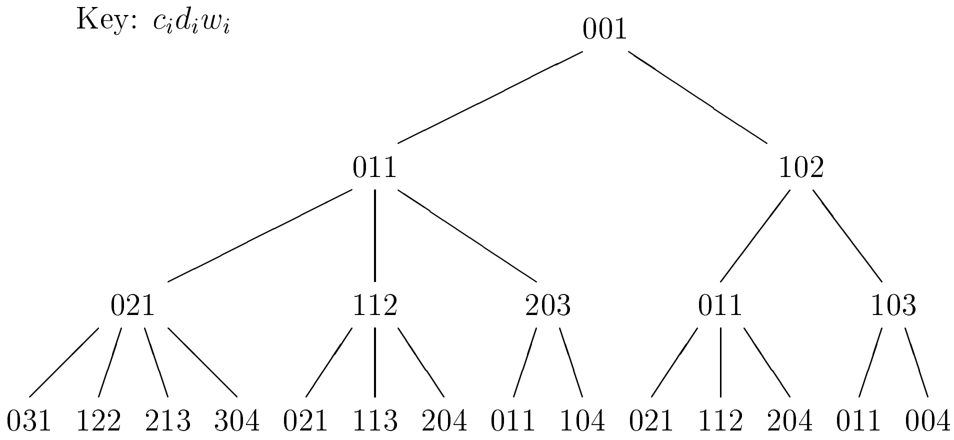


Figure 10: Recursion trees for Alg-Z, Alg-M (mirror image), and Alg-P.

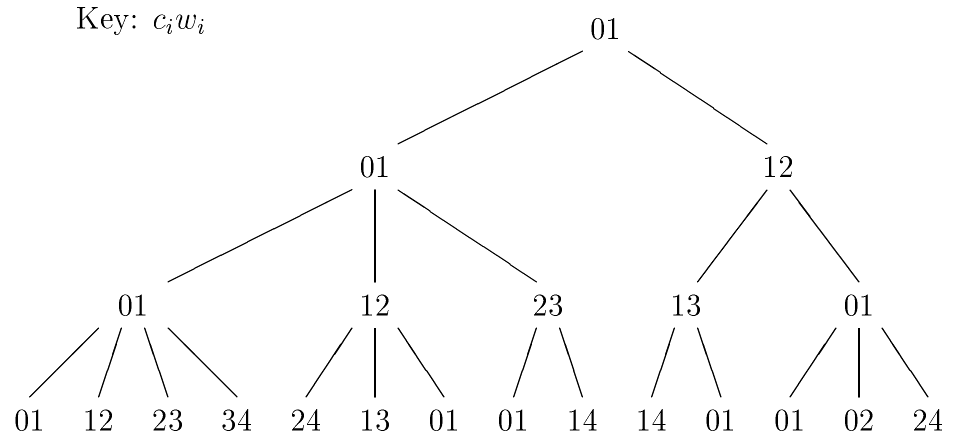


Figure 11: Recursion trees for Alg-L and Alg-RR.

The last "Gray Code" type generation algorithm is `List` as presented in Section 2, which will be referred to here as Alg-LRR.

One might suspect that these algorithms have isomorphic recursive computation trees, but not all binary tree generation algorithms have similar recursion trees. For example, the algorithm presented in [21] has a recursion tree that is not isomorphic to any of the recursion trees of the above algorithms.

THEOREM 3 *For a given value of n, algorithms Alg-L, Alg-M, Alg-P, Alg-LRR and Alg-Z have underlying recursion trees that are isomorphic as rooted unordered trees.*

PROOF: Define the *degree* of a call to be the number of direct recursive calls it makes. The *level* of a call is the value of the parameter i when the call is made. We show that, for each of these algorithms, the initial call is of degree 2, a call of degree p results in recursive calls of degree $2, 3, \ldots, p+1$ or results in the output of p objects, and that the level of recursion that produces output is $n+1$ for all of the algorithms. It is easy to verify that the initial calls are of degree 2 and that all output occurs at level $n+1$. We now show that a call of degree p results in calls of degree $2, 3, \ldots, p+1$.

For Alg-M the result is obvious.

The degree of a call of Alg-L (Alg-Z) at the $(i-1)$st level is $p = i - \sum_{j=1}^{i-1} c_j$. The resulting calls at level i set c_i to be $0, 1, \ldots, p-1$ with corresponding recursive calls of degree $p+1, p, \ldots, 2$, respectively.

Alg-P produces weight sequences lexicographically. Given $w_1, w_2, \ldots, w_{i-1}$, for some p, the possible values for w_i are $s_1 < s_2 < \cdots < s_p$, where $s_1 = 1$, $s_p = i$, and $s_{j+1} = s_j + w_{i-s_j}$ if $j > 0$. If $w_i = s_j$, then the possible values for w_{i+1} are $1, 1+s_j, 1+s_{j+1}, \ldots, 1+s_p$; the corresponding call at level i has degree $p-j+2$. Thus, considering all values of $j = 1, 2, \ldots, p$, the degrees of the calls at level i are $p+1, p, \ldots, 2$, respectively. The same argument hold for Alg-RR, except that the ordering of subtrees can be reversed.

Finally, consider the generation algorithm `List` of this paper. If a call `List(t)` has degree p, then it is possible to do exactly $p-1$ rotations at the parameter node t. We shall assume that the $p-1$ rotations are left rotations; the argument is similar for the case of right rotations. If the call has done, say, k of these rotations before one of the recursive calls `List(t^.sym_succ)`, then the recursive call performs $p-k$ rotations, which means it is a call of degree $p-k+1$. Since $0 \le k < p$, a call of degree p results in calls of degree $2, 3, \ldots, p+1$. □

In the following discussion, we consider ordered trees T and T' to be *equivalent* if they are isomorphic (as ordered trees) or if T is isomorphic to the mirror image of T'. While all of the algorithms considered in Theorem 3 have recursion trees that are isomorphic as rooted trees, not all of the recursion trees are equivalent.

LEMMA 2 *For a given value of n, the recursion trees of Alg-M, Alg-P and Alg-Z are equivalent.*

PROOF: From Theorem 3 we know that the recursion trees of Alg-M, Alg-P and Alg-Z are isomorphic as rooted trees. We must now consider the order of the calls. In Alg-P and Alg-Z, each call of degree k gives rise to calls of degree $k+1, k, \ldots, 2$, in that order. In Alg-M, each call of degree k gives rise to calls of degree $2, 3, \ldots, k+1$, in that order. Therefore, the recursion trees of Alg-M, Alg-P and Alg-Z are equivalent. □

We just showed that the algorithms that list the trees in lexicographic order have equivalent recursion trees. The two algorithms for generating trees by single rotations also have equivalent recursion trees.

LEMMA 3 *For a given value of* n*, the recursion trees of Alg-L, Alg-RR, and Alg-LRR are equivalent.*

PROOF: From Theorem 2 we know that the recursion trees of Alg-L and Alg-LRR are isomorphic as rooted trees; hence, we must only consider the order of the nodes. It is sufficient to show that, for both recursion trees and at a given level going from left to right, the children of the first node have decreasing degrees, the children of the second have increasing degrees, the children of the third have decreasing degrees, and so on, alternately increasing and decreasing.

But this follows from the descriptions of the algorithms given earlier. □

An additional algorithm which appears, on the surface, to have no relationship to the ones discussed above because its calls have degree at most two is now considered. The algorithm of Ruskey and Proskurowski [14], modified to print the bitstring representation in reverse lexicographic order, is given in Figure 12. We refer to this algorithm as Alg-RP. The initial call is `T(n+1,1,1)` and the bitstrings occur in the array `b[1..2n]`.

There is a well known "natural correspondence" between ordered forests and binary trees [4],[16]. This correspondence may be expressed as a bijection θ from the set of all binary trees with n nodes to the set of all ordered forests of n nodes. Given a node r in a binary tree T with left child s and right child t, then in the corresponding ordered forest $\theta(T)$, the node s is the leftmost child of r and t is the sibling of r immediately to the right of r. Via this correspondence, there is a one-to-one correspondence between the binary recursion tree of Alg-RP and the ordered recursion tree of Alg-Z.

THEOREM 4 *If* $\mathcal{B}$ *is the recursion tree of Alg-RP with all leaves removed and* $\mathcal{T}$ *is the recursion tree of Alg-Z, then* $\theta(\mathcal{B}) = \mathcal{T}$*.*

PROOF: In $\mathcal{B}$, a call where the second argument is $k+1$ is a left branch and a call where second argument is $k-1$ is a right branch. Refer to Figure 13.

```
procedure T( n, k, pos : integer );
begin
   if k = n then PrintIt
   else begin
      b[pos] := 1;  T( n,   k+1, pos+1 );  b[pos] := 0;
      if k > 1 then T( n-1, k-1, pos+1 );
   end;
end {of T};
```

Figure 12: Algorithm Alg-RP.

Clearly, the root of $\mathcal{T}$ has degree 2. The initial call of Alg-RP is $T(n, 1)$ which results in the call $T(n, 2)$ which in return results in the calls $T(n, 3)$ and $T(n-1, 1)$. Therefore, the root of $\mathcal{B}$ has a left child that has one right descendant, and thus the roots of the two trees are equivalent under θ. Consider an arbitrary call $T(n, k-1)$. It results in the *left* call $T(n, k)$ which then results in *right* calls in the order $T(n-1, k-1), T(n-2, k-2), ..., T(n-k+1, 1)$. So the call $T(n, k-1)$ is equivalent to a call of degree k in an ordered tree. By the above argument, each of the calls $T(n-i, k-i)$ are equivalent to calls of degree $k-i+1$, and since the degree of each of the calls is decreasing, $\theta(\mathcal{B}) = \mathcal{T}$. □

6 Conclusion

There has been extensive research in the area of generating binary trees. We summarized much recent research and showed how it is related. We also present the first loopless algorithm that generates binary trees in their usual pointer representation.

References

[1] S.W. Bent. Ranking trees generated by rotations. *Proc. 2nd Scandanavian Workshop on Algorithm Theory*, LNCS 447:132–142, 1991.

[2] R. Cole. On the dynamic finger conjecture for splay trees. *Dept. of Computer Science, NYU, Tech. Report 470, 471*, 1989.

[3] J.T. Joichi, D.E. White, and S.G. Williamson. Combinatorial Gray codes. *SIAM J. Computing*, 9:130–141, 1980.

[4] D.E. Knuth. *Fundamental Algorithms.* Addison-Wesley, 1973.

[5] D.E. Knuth. *Sorting and Searching.* Addison-Wesley, 1973.

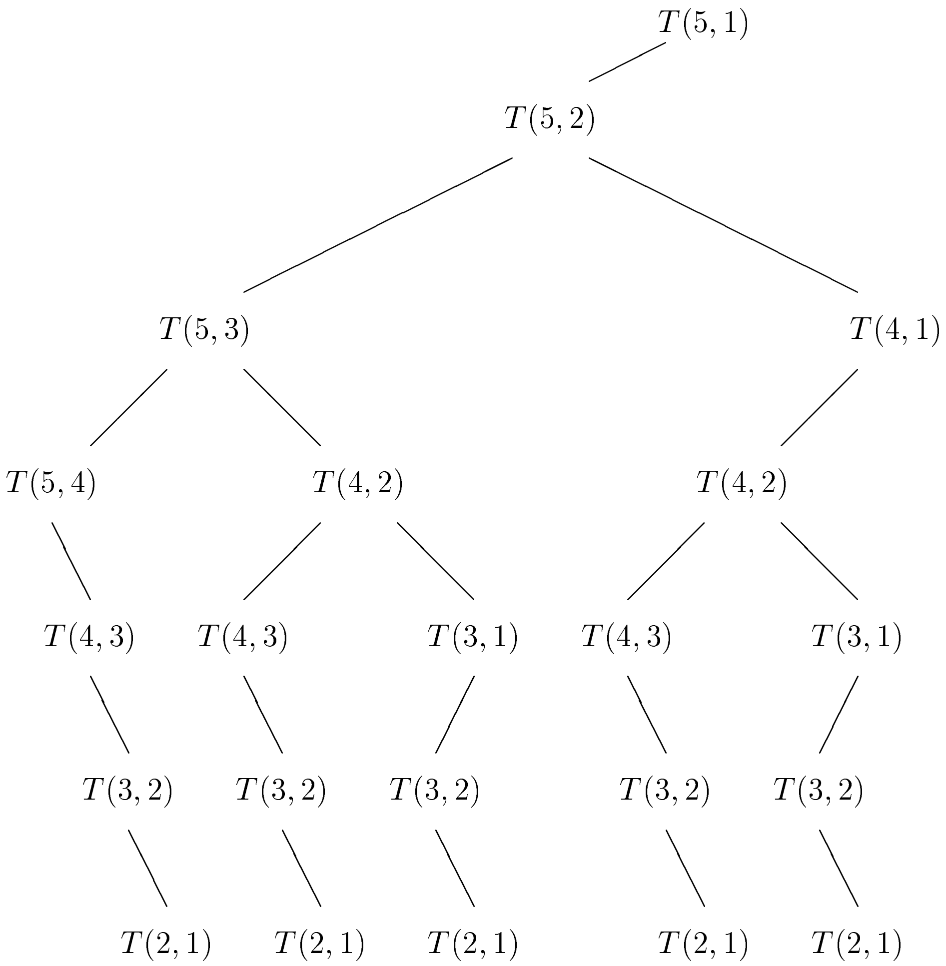


Figure 13: Recursion tree for Alg-RP.

[6] J. Lucas. Arbitrary splitting in splay trees. *Dept. of Computer Science, Rutgers Univ., Tech. Report 234*, 1988.

[7] J. Lucas. The rotation graph of binary trees is Hamiltonian. *J. Algorithms*, 9:503–535, 1988.

[8] Erkki Mäkinen. Left distance binary tree representations. *BIT*, 27:163–169, 1987.

[9] A. Nijenhuis and H.S. Wilf. *Combinatorial Algorithms, 2nd Ed.* Academic Press, 1978.

[10] J. Pallo. Enumerating, ranking, and unranking binary trees. *Computer J.*, 29:171–175, 1986.

[11] J. Pallo. Some properties of the rotation lattice of binary trees. *Computer J.*, 31:564–565, 1988.

[12] A. Proskurowski and F. Ruskey. Binary tree Gray codes. *J. Algorithms*, 6:225–238, 1985.

[13] F. Ruskey and T.C. Hu. Generating binary trees lexicographically. *SIAM J. Computing*, 6:745–758, 1977.

[14] F. Ruskey and A. Proskurowski. Generating binary trees by transpositions. *J. Algorithms*, 11:68–84, 1990.

[15] D.D. Sleator, R.E. Tarjan, and W.P. Thurston. Rotation distance, triangulations, and hyperbolic geometry. *18th ACM Symposium on Theory of Computing*, pages 122–135, 1986.

[16] Thomas A. Standish. *Data Structure Techniques.* Addison Wesley, 1980.

[17] R. Sundar. Twists, turns, cascades, deque conjecture, and scanning theorem. *30th IEEE Symposium on Foundatins of Computer Science*, pages 555–559, 1989.

[18] R. E. Tarjan. Sequential access in splay trees takes linear time. *Combinatorica*, 5:367–378, 1985.

[19] D. Roelants van Baronaigien. A loopless algorithm for generating binary tree sequences. *Info. Proc. Lett.*, 39:189–194, 1991.

[20] D. Roelants van Baronaigien and F. Ruskey. A Hamilton path in the rotation lattice of binary trees. *Congressus Numerantium*, 59:313–318, 1987.

[21] D. Roelants van Baronaigien and F. Ruskey. Generating t-ary trees in A-order. *Info. Processing Letters*, 27:205–213, 1988.

[22] S. Zaks. Lexicographic generation of ordered trees. *Theoretical Computer Science*, 10:63–82, 1980.

[23] D. Zerling. Generating binary trees using rotations. *JACM*, 32:694–701, 1985.

A Loopfree Generation Algorithm

The following is a complete listing of an implementation of our $O(1)$ generation algorithm.

```
program generate(input,output);
const
   treesize = 30;
   UP = true;
   DOWN = false;
type
   ptr = ^cell;
   cell = record
             left,right,parent: ptr;
             symsucc,next: ptr;
             direct: boolean; { direction }
          end;
var
   tree: ptr;  { root of the tree }
   uhead: ptr; { first node of unfinished node }
   fhead: ptr; { first node of finished nodes  }
   x,c,n: integer;

function DoneInDirection(t:ptr) : boolean;
begin {** assumes short circuit and,or **}
DoneInDirection :=
  (( t^.direct = DOWN) and (t^.left = nil )) or
  (( t^.direct = UP) and ( t^.parent = nil)) or
  (( t^.direct = UP) and ( t^.parent^.left = t ));
end {DoneInDirection};
```

```
procedure RotateLeft(t:ptr);
var
   p : ptr;   { parent of t }
   gp : ptr;  { grandparent of t }
   c : ptr;   { child of t whose parent changes }
begin
   p := t^.parent;
   gp := p^.parent;
   c := t^.left;

   if gp <> nil then
      if gp^.right = p then gp^.right := t
      else gp^.left := t
   else tree := t;

   p^.right := c;
   p^.parent := t;
   t^.left := p;
   t^.parent := gp;
   if c <> nil then c^.parent := p;
end {of RotateLeft};

procedure RotateRight(t:ptr);
var
   p : ptr;   { parent of t }
   lc : ptr;  { left child of t }
   gc : ptr;  { grandchild whose parent changes }
begin
   p := t^.parent;
   lc := t^.left;
   gc := lc^.right;

   if p <> nil then
      if p^.right = t then p^.right := lc
      else p^.left := lc
   else tree := lc;

   lc^.right := t;
   lc^.parent := p;
   t^.left := gc;
   t^.parent := lc;
   if gc <> nil then gc^.parent := t;
end {of RotateRight};
```

```
procedure PrintIt( t : ptr );
begin {** details left to reader **}
end {of PrintIt};

procedure CreateNextTree;
var current: ptr;
begin
  current := uhead; { Next node to be rotated }

  { Nodes > current are added to unfinished list }
  if uhead^.symsucc <> nil then begin
     uhead := fhead;
     fhead := current^.symsucc^.next;
     current^.symsucc^.next := current;
  end;

  { Rotate current node in appropriate direction }
  if current^.direct = UP then RotateLeft(current)
  else RotateRight(current);

  PrintIt( tree );

  { If current node has finished rotating in its }
  { direction, move it to the front of the list  }
  { of finished nodes and change its direction.  }
  if DoneInDirection( current ) then begin
     if current = uhead then uhead := current^.next
     else current^.symsucc^.next := current^.next;
     current^.next := fhead;
     fhead := current;
     current^.direct := not current^.direct;
  end;
end {of CreateNextTree};
```

```
{Initial tree consists of a right path of n nodes.}
procedure Initialize;
var
   i: integer;
   s: ptr;
begin
   new(tree);
   tree^.parent := nil;
   tree^.left := nil;
   tree^.direct := UP;
   tree^.next := nil;
   s := tree;
   for i := 2 to n do begin
      new(s^.right);
      s^.symsucc := s^.right;
      s^.right^.parent := s;
      s^.right^.next := s;
      s := s^.right;
      s^.left := nil;
      s^.direct := UP;
   end;
   s^.right := nil;
   s^.symsucc := nil;
   {Unfinished list consists of all but first node}
   fhead := tree;
   uhead := s;
   tree^.right^.next := nil;
end {of Initialize};

begin
   write('enter n -> ');  readln(n);
   Initialize;
   PrintIt( tree );
   while uhead <> nil do CreateNextTree;
end.
```