Ordered Tree Generation Algorithms

By

Dominique Roelants van Baronaigien

B Sc , University of Calgary, 1980

B Sc , University of Victoria, 1982

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the Department of
Computer Science

We accept this thesis as conforming
to the required standard

[redacted]

Supervisor Dr Frank Ruskey

[redacted]

Dr. John Ellis

[redacted]

Dr Ernie Cockayne

[redacted]

Dr. Donald J. Miller

UNIVERSITY OF VICTORIA

May 1986

Supervisor Dr. Frank Ruskey

ABSTRACT

Algorithms have been created for generating all binary trees with n nodes, all k-ary trees with n nodes all ordered trees with n nodes, and all ordered trees with a particular degree sequence. A survey of these ordered tree generation algorithms is presented here.

Two natural orderings [25] are described for listing k-ary trees. These orderings are called A-order and B-order. Many algorithms already exist for generating k-ary trees in B-order, but no algorithm exists for generating k-ary trees in A-order.

This thesis provides an algorithm for generating k-ary trees in A-order. The algorithm is shown to be, on average, constant time per tree generated for $k=2$. The thesis also provides the ranking and unranking algorithms associated with A-order.

Examiners

[redacted]

Dr. Frank Ruskey

[redacted]

Dr. John Ellis

[redacted]

Dr. Ernie Cockayne

[redacted]

Dr. Donald J. Miller

TABLE OF CONTENTS

TABLE OF FIGURES

CHAPTER 1

Introduction

Many authors have commented on the importance of studying trees. Read [10] stated that examining lists of combinatorial objects, such as trees, would be useful for generating hypotheses about the properties of the objects. Proskurowski [8] suggested that generation algorithms might be useful in creating random data or exhaustive data for testing or analyzing programs. Arnold and Sleep [1] have suggested that the generation of balanced parenthesis strings (which are in one to one correspondence with binary trees) is useful to create sample test data for programming language error recovery. Williamson [23] has stated that the creation of listing algorithms is useful in developing an understanding of natural orderings of the combinatorial objects being listed.

Much work has been done in the area of generation algorithms for ordered trees of size N [5,7,8,10,12,14,15,16,17,18,20,24,25,26,27]. Algorithms have been produced which generate all binary trees [7,8,12,14,15,18,21,27], all k-ary trees [16,20,25,26] and ordered trees with particular properties [17,24].

This thesis discusses the problem of generation algorithms for ordered trees. An algorithm for generating trees will induce an ordering on those trees, and thus it is of interest to be able to define algorithms for determining the position of a tree in the list created by the generating algorithm or, constructing a tree with a given position. We call these algorithms ranking and unranking algorithms respectively.

There are many algorithms for generating binary trees and since there is a one to one correspondence between binary trees and ordered trees, there are many algorithms that generate all ordered trees. Some of the results have been extended to k-ary trees, and there are several algorithms for generating all ordered trees with a particular degree sequence. This thesis will review the known algorithms and it will also add a new algorithm.

Zaks [25], defines two *natural orderings*, A-order and B-order. Most of the known algorithms generate the trees in B-order. He states that, although there is an algorithm that generates binary trees in A-order, there is no known algorithm that generates all k-ary trees in A-order. This thesis will provide an algorithm for generating all k-ary trees in A-order as well as providing the ranking and unranking algorithms associated with A-order. The generating algorithm is constant average time per tree generated when $k=2$, and it all most certainly seems to be constant average time per tree generated for any other values of k.

Most of the algorithms generate a set of combinatorial objects that is equivalent to the set of trees, and thus if they generate all of the combinatorial objects, they generate all of the trees. Some of the objects being created include ballot sequences [14], bit strings with particular properties [7,8,10,12,25,26], set partitions with particular properties [21], and integer sequences with certain restrictions [14,15,16,17,20,26,27]. A review of the different representations used is given in chapter 2.

In chapter 3 we discuss the generating algorithms for ordered trees. The tree generation algorithms discussed in this thesis fall into three categories. Many of the algorithms [7,8,12,14,15,18,23,27] just generate binary trees. Some of the algorithms generate k-ary trees [16,21,25,26], and of course, binary trees are a special case of k-ary trees. Read [10] created an algorithm for generating all ordered trees, and Zaks and Richards [24] and Ruskey and Roelants van Baronaigien [17] have created algorithms for generating all ordered trees with a particular degree sequence.

A new algorithm for generating all k-ary trees in A-order is presented in chapter 4. The equivalence between A-order for k-ary trees and lexicographic order for a certain class of integer sequences is shown. We also give algorithms for ranking and unranking.

In chapter 5 we discuss the time complexity of the generating, ranking and unranking algorithms presented in chapter 4.

Chapter 6 summarizes the work done in the area of tree generation including the new results given in chapter 4, and gives some direction for possible further study.

CHAPTER 2

Definitions and Representations

In this chapter we will discuss different representations of ordered, binary and k-ary trees. We will also discuss the relationship between the representations and walks through a lattice with particular properties.

Definitions

The study of tree generation algorithms requires some definitions from graph theory.

Definition 2.01: A *graph* G consists of a finite nonempty set $V=V(G)$ of n nodes and a set E of edges where each edge is an unordered pair $\{u,v\}$ where $u,v \in V$. Let $\{u,v\} \in E$ then the edge $\{u,v\}$ is said to *join* the nodes u and v. An edge $\{v,v\}$ is called a *loop*.

We will assume that any reference to graphs will mean a *graph* G with no loops and no multiple edges.

Definition 2.02: A *path* is a sequence of nodes and edges $v_0,e_1,v_1,e_2,v_2,\ldots,e_n,v_n$ such that all edges are distinct, and the edge e_i joins the nodes v_{i-1} and v_i. A path is said to be *simple* if all of the nodes are distinct. A *cycle* is a path $v_0,e_1,v_1,e_2,v_2,\ldots,e_n,v_n$ such that all nodes except v_0 and v_n are distinct and $v_0=v_n$.

Definition 2.03: A graph is said to be *connected* if there is at least one simple path between any node and any other node.

Definition 2.04: A *free tree*, is a connected graph with no cycles.

Definition 2.05: A *rooted tree* is a free tree in which one node $r \in V$ is designated as the root. If $u, v \in V, u \neq v$ and there is a path from u to v that does not include r, and the path from r to v includes the node u then v is said to be a *descendant* of u, u is also said to be an *ancestor* of v. If u is an ancestor of v and there is an edge joining u and v then we say that u is the *parent* of v and that v is the *child* of u. If $u, v \in V$ and u and v have the same parent, then we say that u and v are *siblings*. The *degree* of a node $v \in V$ is the number of distinct children of v in T. A node with degree zero is called a *leaf* and a node with a higher degree is called an *internal node*. A *subtree* rooted at v consists of the node v and all its descendents together with their associated edges. The *level* of a node v is the number of edges on the path between the root of the tree and the node v.

Definition 2.06: The *weight* of node $v \in V$ is the number of internal nodes in the subtree rooted at v.

An alternate definition for weight is that the weight of a node v is zero if the node is a leaf, otherwise the weight is 1 plus the sum of the weights of all of the children of node v.

The definitions of sibling, parent, child, degree, weight, leaf, internal node, subtree, ancestor, descendant and level described above also apply to the types of trees defined below.

Definition 2.07: An *ordered tree*, T, is a rooted tree in which the n subtrees rooted at the children of the root r are numbered $T_1, T_2, \ldots, T_n$, and $T_1, T_2, \ldots, T_n$ are each ordered trees. The trees $T_i, 1 \leq i \leq n$ are said to be the *principle subtrees* of T and T_i is said to be to the *right* of T_j if $i > j$. The root of T_i is said to be a *right sibling* of the root of T_j. Left sibling is similarly defined.

Definition 2.08: A *binary tree* T is a set of nodes V, possibly empty, such that, if $V \neq \phi$, one node $r \in V$ is designated as the root and $V - \{r\}$ is partitioned into two disjoint subsets called the *left* and the *right* subtrees.

Definition 2.09: A *full binary tree* is a binary tree such that all nodes have degree two or degree zero. Full binary trees are also called *extended* binary trees.

Definition 2.10: A *k-ary tree* is a set of nodes V, possibly empty, such that, if $V \neq \phi$, one node $r \in V$ is designated as the root and $V - \{r\}$ is partitioned into k disjoint subsets $V_1, V_2, \ldots, V_k$, each of which is a k-ary tree. $V_i, 1 \leq i \leq k$ is said to be the i th subtree of r. V_i is said to be to the *left* of V_j if $i < j$.

Definition 2.11: A *full k-ary tree* is a k-ary tree such that any node in T has degree 0 or degree k. Full k-ary trees are also called *extended* k-ary trees.

It should be noted that full k-ary trees are ordered trees in which every node has degree 0 or degree k. There is a well known correspondence between k-ary trees and full k-ary trees. A full k-ary tree, T, is equivalent to the k-ary tree generated by removing all the leaves from T.

Definition 2.12: A sequence $S = s_0, s_1, \ldots, s_n$ is said to be the *degree sequence* of a tree T if for all i, $0 \leq i \leq n$, there are s_i nodes that have degree i, and $s_n \neq 0$.

The following definition gives a linear ordering on the set of all k-ary trees. It is referred to as *A-order*. Algorithms by Knott [5] and Soloman and Finkel [18] generate trees in A-order.

Definition 2.13:

Given two k-ary trees T and T', we say that $T < T'$ if

(1) $|T| < |T'|$ or

(2) $|T| = |T'|$, and for some $1 \leq i \leq k$ we have

(a) $T_j = T_j'$ for $j = 1,2,3,\ldots,i-1$, and

(b) $T_i < T_i'$,

where $|T|$ is the number of internal nodes in the tree T.

An alternative linear ordering is given by Zaks [25]. This definition is called *B-order*. Most of the tree generation algorithms [7,8,12,15,16,20,25] list trees in B-order.

Definition 2.14:

Given two k-ary trees T and T', we say that $T < T'$ if

(1) T is empty and T' is not, or

(2) T is not empty, and for some i, $1 \leq i \leq k$, we have

(a) $T_j = T_j'$ for $j = 1,2,3,\ldots,i-1$, and

(b) $T_i < T_i'$.

Zaks [25], when defining these orderings, states that no algorithm exists that generates k-ary trees in A-order. He stated that all direct algorithms known at the time [16,20,25] generated k-ary trees in B-order. Not all algorithms generate binary or k-ary trees in either A-order or B-order. Algorithms by Zerling [27], and Rotem and Varol [14] generate binary trees and Zaks algorithm [26] generates k-ary trees in some order distinct from A-order or B-order.

Almost all of the generation algorithms generate sequences, in lexicographic order, that represent trees, and thus it is necessary to define what is meant by a lexicographic ordering of sequences.

Definition 2.15: Given two sequences of integers, $S=s_1,s_2,\ldots,s_n$ and $S'=s_1',s_2',\ldots,s_m'$, we say that $S<S'$ in *lexicographic order* if i, $1\leq i\leq min(m,n)$ is the smallest integer such that $s_j=s_j'$ for all $1\leq j<i$ and $s_i<s_i'$ or $i=n<m$ and $s_j=s_j'$ for all $j\leq i$.

Although this definition describes lexicographic order for most of the sequence representations used in the tree generation algorithms, Rotem and Varol [14] define lexicographic ordering by making the item that occurs last in the sequence the most significant item. As an example, the sequence 1,2,3,4 > 1,1,4,4 according to definition 2.15, but 1,2,3,4 < 1,1,4,4 according to the order described by Rotem and Varol.

Now that we have defined what is meant by an ordering on a set of trees we can give a more formal definition of the *rank* of a tree.

Definition 2.16: Let $<$ be an ordering on a set of trees. The *rank* of a tree T is the number of distinct trees T' such that $T'<T$ and $|T|=|T'|$.

Representations

The most traditional representation used is the linked representation [18]. A binary tree is represented by a node that has two links, a left link and a right link.

Other representations used in the generation algorithms are set partitions, bit sequences, or integer sequences. The first algorithm written to generate binary trees used set partitions as the representation.

Set partitions

Wells [21] represents full binary trees with n internal nodes by partitions of the set $\{0,1,2,\ldots,2n-1\}$ with the following properties.

Let $V_1, V_2, \ldots, V_r$ be a partition of the set $\{0,1,2,\ldots,2n-1\}$, then the set partition represents a full binary if

(1) $V_i \subseteq \{0,2,4,\ldots 2n-2\}$ or $V_i \subseteq \{1,3,5,\ldots,2n-1\}$ for all i, $1 \leq i \leq r$, and

(2) that there is no pair of subsets V_i and V_j such that there exists values v_1, v_2, v_3, v_4, $v_1, v_2 \in V_i$ and $v_3, v_4 \in V_j$ and $v_1 > v_3 > v_2 > v_4$, and

(3) r is the smallest integer such that the first two conditions are true.

To obtain such a set partition from a full binary tree, edges are labeled such that left falling links are labeled with even numbers and right falling links are labeled with odd numbers. The order defined for labeling the edges is not in common use. The following algorithm describes how to label the edges.

Algorithm 2.01

```
procedure label(tree)
if tree ≠ leaf then
    initialize stack
    label left edge with a 0
    push(tree)
    tree ← left(tree)
    c ← 1
    while not empty do
        if tree = leaf then
            pop(tree)
        else
            push(tree)
        if odd(c ) then
            label right edge with c
            tree ← right(tree)
        else
            label left edge with c
            tree ← left(tree)
        c ← c + 1
```

The labels 0,1,2, ...,$2n-1$ are used. The set of integers $\{0,1,2,\ldots,2n-1\}$ is split into partitions using the following equivalence relation: Let u_1 and v_1 be nodes incident with the edge e_1, and let u_2 and v_2 be nodes incident with e_2, then $e_1 \equiv e_2$ if and only if there is a path through the nodes u_1, v_1, u_2, v_2, in any order, such that all of the edges on that path are either all left links or all right links. It can be observed that this relation will not group any odd numbers with any even numbers.

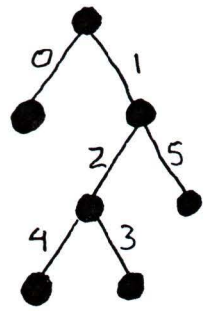

Figure 2.01
The binary tree represented by the set partition { {0} {1,5} {2,4} {3} }

1 {0,2,4} {1} {3} {5}
2 {0,4} {1,3} {2} {5}
3 {0,2} {1} {3,5} {4}
4 {0} {1,5} {2,4} {3}
5 {0} {1,3,5} {2} {4}

Figure 2.02
The set partitions for all full binary trees with 3 internal nodes.

This representation takes $O(n^2)$ space. Most algorithms either require $O(n)$ or $O(n \log n)$ space. This representation is not used by any other authors. The most common representation is the preorder bit sequence representation.

1	101010100	1,3,5,7	8	110100100	1,2,4,7
2	101011000	1,3,5,6	9	110101000	1,2,4,6
3	101100100	1,3,4,7	10	110110000	1,2,4,5
4	101101000	1,3,4,6	11	111000100	1,2,3,7
5	101110000	1,3,4,5	12	111001000	1,2,3,6
6	110010100	1,2,5,7	13	111010000	1,2,3,5
7	110011000	1,2,5,6	14	111100000	1,2,3,4

Figure 2.04
A table of the preorder bit sequences, in lexicographic order, and the position sequences in reverse lexicographic order, representing all binary trees with 4 internal nodes.

A slight variation of this representation occurs in Zaks [26] where the labels are listed in postorder. The conditions for a *postorder bit sequence* to represent a k-ary tree a similar to those described above. Again, Zaks translates the bit sequence to a sequence of positions.

Read [10] also uses bit sequences but his *bit codes* represent ordered trees. A bit code is generated by traversing the ordered tree in preorder and writing a zero every time an edge is traversed in a direction away from the root and writing a one every time an edge is traversed in a direction toward the root. This correspondence between the bit codes and ordered trees was proven by De Bruijn and Morselt [3]. They showed that a bit code $S = s_1, s_2, \ldots, s_{2n}$ represents an ordered tree if and only if $\sum_{i=1}^{2n} s_i = n$ and $\sum_{i=1}^{j} 2s_i \leq j$ for all $j < 2n$.

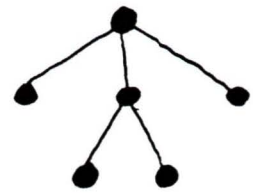

Figure 2.05
An ordered tree represented by the bit code 0100101101.

1:	00001111	8:	00110011
2:	00010111	9:	00110101
3:	00011011	10:	01000111
4:	00011101	11:	01001011
5:	00100111	12:	01001101
6:	00101011	13:	01010011
7:	00101101	14:	01010101

Figure 2.06
A list of the bit codes representing all ordered trees with five nodes.

The algorithms of Ruskey and Roelants van Baronaigien [17] and Zaks [24] use another integer sequence to represent ordered trees. Instead of labeling each internal node with a 1, each internal node is labeled with its degree.

Preorder enumeration of degrees

A sequence S is a *preorder enumeration of degrees* if there is an ordered tree such that if you labeled each node with its degree, and then read the labels off in preorder, you would obtain the sequence S. A sequence $S = s_1, s_2, \ldots, s_N$ is a preorder enumeration of degrees if and only if $\sum_{i=1}^{N} s_i = N-1$ and $\sum_{i=1}^{j} s_i \geq j$ for all $j < N$. The correspondence was shown by Zaks and Richards [24] although they again left the last zero off of the end of their sequence. The algorithm of Ruskey and Roelants van Baronaigien also drops the last zero from the representation. This is done because the last element in any preorder enumeration of degrees will be an element that represents a leaf.

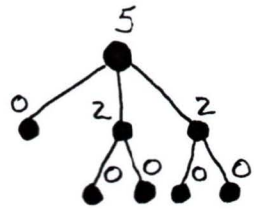

Figure 2.07
An ordered tree with degree sequence 5,0,2,1 and represented by the preorder enumeration of degrees 3,0,2,0,0,2,0,0

1 2,0,2,0,3,0,0,0	8 2,2,3,0,0,0,0,0	15 3,0,2,0,0,2,0,0
2 2,0,2,3,0,0,0,0	9 2,3,0,0,0,2,0,0	16 3,0,2,0,2,0,0,0
3 2,0,3,0,0,2,0,0	10 2,3,0,0,2,0,0,0	17 3,0,2,2,0,0,0,0
4 2,0,3,0,2,0,0,0	11 2,3,0,2,0,0,0,0	18 3,2,0,0,0,2,0,0
5 2,0,3,2,0,0,0,0	12 2,3,2,0,0,0,0,0	19 3,2,0,0,2,0,0,0
6 2,2,0,0,3,0,0,0	13 3,0,0,2,0,2,0,0	20 3,2,0,2,0,0,0,0
7 2,2,0,3,0,0,0,0	14 3,0,0,2,2,0,0,0	21 3,2,2,0,0,0,0,0

Figure 2.08
The preorder enumerations of degrees representing all ordered trees with degree sequence (5,0,2,1)

Standish [19] gives another method of representing trees, similar to the preorder enumeration of degrees. This method involves labeling each node with its weight and then enumerating the labels in preorder. This representation has not been used in any of the known algorithms. Standish does not prove the equivalence between these sequences and ordered trees. This will be done in chapter 4.

Another common representation [5,14,20] involves labeling the nodes in one order and then enumerating the labels in a different order.

Integer sequences

Knott [5], uses a representation that involves labeling the nodes in inorder and then enumerating them in preorder. He defines this representation as a *tree permutations*. Let $S = s_1, s_2, \ldots, s_n$ be a sequence and let $j = s_1$ then S is a tree permutation if and only if the subsequence $s_2, s_3, \ldots, s_j$ is a tree permutation, the subsequence $s_{j+1} - s_1, s_{j+2} - s_1, \ldots, s_n - s_1$ is a tree permutation, $s_i < s_1$ for all i, $1 < i < j$ and $s_i > s_1$ for all i, $j < i \leq n$.

Rotem and Varol [14] go further. After having generated an initial sequence, $S = s_1, s_2, \ldots, s_n$ by labeling the nodes of the binary tree in inorder and enumerating them in preorder, they then translate S into a *ballot sequence* S' as follows. Let $s_j = i$ then s_i' is the number of elements in the subsequence $s_{j+1}, s_{j+2}, \ldots, s_n$ that are greater than i.

This construction guarantees that the sequence S' is in descending order. Rotem and Varol [14] show that there is a one to one correspondence between ballot sequences and binary trees.

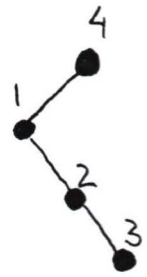

Figure 2.09
The binary tree with 4 nodes that is represented by the ballot sequence 2,1,0,0 and tree permutation 4,1,2,3.

Trojanowski's representation differs slightly from the two previous representations. Trojanowski labels the nodes in preorder and then enumerates them in inorder. Trojanowski then created a set of sequences $\mathbf{P}(n)$. $\mathbf{P}(n)$ is defined recursively as follows.

(1) $\mathbf{P}(1) = 1$

(2) if $n > 1$ then $S = s_1, s_2, \ldots, s_n \in \mathbf{P}(n)$
if and only if there exists a j, $1 \leq j \leq n$ such that the subsequence $s_1, s_2, \ldots, s_{j-1}, s_{j+1}, \ldots, s_n \in \mathbf{P}(n-1)$ and $s_j = n$ and $S_k = n-1$ implies $j \geq k-1$

These two conditions are equivalent to stating that if $S = s_1, s_2, \ldots, s_n$ then if there is an $s_i > s_j, i < j$ such that there is an $s_k < s_j, i < k < j$ then S is not an element of $\mathbf{P}(n)$. He then describes a recursive, "direct insertion" order for the sequences in $\mathbf{P}(n)$ and proves that the set $\mathbf{P}(n)$ is equivalent to the set of binary trees with n nodes.

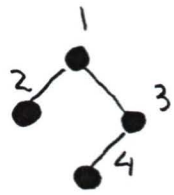

Figure 2.10
The binary tree with 4 nodes that is represented by 2,1,4,3 in $\mathbf{P}(4)$

Although no generation algorithm is given by Knott [5], the ordering induced by Knott's tree permutations, and the orderings induced by Trojanowski's [20] $\mathbf{P}(n)$ and Rotem and Varol's [14] ballot sequences are all based on the same initial representation method, and yet they list the binary trees in different orders.

	Trojanowski[20]	Rotem[14]	Knott[5]
1	1,2,3,4	3,2,1,0	1,2,3,4
2	1,2,4,3	3,2,0,0	1,2,4,3
3	1,3,2,4	3,1,1,0	1,3,2,4
4	1,3,4,2	3,1,0,0	1,4,2,3
5	1,4,3,2	3,0,0,0	1,4,3,2
6	2,1,3,4	2,2,1,0	2,1,3,4
7	2,1,4,3	2,2,0,0	2,1,4,3
8	2,3,1,4	2,1,1,0	3,1,2,4
9	2,3,4,1	2,1,0,0	4,1,2,3
10	2,4,3,1	2,0,0,0	4,1,3,2
11	3,2,1,4	1,1,1,0	3,2,1,4
12	3,2,4,1	1,1,0,0	4,2,1,3
13	3,4,2,1	1,0,0,0	4,3,1,2
14	4,3,2,1	0,0,0,0	4,3,2,1

Figure 2 11
All sequences in $\mathbf{P}(4)$, all 4 element ballot sequences and all tree permutations. The sequences in $\mathbf{P}(4)$ are in lexicographic order.

It is easy to see that the orderings of Trojanowski [20] and Knott [5] are different. Since Rotem and Varol [14] defined lexicographic by stating that the element that occurred last in the ballot sequence was the most significant, clearly Rotem and Varol generated trees in an order different from Trojanowski.

It should be noted that although the ballot sequences used by Rotem and Varol [14] are generated from the sequences used by Knott [5], they could just as easily been derived from the sequences used by Trojanowski [20]. If $S = s_1, s_2, \ldots, s_n$ is an element of $\mathbf{P}(n)$ then the sequence

$S' = s_1', s_1', \ldots, s_n'$, where s_i' is the number of elements of S that occur after position i that are greater than the element in position i, is a ballot sequence.

All of the preceding representations were based on either a preorder or an inorder labeling of all the nodes or of all the edges. The following representations do not label all of the nodes of a tree.

Feasible sequences

The algorithms by Ruskey and Hu [15] and Ruskey [16] use a *feasible sequence* representation of extended binary and k-ary trees. This representation differs from the preceding ones in that it is not based on labeling either all edges or all nodes. A *feasible sequence* is any sequence generated by labeling every leaf in a full k-ary tree T with its level and then enumerating the labels from left to right.

Ruskey and Hu [15] shows that if a sequence $S = s_1, s_2, \ldots s_N$ is feasible then $\sum_{i=1}^{N} 2^{-s_i} = 1$. Ruskey [16] proves an similar formula for k-ary trees.

Let $S = s_1, s_2, \ldots, s_N$ be a sequence where j is the smallest integer such that $s_{j-1} = s_j$, Ruskey and Hu [15] define a reduction from the left (for binary trees) as the replacement of the the pair s_{j-1}, s_j with the single integer $s_j - 1$. A reduction from the right is defined similarly. Ruskey [16] defines reduction from the left and right for k-ary trees as well.

The authors [15,16] show that a sequence is feasible if and only if a series of reductions from the left reduce the original sequence to the single integer 0.

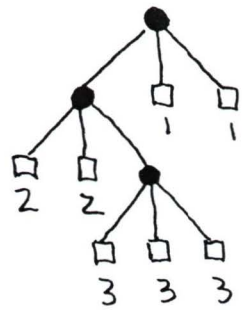

Figure 2.12
The 3-ary tree with seven leaves represented by the feasible sequence 2,2,3,3,3,1,1

1	1,1,2,2,3,3,3	7	1,3,3,3,2,2,1
2	1,1,2,3,3,3,2	8	2,2,2,1,2,2,2
3	1,1,3,3,3,2,2	9	2,2,2,2,2,2,1
4	1,2,2,2,2,2,2	10	2,2,3,3,3,1,1
5	1,2,2,3,3,3,1	11	2,3,3,3,2,1,1
6	1,2,3,3,3,2,1	12	3,3,3,2,2,1,1

Figure 2.13
A list of the sequences representing all 3-ary trees with seven leaves.

The last representation discussed in this chapter is a representation based on tree rotations.

Code words

Zerling [27] uses code words to represent binary trees. The integers in the code word describe how to construct the binary tree from a binary tree that is structured such that no node has a right child. The integer code

represents the number of single right rotations to perform on any given subtree.

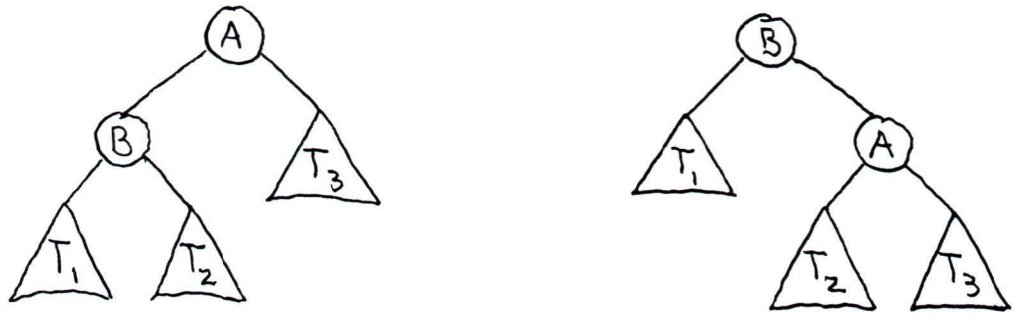

Figure 2.14
A single right rotation

A binary tree with n nodes would be represented by a sequence of integers $s_{n-1}, s_{n-2}, \ldots, s_1$ where $\sum_{i=1}^{n-1} s_i < n-1$. If you label the nodes in the initial binary tree in preorder, then each s_i represents the number of single right rotations to perform on the subtree rooted at node i.

Zerling [27] shows the correspondence between his code words of length $n-1$ and binary trees with n nodes. He then goes on to prove that a sequence $S = s_{n-1}, s_{n-2}, \ldots s_1$ is a valid code word if for $i = n-1, n-2, \ldots, 1$, $0 < s_i < n - i - \sum_{j=i+1}^{n-1} s_j$

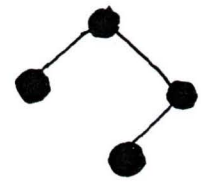

Figure 2.15
The binary tree with four nodes represented by the code word 0,1,1

1	0,0,0	8	0,2,0
2	0,0,1	9	0,2,1
3	0,0,2	10	1,0,0
4	0,0,3	11	1,0,1
5	0,1,0	12	1,0,2
6	0,1,1	13	1,1,0
7	0,1,2	14	1,1,1

Figure 2.16
A list of the code words representing all binary trees with four nodes listed in lexicographic order.

Lattice Paths

Let $S=s_0,s_1,\ldots,s_n$, be a degree sequence for some collection of trees. Let $k_i=j$ where s_j is the $i+1st$ non zero entry in S. Let $L=\{(x_0,x_1,\ldots,x_t)\ x_i\geq 0$ integers, $x_0\geq \sum_{i=1}^{t}(k_i-1)x_i\}$. If we consider walks

along the lattice L that are restricted to moving toward the origin, then these walks along the lattice path are in one to one correspondence to some of the tree representations given earlier in this chapter

In the case of k-ary trees, the above formula for the lattice simplifies to $L = \{(x_0, x_1) : x_0, x_1 \geq 0 \text{ integers}, x_0 \geq (k-1)x_1\}$ Ruskey [16] shows that his feasible sequence representation of k-ary trees is in one to one correspondence with walks in the the lattice L

Zaks and Richards [24] also show that their representation of ordered trees with a particular degree sequence is equivalent to walks in the lattice L The representation used by Ruskey and Roelants van Baronaigien [17] is the same as the one used by Zaks and Richards [24]

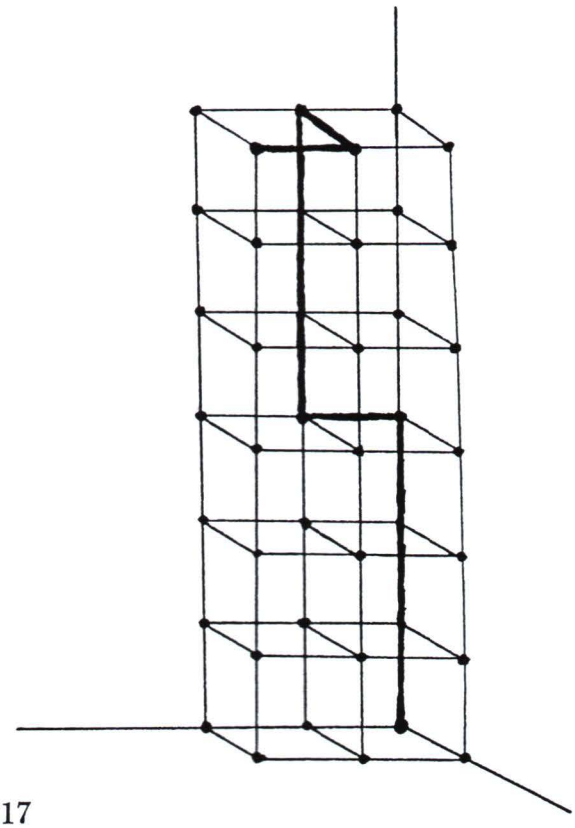

Figure 2.17
The lattice path corresponding to the ordered tree with preorder enumeration of degrees 3,2,0,0,0,3,0,0,0

Zaks [26] also show that the bit sequence representation of k-ary trees is in one to one correspondence to walks through a lattice. Although Zaks [26] used a postorder traversal for his representation, the preorder bit sequences used by Proskurowski [7], Proskurowski and Laiman [8], Read [12] and Zaks [25] are also equivalent to walks on a lattice.

To go from a preorder bit sequence $S=s_1,s_2,\ldots,s_{nk}$ to a walk through the lattice L, each step would be from (x_0,x_1) to $(x_0-(1-s_i),x_1-s_i)$ for $i=1,2,3,\ldots,kn$. The last step, since the trailing zero is left off in some of the bit sequences, would be (1,0) to (0,0). It is easily seen how one would go from the lattice path to the preorder bit sequence.

The representations used by Knott [5], Trojanowski [20] and Rotem and Varol [14] are also in one to one correspondence to walks on the lattice L. The representation of Rotem and Varol [14] can be translated into a lattice path as follows. Let $S=s_1,s_2,\ldots,s_n$ be a ballot sequence, then the lattice path described by the ballot sequence S is $(n,n)\rightarrow(n,s_1)\rightarrow(n-1,s_1)\rightarrow(n-1,s_2)\rightarrow\cdots\rightarrow(1,s_{n-1})\rightarrow(1,s_n)\rightarrow(0,0)$. Again, it is simple to determine the ballot sequence from the lattice path.

As described earlier, the representations used by Trojanowski [20] and Knott [5] can be translated into the one used by Rotem and Varol [14] and thus those representations are in one to one correspondence to walks through the lattice L.

Zerling [27] shows that his representation is also equivalent to walks through the lattice path L. Let $S=s_1,s_2,\ldots,s_{n-1}$ be a code word. Let $t_n=0$, we define $t_j=t_{j+1}+s_j$ for $j=n-1,n-2,\ldots,2,1$. The lattice path defined by the code word S is $(n,n)\rightarrow(n,t_1)\rightarrow(n-1,t_1)\rightarrow(n-1,t_2)\rightarrow\cdots\rightarrow(1,t_{n-1})\rightarrow(1,t_n)\rightarrow(0,0)$. This definition is very similar to the one used to describe the lattice path generated from Rotem and Varol's [14] representation. The code word is easily determined from the lattice path.

CHAPTER 3

Tree Generation Algorithms

Introduction

There have been many algorithms developed for generating binary trees [5,7,12,14,15,18,21,25,27] Some of the algorithms have been extended to k-ary trees [16,20,25,26] Read's algorithm [10] generates ordered trees Two algorithms have been developed to generate all ordered trees with a particular degree sequence[17,24]

Representation

All but one of the algorithms presented here represent the tree as a sequence of some type Each algorithm, thus, generates trees in some order that is affected by the representation Algorithms by Proskurowski [7], Proskurowski and Laiman [8], Read [12], Ruskey and Hu [15], Ruskey [16], Trojanowski [20], and Zaks [25] generate trees in B-order Most of these algorithms generate bit sequences which are in one to one correspondence with the trees that they represent Ruskey and Hu [15] and Ruskey [16] represent trees with feasible sequences Zaks [25] shows that lexicographic order for the sequences generated by the algorithms in [15,20,25], even though the representations are not the same, induces the same order on the trees that these sequences represent

The other natural ordering defined by Zaks [25] is A-order Only the linked representation generation algorithm of Solomon and Finkel [18] actually creates binary trees in this order.

Other representations used in the generation algorithms include set partitions [21], ballot sequences [14], position sequences [26], and code words [27]

These algorithms do not generate the trees in either natural ordering.

Generation Algorithms

The algorithms of Solomon and Finkel [18] and Wells [21] are both based on the following recurrence relation for catalan numbers (the number of binary trees with n nodes):

$$C_{n+1} = \sum_{i=0}^{n} C_i * C_{n-i}$$

$$C_0 = 1$$

The first binary tree generated by Solomon and Finkel is a binary tree such that the left child of every internal node is a leaf. The last tree is a binary tree such that the right subtree of every internal node is a leaf. The following recursive procedure describes the algorithm:

Algorithm 3.01

```
procedure next(tree)
    if the right subtree is the last of its size then
        n_right ← number of internal nodes in right subtree
        if the left subtree is the last of its size then
            if n_right =0 then
                no more trees
            else
                n_left ← number of internal nodes in left subtree
                left subtree ← first tree with n_left +1 internal nodes
                right subtree ← first tree with n_right -1 internal nodes
        else
            right subtree ← first tree with n_right internal nodes
            next( left subtree of tree )
    else
        next( right subtree of tree )
```

Solomon and Finkel also show that their algorithm runs in $O(n)$ time per tree. The proof is based on the fact that there are no loops in the procedure, and thus the procedure, excluding recursive calls and the time taken to create new subtrees with *n_left* or *n_right* internal nodes, takes constant time. The time taken to create the new subtrees would sum to a maximum of $O(n)$, regardless of the number of recursive calls. The maximum number of calls of next that could result from the original call of next would be n, and thus, it would take $O(n)$ time to determine the next tree.

The algorithm of Solomon and Finkel is faster than the algorithm by Wells, but it has an even more important advantage. The Solomon and Finkel algorithm does not require that representations of preceding trees be stored. Their algorithm generates the next tree from the current tree. Wells' algorithm, which generates all extented binary trees, requires that all trees with less than n internal node be created before the algorithm can create the

trees with n internal nodes. The problem with having to store preceding trees is that the number of trees grows exponentially.

A number of algorithms [7,8,10,12,25,26] represent extended binary trees as bit sequences. The algorithm of Proskurowski and Laiman [8] is based on this representation. The first binary tree generated by their algorithm is a tree such that the left subtree of every internal node is a leaf. This tree is represented by the sequence 1,0,1,0,1,0,...,1,0,0. The following procedure generates the next sequence. The variable *tree* is the preorder bit sequence representing the current binary tree. The global variable *last* denotes the position of the last element of the sequence that is a 1. There are n internal nodes in the binary tree.

Algorithm 3.02

```
procedure next
    w ← 0
    i ← last
    while tree_i =1 do
          tree_i ← 0
          w ← w + 1
          i ← i - 1
    tree_i ← 1
    if i = 0 then
          no more trees
    else if w > 1 then
          for i ← 1,2,...,w-1 do
                tree_{2(n-i)+1} ← 1
          last ← 2n - 1
    else
          last ← i
```

The authors of the paper maintain that their algorithm generates all binary trees with n internal nodes in time proportional to the number of binary trees with n internal nodes. The authors prove their claim by developing a charging scheme as follows.

The amount of work done by algorithm 3.02 is clearly proportional to the variable w. If $w > 1$ then in each of the preceding $w-1$ calls, the amount of work done would be proportional to 1. If we divide the cost of the current call up between the current call and the preceding $w-1$ calls, we end up with an average cost per call of 2. Thus the algorithm is on average, constant time per tree generated.

With some changes, the algorithm of Proskurowski and Laiman can be modified to generate all k-ary trees with n internal nodes. If they had substituted $k(n-i)+1$ for $2(n-i)+1$ and $k(n-1)+1$ for $2n-1$ then the algorithm would have generated all k-ary trees with n internal nodes.

Read [12], and Zaks [25,26] give algorithms that generate binary trees (or k-ary trees) by producing a series of bit strings that represent the binary trees. Zaks [26] translates the bit sequence to a position sequence. All of these algorithms, like the one by Proskurowski and Laiman [8] generate the next sequence from the current sequence. Read's algorithm takes $O(n)$ per tree generated. Both algorithms by Zaks are, on average, $O(1)$ per tree generated.

Proskurowski [7] uses the same bit sequence representation but his algorithm generates all the binary trees on n internal nodes by performing an augmenting operation on all the binary trees with $n-1$ internal nodes. His algorithm is recursive and does not need to store the trees with fewer internal nodes. His algorithm is also, on average, constant time per tree generated.

Read [10] also uses bit codes for his representation. Read's algorithm, however, generates all ordered trees. Read's paper defines an *orderly algorithm* as an algorithm that produces the canonical representations of all items of size n, in some specified order, from the list of canonical representations of all items of size $n-1$.

Read goes on to describe three conditions that must be satisfied if an algorithm is orderly. Given the existence of a list ordering $<$, the first condition is that all possible configurations of items of size n can be generated by augmenting some configuration of size $n-1$. The second condition is that the augmenting operation be such that if X and Y are configurations of size $n-1$ and X' is derived from X and Y' is derived from Y, and $X<Y$ then $X' \leq Y'$. The last condition is that the augmenting operation must produce the elements in the order defined by $<$.

As an example, he gives an orderly algorithm for generating all ordered trees with n internal nodes. The bit code *tree* represents the ordered tree. $tree_0$ is initialized to zero. The procedure call augment(1) will generate all bit codes with $2n$ elements. The algorithm is as follows:

Algorithm 3.03

```
procedure augment(a)
    if a = n then
        writetree
    else
        i ← 2a - 2
        tree_{2a} ← 1
        tree_{2a-1} ← 1
        while tree_i = 1 do
            i ← i - 1
        i ← i + 1
        tree_i ← 0
        for j ← i + 1, i + 2, ..., 2a
            augment(a + 1)
            tree_j ← 0
            tree_{j-1} ← 1
```

Although Read did not show that his algorithm was $O(1)$ per tree generated, his algorithm satisfies the conditions specified in a paper by Ruskey and Roelants van Baronaigien [17] for a backtracking algorithm to be $O(1)$, and thus Read's algorithm is on average, $O(1)$ per tree generated.

In Ruskey and Roelants van Baronaigien's [17] paper they present an algorithm for generating all ordered trees with a particular degree sequence. Zaks and Richards [24] also presented an algorithm to generate all ordered trees with a particular degree sequence. Both algorithms represent the trees with a preorder enumerations of degrees.

Ruskey and Roelants van Baronaigien [17] describe a general backtracking algorithm for the generation of combinatorial objects. If the algorithm satisfies certain conditions, it is $O(1)$ per object generated. As an example of such an algorithm, Ruskey and Roelants van Baronaigien produce an algorithm that generates all ordered trees with a particular degree sequence. The variable *head* is a pointer to a linked list that contains the degree sequence of the tree to be created. The functions *next* and *data* are standard linked list operations. The procedure *delete* reduces the number of nodes of degree *data(next(p))* by 1. The procedure *insert* restores the linked list to how it was before the call of *delete(p)*. The algorithm is as follows.

Algorithm 3.04

```
procedure buildtree(min,max)
    if next(head) = nil then
        printtree
    else
        for i ← max ,max -1,...,min do
            p ← head
            p1 ← next(head)
            while p1 <> nil do
                tree_i ← data(p1)
                delete(p)
                buildtree(i + 1, max + tree_i )
                insert(p,p1)
                tree_i ← 0
                p ← p1
                p1 ← next(p1)
```

The algorithm of Ruskey and Roelants van Baronaigien is $O(1)$ per tree generated. Zaks and Richards [24] algorithm is $O(n)$ per tree generated, where n is the total number of nodes in the tree and their algorithm is more complicated.

Several algorithms [14,20] are based on the integer sequence representation. They represent binary trees by labeling nodes in one order and reading off the labels in a different order. These authors use preorder and inorder. Trojanowski develops an m-inorder enumeration for k-ary trees where the first m subtrees are visited, the root is written and then the remaining subtrees are visited.

Rotem [13] show that there is a correspondence between ballot sequences and binary trees. To derive a ballot sequence from a binary tree, Rotem and

Varol translate the sequence generated by labeling the nodes in inorder and reading them in preorder, $S = s_1, s_2, \ldots, s_n$, into a ballot sequences by letting $S' = s_1', s_2', \ldots, s_n'$ where S_i' is the number of elements of S after the element i that are greater than i.

The algorithm presented here generates all ballot sequences with n elements. The first ballot sequence is a string of n zeroes. The procedure next will translate the current ballot sequence into the next ballot sequence.

Algorithm 3.05

```
procedure next
    tree_m ← tree_m + 1
    if tree_m > n - m then
        repeat
            m ← m + 1
            if m ≠ n then
                tree_m ← tree_m + 1
        until m = n or tree_m ≤ n - m
    if m = n then
        no more sequences
    else
        for i ← 1,2,...,m - 1
            tree_i ← tree_m
        m ← 1
```

Of the algorithms known at the time, Rotem and Varol [14] maintain that their algorithm was superior since they could generate a sequence of consecutive ballot sequences without having to generate all of them. Rotem and Varol did not analyse their algorithm but it is, on average, $O(1)$ per tree generated as we will now show.

Theorem 3.01: The algorithm of Rotem and Varol is average constant time per tree generated.

Proof: For any given call of the procedure next, the amount of work done by the procedure must be proportional to m. The amount of work done, in generating all the binary trees, by the procedure next would be proportional to $\sum_{i=1}^{n} i\,\tau(i)$ where $\tau(i)$ is the number of sequences where i is the largest number such that s_i was changed.

Let $S = s_1, s_2, \ldots, s_n$ be a ballot sequence representing a binary tree. Any subsequence $s_{m+1}, s_{m+2}, \ldots, s_n$ is also a ballot sequence and it represents a binary tree with $n-m$ nodes. Since there are C_{n-m} binary trees with $n-m$ nodes, there must be C_{n-m} ballot sequences with $n-m$ elements. It follows then that $\tau(i) = C_{n-i+1} - C_{n-i}$.

Thus the time taken to generate all trees with n nodes would be $T(n) = c\sum_{i=1}^{n} i\,(C_{n-i+1} - C_{n-i})$.

$$T(n) = c\sum_{i=1}^{n} i\,(C_{n-i+1} - C_{n-i})$$

$$= c\sum_{i=1}^{n} (C_{n-i+1} - C_{n-i}) + cT(n-1)$$

$$= cC_n + cT(n-1)$$

$$= c\sum_{i=1}^{n} C_i$$

and, since $C_n > \sum_{i=1}^{n-1} C_i$ for all $n > 2$, and the average amount of work per tree would be $cT(n)/C_n$, the generation algorithm is, on average, $O(1)$ per tree generated. ■

Trojanowski's [20] algorithm for generating binary trees is also, on average, $O(1)$ per tree generated.

Ruskey and Hu [15] developed an algorithm for generating extended binary trees. Their algorithm was based on a feasible sequence representation. A necessary condition for a sequence $S = s_1, s_2, \ldots, s_{n+1}$ to be feasible is that $\sum_{i=1}^{n+1} 2^{-s_i} = 1$. The feasible sequence is created by labeling each leaf with its level and then reading off the labels from left to right.

Ruskey [16] extended the results of Ruskey and Hu [15] so that the algorithm would generate all extended k-ary trees. Both the algorithm of Ruskey [16] and Ruskey and Hu [15] generate trees in B-order. The representation used by Ruskey was the same as the one used by Ruskey and Hu. The first tree generated by Ruskey's algorithm is a k-ary tree such that all but the right most child of any internal node is a leaf. That tree is represented by the sequence $\{1\}^{k-1}, \{2\}^{k-1}, \ldots, \{n-1\}^{k-1}, \{n\}^k$.

Algorithm 3.06

```
procedure next
    k ← N
    R ← 0
    r ← 0
    I ← 0
    while a_{k-t+1} <> a_k do
        k ← k-1
    if k = t then
        no more trees
    else
        if k+R+t-1 < N then
            while a_{k+R+t-1} = a_k - r - 1 and k+R+t-1 < N do
                R ← R+t-1
                r ← r+1

        for i ← 1,2,...,t do
            a_{k-i} ← a_{k-t}+1
        a_k ← a_k - r - 1
        for i ← k+1,k+2,...,N-R-1 do
            a_i ← a_{i+R}
        for i ← 0,1,...,r-1 do
            for j ← 0,1,...,t-2 do
                a_{N-R+I+j} ← a_N + i + 1
            I ← I+t-1
        a_N ← a_N + r
```

Ruskey's algorithm is, on average, $O(k)$ per tree generated. The time complexity of Ruskey and Hu's algorithm is $O(1)$.

Zerling's algorithm [27] is based on a representation that is not in common use. His algorithm generates all possible code words for binary trees. These code words describe how to convert a left linear tree, a binary tree such that all internal nodes have empty right subtrees, into the binary tree. If we were to label the left linear tree in preorder, the code word $S = s_1, s_2, \ldots, s_{n-1}$ would be such that s_i would denote the number of single right rotations to perform on node $n-i$.

The algorithm is a recursive backtracking algorithm and it generates all the code words in increasing lexicographic order. The array *tree* contains the code word. The element t_n must be initialized to zero. A call of *generate(n-1)* will list all of the code words for the binary trees with n nodes. The procedure generate is as follows:

Algorithm 3.07

```
procedure generate(k)
    unfolds ← 0
    max ← nodes − k − t_{k+1}
    while unfolds ≤ max do
        tree_k ← unfolds
        t_k ← t_{k+1} + tree_k
        unfolds ← unfolds + 1
        if k = 1 then
            output_tree
        else
            generate(k-1)
```

Zerling shows that his algorithm is, on average, $O(1)$ per sequence generated.

Most of the recent algorithms that have been created are constant time per tree generated. All but one, the algorithm of Solomon and Finkel [18], generate sequences that are equivalent to the class of trees which the

algorithm is trying to generate.

Author	ordering	time complexity	space complexity	Representation type
Proskurowski [7]	B-order	O(1)	O(n)	bit
Proskurowski and Laiman [8]	B-order	O(1)	O(n)	bit
Read [12]	B-order	O(n)	O(n)	bit
Rotem and Varol [14]	neither	O(1)	O(n)	ballot
Ruskey and Hu [15]	B-order	O(1)	O(n)	feasible
Solomon and Finkel [18]	A-order	O(n)	O(n)	linked
Trojanowski [20]	B-order	O(1)	O(n)	permutations
Wells [21]	neither	$O(e^n)$	$O(e^n)$	set partitions
Zaks [25]	B-order	O(1)	O(n)	position
Zaks [26]	neither	O(1)	O(n)	bit
Zerling [27]	neither	O(1)	O(n)	code words

Figure 3.01
Authors, the generation order and complexity of their algorithm.

The above table compares algorithms that generate binary trees or k-ary trees where $k=2$. The table references each algorithm by author, and specifies for each algorithm, the order that the trees are generated in, and the time and space complexity of the algorithm.

Ranking and Unranking

The preceding generation algorithms define an ordering on the trees. A ranking algorithm takes a tree and determines which position that tree would occur in in the list of all trees of its size. The unranking algorithm takes an integer N and determines the tree that would occur N-th in a list of trees of its size. These algorithms are useful for generating random trees for use when testing programs. Similar to generating algorithms, ranking and unranking algorithms also give information about the structure and properties of the trees.

There is very little difference between the ranking algorithms produced by different authors[5,8,14,15,16,18,20,24,25,26]. The general format is to recursively define the ranking function. Determining the rank of a particular tree involves determining the rank of each of its principle subtrees and multiplying the ranks of each principle subtrees by the number of possible ways of arranging the nodes that are in the principle subtrees that follow that particular subtree. We must also include the number of trees that would have had smaller principle subtrees further to the left of the tree. The ranking function of Knott[5] is a typical example of such an algorithm.

Algorithm 3.08

function rank(T)

if $|\ T\ | = 0$ **then**

rank $\leftarrow 1$

else

$$\text{rank} \leftarrow C_{|\ right(T)\ |} * \text{rank}(left(T)) + \text{rank}(right(T)) + \sum_{i=1}^{|\ left(T)\ |-1} C_i * C_{|\ T\ |-i-1}$$

The algorithm becomes more complex when we rank k-ary trees, but the principle is the same. In the above algorithm, use is made of several constants. In particular, C_n and $C_i\, C_j$. Many authors argue that ranking and unranking are done more than once, and thus the computation of these constants should not be included when determining the time complexity of the ranking or unranking algorithm. Such constants are said to be precomputed.

In analysing this algorithm, one can see that if the constants are precomputed, then the algorithm is $O(n)$ where n is the number of nodes in the binary tree.

The algorithms that do unranking are similar to the ranking algorithms. The process is inverted.

CHAPTER 4

A New Algorithm for Generating K-ary Trees in A-order

Introduction

Over the last ten years there has been much work done on the generation of different types of trees. Most of the work has been concentrated on the generation of ordered trees, although some work has been done on the generation of rooted trees. In this chapter, any reference to binary or k-ary trees means full binary or full k-ary trees.

Within the class of ordered trees, work has been done in several areas. Work done by Ruskey and Hu [15] and Ruskey [16] has been in the area of generating binary trees and then k-ary trees. Both of the algorithms were based on the generation of sequences which were successively altered so as to remain feasible.

The generation algorithm of Zaks [25], also designed to generate k-ary trees, uses position sequences to represent the k-ary trees. Zaks defines two orderings on the set of k-ary trees, A-order and B-order. The algorithm developed by Zaks [25] generates k-ary trees in B-order. Zaks goes on to say that there was no known algorithm which generates k-ary trees in A-order.

This chapter will describe an algorithm for generating all k-ary trees in A-order. Algorithms for ranking and unranking k-ary trees will also be developed.

Representation

A number of papers have been written that describe algorithms for generating k-ary trees that involve the generation of sequences. In Zaks [25] paper, a k-ary tree is represented by labeling the internal nodes with a one and the leaves with a zero, and then enumerating the labels in preorder. The

positions of the ones are then recorded.

The papers by Ruskey and Hu[15] and Ruskey[16] represent binary trees, and then generalize to k-ary trees, by labeling the leaves with their level number, and then listing them in a left to right order.

The algorithm developed here is also based on the generation of sequences. A k-ary tree is represented in the following manner. We first label each node with its weight, and then we read off the labels in preoreder, thus generating a sequence.

Definition 4.01 A sequence $S=s_1,s_2,\ldots,s_{nk+1}$ is called a *preorder enumeration of weights* if there is full k-ary tree with n internal nodes such that if each node of the tree were to be labeled with the weight of the node, and then the labels were read off in preorder, the resulting sequence would be S.

Lemma 4.01: Let $n \geq 0$ and $k > 1$ then $S=s_1,s_2,\ldots,s_{nk+1}$ is a preorder enumeration of weights for a k-ary tree with n internal nodes if and only if one of the following properties hold.

Either:

(1) $n=0$ and $S=0$, or

(2) $n>0$ and $s_1=n$, and

letting $n_1=2$, and $n_i=n_{i-1}+ks_{n_{i-1}}+1$,

the sub-sequence $s_2,s_3,\ldots,s_{nk+1}$ is such that

the k sub-sequences $s_{n_i},s_{n_i+1},\ldots,s_{n_{i+1}-1}$

are preorder enumerations of weights for all i, $1 \leq i \leq k$, and

$\sum_{i=1}^{k} s_{n_i}=n-1$

Proof: $\rightarrow$ We will use induction to show that if a sequence is a preorder enumeration of weights, then the sequence satisfies either property 1 or property 2.

If $n=0$ then the k-ary tree is a single leaf. If we label this tree with its weights and then read off the labels of the tree in preorder, we would get the sequence 0, and thus property 1 is satisfied.

Assume that one of the conditions described in lemma 4.01 is satisfied for each k-ary trees with less than n internal nodes.

Let T be a k-ary tree with n internal nodes, and let $T_1, T_2, \ldots, T_k$ be the k principle sub-trees of T. Let $S_1, S_2, \ldots, S_k$ be preorder enumerations of weights such that each sequence S_i is the preorder enumeration of weights of T_i, and let $n_i = 2 + \sum_{j=1}^{i-1} |S_j|$. It can be seen that the n_i defined here have the same value as the n_i defined in property (2). Because the labels are read in preorder, we know that all of the labels in the i-th principle sub-tree will be listed before any label from the $i+1$-st principle sub-tree is listed, and we also know that $s_1 = n$. Clearly, n_i is the position that the sequence S_i starts in the sequence S. We know from our inductive hypotheses, since the weight of any given principle sub-tree of T is less than n, that each sequence S_i satisfies either property (1) or property (2), and that the first element in any sequence S_i is the weight of the i-th subtree of T and thus $\sum_{i=1}^{k} s_{n_i} = n-1$, and thus the sequence S is such that it satisfies property (2).

$\leftarrow$ Let $S = s_1, s_2, \ldots, s_{nk+1}$ be a sequence satisfying property (1) or property (2). The following procedure will construct the k-ary tree that is represented by S.

```
procedure build(S,T)
  label(T_root) ← s_1
  if s_1 > 0 then
    f ← 2
    for i ← 1,2,...,k do
      l ← f + k s_f
      S' ← s_f, s_{f+1}, ..., s_l
      build(S', T_i)
      f ← l + 1
```

■

Lemma 4.01 defines a sequence of numbers $n_1, n_2, \ldots, n_k$ which are the starting positions of the sequences representing the principle sub-trees of of the k-ary tree represented by S. The number n_{k+1} is also defined. It is the value ks_1+2 which is 1 more than the position of the last element in the sequence S. These numbers will be referred to later in the chapter. The sequences $s_{n_i}, s_{n_i+1}, \ldots, s_{n_{i+1}-1}$ may be referred to as *principle subsequences*.

Let $S = s_1, s_2, \ldots, s_{kn+1}$ be a preorder enumeration of weights. It can be observed that lemma 4.01 implies that if we replace a subsequence $s_j, s_{j+1}, \ldots, s_{j+ks_j}$ of S with a subsequence $S' = s_j', s_{j+1}', \ldots, s_{j+ks_j}'$ of the same length, and the subsequence, S', is a preorder enumeration of weights, then the resulting sequence is a preorder enumeration of weights.

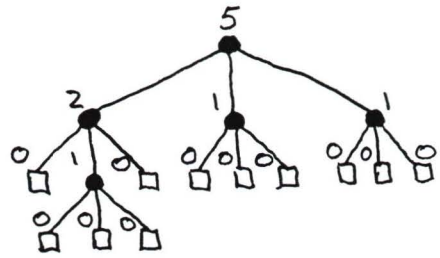

Figure 4.01
The 3-ary tree that is represented by the sequence
5,2,0,1,0,0,0,0,1,0,0,0,1,0,0,0

This representation of a full k-ary tree varies only slightly from the representation used by Zaks[25]. In Zaks representation, instead of labeling the internal nodes with their weight, internal nodes are labeled with a 1. Both labelings are read off in preorder.

Lemma 4.02: Let S and S' be two feasible sequences representing k-ary trees T and T' respectively, then $S > S'$ in lexicographic order if and only if $T > T'$ in A-order.

Proof: $\rightarrow$ We will use induction on the size of the size of T to prove that if T and T' are two k-ary trees such that S is the preorder enumeration of weights representing T and S' is the preorder enumeration of weights representing T' and $T > T'$ then $S > S'$.

Let T be a k-ary tree with one internal node, and let T' be a k-ary tree such that $T > T'$. Clearly T' must be a leaf. The sequence representing T is $S = 1\{0\}^k$ and the sequence representing T' is $S' = 0$. Clearly $S > S'$.

Assume that lemma 4.02 is true for all k-ary trees with fewer than n internal nodes.

Let T and T' be two k-ary trees such that $T > T'$ in A-order. There are two possibilities, either $|T| > |T'|$ or $|T| = |T'|$ and there exists some i, $1 \leq i \leq k$ such that $T_j = T_j'$ for all $j < i$ and $T_i > T_i'$.

If $|T| > |T'|$ then, since the first element in S, the sequence representing T, is $|T|$, and the first element in S', the sequence representing T', is $|T'|$, $S > S'$.

Assume that $|T| = |T'|$. Let S be the preorder enumeration of weights representing T and let S' be the preorder enumeration of weights representing T'. We know that there exists some i, $1 \leq i \leq k$ such that $T_j = T_j'$ for all $j < i$ and $T_i > T_i'$. Let v be the total number of internal nodes in the principle sub-trees $T_1, T_2, \ldots, T_{i-1}$. Since the sequences representing T and T' are preorder enumerations, the sub-sequences $s_1, s_2, \ldots, s_{kv+i}$ and $s_1', s_2', \ldots, s_{kv+i}'$ will be identical.

We know from our inductive hypotheses that, since $T_i > T_i'$ the sequence S_i representing T_i is greater than the sequence S_i' representing T_i', and since both sequences start at position $kv+i+1$, $S > S'$.

$\leftarrow$ We will use induction on the length of the sequences to prove that if S and S' are two preorder enumerations of weights representing k-ary trees T and T' respectively, and if $S > S'$ then $T > T'$.

Let $S = 1, \{0\}^k$ and let S' be a preorder enumeration of weights such that $S > S'$. Clearly $S' = 0$. The k-ary tree represented by $1, \{0\}^k$ is the k-ary tree with 1 internal node. The k-ary tree represented by 0 is the k-ary tree with 0 internal nodes. In A-order, a k-ary tree with one internal node is greater than a k-ary tree with zero internal nodes.

Assume lemma 4.02 is true for all sequences of length less than $kn+1$.

Let $S = s_1, s_2, \ldots, s_{kn+1}$ be a preorder enumeration of weights representing the k-ary tree T, and let $S = s_1', s_2', \ldots, s_{km+1}'$ be a preorder enumeration of weights representing the k-ary tree T', and let $S > S'$.

The case where $m \neq n$ is trivial so assume $m = n$. Since $S > S'$ we know there exists an integer i, $1 < i \leq kn+1$ such that $s_j = s_j'$ for all j, $1 \leq j < i$, and that $s_i > s_i'$. Since S and S' are preorder enumerations of weights, we know that s_i is the label of some node in some principle sub-tree of T. Without loss of generality assume that s_i is a label in the t-th principle sub-tree. Clearly s_i' must be a label in the t-th principle sub-tree of T'. Since the sequences are identical up to position i, $T_j = T_j'$ for all $j < t$. By lemma 4.01 we know that S contains k principle subsequences. Let S_t be a principle subsequence representing T_t and let S_t' be a principle subsequence representing T_t'. Since $s_i > s_i'$, $S_t > S_t'$, and thus, by our induction hypotheses, $T_t > T_t'$. Thus $T > T'$ ■

It follows from the above lemma that if we were to list all the feasible sequences, in lexicographic order, the corresponding list of full k-ary trees would be in A-order.

The generating algorithm is based on this fact.

Algorithm

This algorithm generates preorder enumerations of weights representing full k-ary trees with n internal nodes in descending lexicographic order. Within algorithm 4.01, $init(n)$ refers to the sequence representing the first tree with n internal nodes. The first tree with n internal nodes is represented by the sequence $S = n, n-1, n-2, \ldots, 2, 1, \{0\}^{n(k-1)+1}$.

The procedure next transforms the current preorder enumeration of weights into the next preorder enumeration of weights, in lexicographic order. The procedure next makes use of several functions which are defined later.

Algorithm 4.01

procedure next
  $j \leftarrow \text{hrs}(s_1, s_2, \ldots, s_{nk+1})$
  if $j = 1$ **then**
    no more trees
  else
    $s_j, s_{j+1}, \ldots, s_{j+k(s_j-1)} \leftarrow init(s_j - 1)$ **substitution 1**
    $w \leftarrow 1 + \text{siblingweights}(j)$
    $j \leftarrow j + ks_j + 1$
    $s_j, s_{j+1}, \ldots, s_{j+kw} \leftarrow init(w)$ **substitution 2**
    $j \leftarrow f(j)$
    while $j > 1$ **do**
      $w \leftarrow \text{siblingweights}(j)$
      $j \leftarrow j + ks_j + 1$
      $s_j, s_{j+1}, \ldots, s_{j+kw} \leftarrow init(w)$ **substitution 3**
      $j \leftarrow f(j)$

The function siblingweights is defined as follows:

function siblingweights(n)
  $\sigma \leftarrow 0$
  $\alpha \leftarrow n + ks_n + 1$
  while $p(n) = p(\alpha)$ **do**
    $\sigma \leftarrow \sigma + s_\alpha$
    $\alpha \leftarrow \alpha + ks_\alpha + 1$
  return(σ)

The function p is defined as follows:

```
function p(n)
  α ← n−1
  while ks_α+α<n do
    α ← α−1
  return(α)
```

The function f is defined as follows:

```
function f(j)
  α ← j+ks_j+1
  while s_α=0 and α<kn+1 do
    α ← α+1
  if s_α>0 then
    i ← j
    while p(α)≠p(i) or s_i=0 do
      i ← i−1
    return(i)
  else
    return(0)
```

The function hrs is defined as follows:

```
function hrs(s_1,s_2,...,s_nk+1)
  j ← nk+1
  while (k(s_p(j)−s_j)+p(j)≤j) or (s_j=0) do
    j ← j−1
  return(j)
```

Lemma 4.03: If S' is the sequence generated by the application of algorithm 4.01 to the preorder enumeration of weights S, then S' is also a preorder enumeration of weights.

Proof: We know from property 2 of lemma 4.01 that if we replace one subsequence that is a preorder enumeration of weights with another subsequence of the same size that is also a preorder enumeration of weights then the resulting sequence is a preorder enumeration of weights.

Let $S = s_1, s_2, \ldots, s_{kn+1}$ be a preorder enumeration of weights representing the k-ary tree T with n internal nodes. Let $S' = s_1', s_2', \ldots, s_{kn+1}'$ be the sequence created by applying algorithm 4.01 to the sequence S.

To show that S' is a preorder enumeration of weights, we must show that S' satisfies the properties defined in lemma 4.01.

Let $j =$ hrs(S). We know that the sequence $S_1 = s_{p(j)}, s_{p(j)+1}, \ldots, s_{p(j)+ks_{p(j)}}$ is a preorder enumeration of weights. Because of the definition of the function p(j), the sequence S_1 contains the subsequence $s_j, s_{j+1}, \ldots, s_{j+ks_j}$. Let n_i be defined, as described in property (2) of lemma 4.01, for the sequence S_1. Without loss of generality, assume that s_j would be the m-th element in the summation described in property 2 of lemma 4.01. The function siblingweights(j) returns $\sum_{i=m+1}^{k} s_{n_i}$. Let $w =$ siblingweights(j)+1. Let

$$S_1' = s_{p(j)}, s_{p(j)+1}, \ldots, s_{j-1}, s_j - 1, s_j - 2, \ldots, 1, \{0\}^{(k-1)(s_j-1)+1}, w, w-1, w-2, \ldots, 1, \{0\}^{w(k-1)+1}, \{0\}^{k-(m+1)}$$

Clearly the sequences $w, w-1, w-2, \ldots, 1, 0^{w(k-1)+1}$ and $s_j - 1, s_j - 2, \ldots, 1, \{0\}^{(k-1)(s_j-1)+1}$ are preorder enumerations of weights. The subsequence $s_{p(j)+1}, s_{p(j)+2}, \ldots, s_{j-1}$ is unchanged by the substitution, and thus $\sum_{i=1}^{m-1} s_{n_i}$ would remain unchanged. The element s_j has been reduced by 1, but the element s_{j+ks_j+1} is 1 larger than $\sum_{i=m+1}^{k} s_{n_i}$. Since the last k elements

in any preorder enumeration of weights are always zero, the new sequence S_1' satisfies property (2) of lemma 4.01. Thus the application of substitutions (1) and (2) produce a new sequence that satisfies property (2) of lemma 4.01.

A similar argument can be made to show that the application of substitution (3) will also produce a new sequence that satisfies property (2) of lemma 4.01. Since all three substitutions create sequences that satisfy property 2 of lemma 4.01, the new sequence S' is a preorder enumeration of weights. ■

Theorem 4.01: Given a feasible sequence S, algorithm 4.01 generates the next feasible sequence in decreasing lexicographic order.

Proof: By lemma 4.03, algorithm 4.01, if given a feasible sequence, will only generate a new feasible sequence. Therefore, all we need to show is that the algorithm generates feasible sequences in lexicographic order.

Let $S = s_1, s_2, \ldots, s_{nk+1}$, and let S be a preorder enumeration of weights. Let $S' = s_1', s_2', \ldots, s_{nk+1}'$, and let S' be the preorder enumeration of weights generated by the application of algorithm 4.01 on the sequence S. Let j be the smallest integer such that $s_j \neq s_j'$.

Since substitution 1 affects the left most subsequence of S to be affected, and substitution 1 replaces s_j with $s_j - 1$, $S > S'$. We also know that j is the integer that would be produced by the first loop in the procedure next. We still must show that there are no sequences missed by algorithm 4.01.

Let $Z = z_1, z_2, \ldots z_{nk+1}$ be a preorder enumeration of weights such that $S > Z > S'$. Let i be the smallest integer such that $s_i' < z_i$. Since $S > Z$ and S and S' are identical up to position j, $i \geq j$. If $i = j$ then $s_i = z_i$ and thus there would be a larger value of j that would be found by the first loop in procedure next, thus $i > j$, and $s_i' = z_i$. Since substitution 1 replaces the subsequence $s_j, s_{j+1}, \ldots, s_{j+ks_j}$ with $s_j - 1, s_j - 2, \ldots, 1, \{0\}^{(s_j-1)(k-1)+1}$ which is the lexicographically greatest subsequence of the same length,

$i > j + ks_j$. Substitutions 2 and 3 replace subsequences with the lexicographically greatest possible subsequence of the same length. These lexicographically greatest subsequences occur at the left most possible position of S', and thus i must be the position of an element of S' and Z after the last element of S that was affected by algorithm 4.01. Since the last element of S that would be affected by algorithm 4.01 is the last non zero element of S', any element after that must be a zero, and since S' and Z are identical up to position i, $z_i = 0$. So there is no value of i such that $z_i > s_i'$, and thus S' follows S in lexicographic order. ■

Any algorithm that lists all k-ary trees induces an order on those trees. It is therefore useful to consider an algorithm which determines the position of a particular tree in the list created by the generation algorithm. Such an algorithm is known as a ranking algorithm.

Ranking Algorithm

The ranking algorithm presented here takes a preorder enumeration of weights, S, of size $kn+1$ and returns the rank of the sequence S in the list of all possible preorder enumerations of weights of the same size. The number of preorder enumerations of weighs greater than S is the rank of S. The ranking algorithm is based on the following: let n_j be as defined in lemma 4.01, the number of preorder enumerations of weights of size $nk+1$ that start with $n+1$ is zero, and if $i = n_{k-p+1}$ then the number of preorder enumerations of weights that start with a subsequence greater than $s_1, s_2, \ldots, s_{i+ks_i}$ is the number of sequences that start with subsequence greater than $s_1, s_2, \ldots, s_{i-1}$ plus the number of sequences that start with $s_1, s_2, \ldots, s_{i-1}, q$ where $q > s_i$ plus the number of preorder enumerations of weights greater than $s_i, s_{i+1}, \ldots, s_{i+ks_i}$ that are of the same length times the number of ways of creating $p-1$ preorder enumerations of weights that represent $p-1$ k-ary trees with $\sum_{j=k-p}^{k} s_{n_j}$ internal nodes between them.

We define $N(n, p, m)$ as the number of ways of creating p preorder enumerations of weights $S_1, S_2, \ldots, S_p$ such that $|S_1| \geq km+1$. Thus there

are

$$\sum_{\substack{n_1+n_2+\cdots+n_p=n \\ n_1 \geq m}} T_{n_1} T_{n_2} \cdots T_{n_p}$$

such sequences where T_n is the number of extended k-ary trees with n internal nodes. For convenience, $N(n,0,m)=1$ and if $m>n$ then $N(n,p,m)=0$

A table of $N(n,p,m)$ for binary and 3-ary trees is given in appendix 1. We believe that, for k-ary trees, the values of $N(n,p,m)$ satisfy the following recurrence relation

$$N(n,p,m)=N(n,p-1,m)+N(n-1,p-1+k,m)$$

$$N(n,1,m)=T_n$$

$$N(n,0,m)=1$$

$$N(n,p,n)=T_n$$

where T_n is the number of k-ary trees with n internal nodes.

The ranking function assumes the existence of an array, $N(n,p,m)$, where $N(n,p,m)$ is defined as above. The time needed to compute the array $N(n,p,m)$ would be $O(kn^3)$. The function rank is as follows:

Algorithm 4.02

```
function rank(S)
  σ←0
  n←s_1−1
  if n>0 then
    i←2
    for p←k,k−1,k−2,...,1 do
      σ←σ+N(n,p,s_i+1)
      n←n−s_i
      S'←s_i,s_{i+1},...,s_{i+ks_i}
      σ←σ+rank(S')N(n,p−1,0)
      i←i+ks_i+1
  return(σ)
```

Theorem 4.02: Algorithm 4.02 computes the number of sequences of the same length that occur before the sequence S in lexicographic order.

Proof: The proof will be by induction on the length of the sequence. Clearly, if the preorder enumeration of weights is of size 1 or $k+1$, the function will return the value zero, and since there is only 1 preorder enumeration of weights of either size, the function returns the correct value for all preorder enumerations of weights of size less than or equal to $k+1$.

Assume algorithm 4.02 computes the correct value for all preorder enumerations of weights of a size less than $nk+1$.

Let S be a preorder enumeration of weights, and let $S=s_1,s_2,\ldots,s_{nk+1}$. From the definition of *numbers*, we know that $N(n,p,s_i+1)$ is the number of ways of creating p preorder enumeration of weights such that the first sequence starts with a number greater than s_i, and the sequences represent p k-ary trees with n internal nodes between them. Since these sequences would form the subsequence $s_i,s_{i+1},\ldots,s_{i+kn+p}$, $N(n,p,s_i+1)$ is the number

of sequences that start with $s_1, s_2, \ldots, s_{i-1,q}$ where $q > s_i$

The $k-p$-th time through the loop we add $N(n, p, s_i+1)$ to σ

The value rank$(s_i, s_{i+1}, \ldots, s_{i+ks_i})$, is the number of preorder enumerations of weights that are greater than $s_i, s_{i+1}, \ldots, s_{i+ks_i}$ that are of length ks_i+1. Let n_j be as defined in lemma 4.01. We know that the $k-p+1$-st time through the loop, $n = s_1 - 1 - \sum_{j=1}^{k-p} s_{n_j} = \sum_{j=k-p+1}^{k} s_{n_j}$, and thus $N(n, p-1, 0)$ is the number of ways of creating $p-1$ preorder enumerations of weights that represent $p-1$ k-ary trees with n internal internal nodes between them.

Therefore, the $k-p$-th time through the loop, we add the number of preorder enumerations of weights greater than $s_i, s_{i+1}, \ldots, s_{i+ks_i}$ that are of the same length times the number of ways of creating $p-1$ preorder enumerations of weights that represent $p-1$ k-ary trees with $\sum_{j=k-p}^{k} s_{n_j}$ internal nodes between them.

After we have gone through the loop k times, σ is the number of preorder enumerations of weights that start with a subsequence that is greater than $s_1, s_2, \ldots, s_{n_{k+1}-1}$, which is the number of preorder enumeration of weights that are greater than S. ■

The ordering implied by algorithm 4.01 also gives rise to the question of determining the preorder enumeration of weights S of length $kn+1$ that is preceded by a specified number of preorder enumerations of weights. This is known as unranking.

Unranking Algorithm

The unranking algorithm does the inverse of the ranking algorithm. Given two values, n and r the unranking algorithm will determine S such that the preorder enumeration of weights S would occur in position r in a list of all preorder enumerations of weights of size $kn+1$.

The first element of the sequence S, s_1, would be n. Let n_i be as defined in lemma 4.01, and let $n > 0$, then the preorder enumeration of

weights is made up of k preorder enumerations of weights, $s_{n_i}, s_{n_i+1}, \ldots, s_{n_{i+1}-1}$. Each of these principle subsequences has a rank. For each of the principle subsequences, the while loop determines a value for σ such that the maximum rank of a sequence that starts $s_1, s_2, \ldots, s_{i-1}, \sigma$ would be greater than r, but the maximum rank of a sequence that starts $s_1, s_2, \ldots, s_{i-1}, \sigma+1$ would be less than or equal to r. The number σ is thus the number that belongs at position i. The rank of the preorder enumeration of weights $s_i, s_{i+1}, \ldots, s_{i+k\sigma}$ is then calculated and the procedure unrank is used to determine the sequence $s_i, s_{i+1}, \ldots, s_{i+k\sigma}$. The algorithm is as follows:

Algorithm 4.03

```
procedure unrank(S, n, r)
  s_1 ← n
  if n > 0 then
    n ← n − 1
    i ← 2
    for p ← k, k − 1, ..., 1 do
      σ ← 0
      while N(n, p, σ+1) > r do
        σ ← σ + 1
      if σ < n then
        r ← r − N(n, p, σ+1)
      n ← n − σ
      unrank(S', σ, r div N(n, p, 0))
      r ← r mod N(n, p, 0)
      s_i, s_{i+1}, ..., s_{i+kσ} ← S'
      i ← i + kσ + 1
```

Assuming the array $N(n, m, p)$ has been precomputed, the ranking algorithm is $O(kn)$ and the unranking algorithm is $O(kn \log n)$. A detailed analysis of all three algorithms is contained in chapter 5.

CHAPTER 5

Analysis of the New Algorithms

Introduction

We now determine the space and time complexity of the algorithms given in chapter 4. When analysing the generation algorithm, we first show that the time taken by a call of the procedure *next* is proportional to the number of nonzero elements of the sequence S that are to the right of $s_j > 0$ where $k(s_{p(j)} - s_j) + p(j) > j$. Having shown that, we can calculate the time taken to generate all k-ary trees and then determine the average time taken per tree. At this time, we are unable to determine the time complexity of the general case, however, we conjecture that the binary tree case is the worst case, and in the binary tree case the generation algorithm is $O(1)$ time.

We are able to show that, on average, the generating algorithm is $O(1)$ when $k = 2$. In the case of binary trees, finding the largest value of j such that $s_j > 0$ and $k(s_{p(j)} - s_j) + p(j) > j$ is equivalent to finding the largest value of j such that $s_{j-1} > s_j > 0$.

The analysis of the ranking and unranking algorithms is relatively simple. When analysing these algorithms, we do not include the time taken to produce the array $N(n, p, m)$. The ranking algorithm is $O(nk)$ and the unranking algorithm is $O(kn \log n)$.

Generating Algorithm

We start with a theorem to show that the amount of work done by a call of the procedure *next* is proportional to the number of nonzero entries in S to the right of s_j.

Theorem 5.01: Let $S=s_1,s_2,\ldots,s_{nk+1}$, and let S be a preorder enumeration of weights. Let j be the largest integer such that $s_j>0$ and $k(s_{p(j)}-s_j)+p(j)>j$. Let j' be the number of nonzero elements of S that occur on or after position j. The amount of work done by algorithm 4.01 in generating the next preorder enumeration of weights is proportional to j'.

Proof: We can implement the function $p(n)$ by using an array P_n which is updated any time a substitution is performed and thus any call of the function $p(n)$ is $O(1)$. We can also create an array $LEFTSIB_i$ which is updated any time a substitution is made. The array element $LEFTSIB_i$ would be the value of the largest integer less than i such that $p(i)=p(LEFTSIB_i)$. If no such value exists, $LEFTSIB_i=0$. If in addition to the sequence S we have an array POS where POS_i is the position of the i-th nonzero element in S, we can determine the value of j in $O(j')$ time since we can bypass all nonzero elements of S.

Any of the substitutions, including the updating of POS and $LEFTSIB$ would take $O(s_j)$ time where s_j is the first number in the new sequence being substituted in, and thus the total time taken by all the substitutions in algorithm 4.01 would be $O(j')$.

The while loop in algorithm 4.01 would execute enough times to rearrange the $j'-(s_j-1)-w$ remaining nonzero elements of S. The function $f(j)$ can be implemented in $O(1)$ time using the arrays $LEFTSIB$ and POS. The function $f(j)$ would merely return the value of $LEFTSIB$ of the first nonzero element after position $j+kw+1$, and thus the execution of all of the contents of the while loop, excluding substitution 3, would take $O(j'-(s_j-1)-w)$ time.

The function *siblingweights* would be implemented, using the arrays POS, and P in $O(siblingweights(j))$ time, and thus the sum of the time required for all calls of *siblingweights* would be $O(j')$.

The time taken by a call of the procedure *next* is thus $O(j')+O(j')+O(j'-(s_j-1)-w)+O(j')$, which is $O(j')$ time. ■

Definition 5.01: Let $\tau(n,\iota)$ be the number of preorder enumerations of weights with $nk+1$ elements such that if $\jmath = hrs(S)$, then there would be $\iota-1$ nonzero elements following $s_\jmath$

Definition 5.01 is equivalent to stating that $\tau(n,\iota)$ is the number of extended k-ary trees where the last $\iota-1$ internal nodes that would be enumerated in a preorder traversal have no right sibling, but the internal node ι-th from last does

Theorem 5.02: The time taken to generate all preorder enumerations of weights with $nk+1$ elements is

$$T(n) = c \sum_{\iota=1}^{n} \iota\, \tau(n,\iota)$$

Proof Obvious from the definition of τ and theorem 5.01 ■

We will now show that in the case of binary trees, that algorithm 4.01 is, on average, $O(1)$ per tree generated It is useful to show that the condition $k(s_{p(\jmath)}-s_\jmath)+p(\jmath)>\jmath$ and $s_\jmath>0$ is equivalent to the condition that $s_{\jmath-1}>s_\jmath>0$ We also show a recurrence relation for the $\tau(n,\jmath)$

Lemma 5.01: If $k=2$ then the condition $k(s_{p(\jmath)}-s_\jmath)+p(\jmath)>\jmath$ and $s_\jmath>0$ is equivalent to the condition that $s_{\jmath-1}>s_\jmath>0$

Proof → Let S be a preorder enumeration of weights, and let $S=s_1,s_2,\ldots,s_{nk+1}$ Let $\jmath$ be the largest integer such that $k(s_{p(\jmath)}-s_\jmath)+p(\jmath)>\jmath$ and $s_\jmath>0$. We must show that $\jmath$ is the largest integer such that $s_{\jmath-1}>s_\jmath>0$

Assume that ι, $\iota>\jmath$ is an integer such that $s_{\iota-1}>s_\iota>0$ then clearly $k(s_{p(\iota)}-s_\iota)-p(\iota)>\iota$ so $\jmath$ must be the largest integer such that $s_{\jmath-1}>s_\jmath>0$

← Let S be a preorder enumeration of weights, and let $S=s_1,s_2,\ldots,s_{nk+1}$ Let $\jmath$ be the largest integer such that $s_{\jmath-1}>s_\jmath>0$ We

must show that j is the largest integer such that $k(s_{p(j)}-s_j)+p(j)>j$ and $s_j>0$

Clearly $s_j>0$ Assume there is an integer $i>j$ such that $k(s_{p(i)}-s_i)+p(i)>i$ and $s_i>0$ If $p(i)=i-1$ then we have that $s_{i-1}>s_i>0$, so assume $p(i)\neq i-1$ Since $k=2$ we have

$$2(s_{p(i)}-s_i)+p(i)>i$$

and since every subsequence $s_m, s_{m+1}, \ldots, s_{m+ks_m}$ is a preorder enumeration of weights, we know that $s_{p(i)+1}+s_i=s_{p(i)}-1$, and thus

$$i<2((s_{p(i)+1}+s_i+1)-s_i)+p(i)$$

and hence $i<2(s_{p(i)+1}+1)+p(i)$

and hence $i\leq 2s_{p(i)+1}+p(i)+1$

However, the function $p(i)$ is defined such that $p(i)$ is the largest integer such that $2s_{p(i)}+p(i)\geq i$ and thus we have a contradiction Therefore $p(i)=i-1$ ■

Lemma 5.02: If $k=2$ then $\tau(n,n)=1$, $\tau(n,0)=0$, $\tau(n,-1)=0$, and in all other cases, $\tau(n,i)=\sum_{j=i-1}^{n-1}\tau(n-1,j)$

Proof The initial cases are obvious, $\tau(n,n)$ is the number of sequences generated where there are no more trees, and since that is last one to be generated, there must only be 1 Similarly, a value of $j=0$ or $j=-1$ is impossible, and thus $\tau(n,0)$ and $\tau(n,-1)$ must be zero

Assume the formula is true for all $\tau(n-1,h)$, $1\leq h\leq j$

Let $S=s_1,s_2,\ldots,s_{2k-1}$ be a sequence such that $s_{h-1}>s_h>0$ and $1\leq h\leq j$ Let m be the position of the $j-1$th non zero element in the sequence If we construct a new sequence $s_1',s_2',\ldots,s_m',1+s_{m+1},0,s_{m+1},s_{m+3},\ldots s_{2n-1}$ such that

$$s_i' = \begin{cases} s_i + 1 & \text{if } i \in \{p(m), p(p(m)), \ldots, 1\} \\ s_i & \text{otherwise} \end{cases}$$

it is clear that this new sequence is a preorder enumeration of weights with $2n+1$ elements. Because of the way the original sequence was chosen, and because of the way the new sequence was constructed, the new sequence is such that if the j-th nonzero element was in position m then m would be the largest element such that $s_{m-1} > s_m > 0$. It is also clear from the construction of the new sequences, that no duplicate sequences would be created. Thus the total number of all these sequences is $\sum_{j=i-1}^{n-1} \tau(n-1, j)$. ■

Corollary 5.01: $\tau(n, i) = \tau(n, i+1) + \tau(n-1, i-1)$

Theorem 5.03: Algorithm 4.01 takes, on average, constant time per sequence generated for k=2.

Proof:

$$
\begin{aligned}
T(n) &= \sum_{i=1}^{n} i\,\tau(n,i) \\
&= \sum_{i=1}^{n} i\,(\tau(n-1,i-1)+\tau(n,i+1)) \\
&= \sum_{i=1}^{n} i\,\tau(n-1,i-1)+\sum_{i=1}^{n} i\,\tau(n,i+1) \\
&= \sum_{i=1}^{n} (i-1)\tau(n-1,i-1)+\sum_{i=1}^{n} \tau(n-1,i-1)+\sum_{i=1}^{n} i\,\tau(n,i+1) \\
&= T(n-1)+C_{n-1}+\sum_{i=1}^{n} i\,\tau(n,i+1) \\
&= T(n-1)+C_{n-1}+\sum_{i=1}^{n} (i+1)\tau(n,i+1)-\sum_{i=1}^{n} \tau(n,i+1) \\
&= T(n-1)+C_{n-1}+(\sum_{i=1}^{n} i\,\tau(n,i)-\tau(n,1))-(\sum_{i=1}^{n} \tau(n,i)-\tau(n,1)) \\
&= T(n-1)+C_{n-1}+T(n)-C_n
\end{aligned}
$$

thus $T(n-1)=C_n-C_{n-1}$

and therefore the average time taken to generate a binary tree with n internal nodes would be

$$T(n)/C_n=(C_{n+1}-C_n)/C_n) \quad \blacksquare$$

At this time the author is unable to either show a recurrence relation for the $\tau(n,i)$ for $k>2$ or to find a formula for $\tau(n,i)$. Experimental evidence suggests the conjecture that $\tau(n,i) = T(n,i,0)$ where $T(n,k,p)$ are defined by Ruskey [16]. Ruskey gave both a recurrence relation for these numbers as well as a closed form.

Conjecture 5.01: For all $n \geq 0$ and $k > 1$,

$$\tau(n,\iota) = \frac{k\iota - \iota}{kn - \iota} \binom{kn-\iota}{n-\iota}$$

Theorem 5.04: Given that conjecture 5.01 is true, then the time taken to generate all preorder enumerations of weights with $nk+1$ elements is, on average, $O(1)$ per tree generated.

Proof: We know that the time taken to generate all trees, $T(n) = \sum_{\iota=1}^{n} \iota\, \tau(n,\iota)$ and thus the average time per tree generated would be $\frac{T(n)}{T_n}$ where T_n is the number of k-ary trees with n internal nodes. Therefore

$$\begin{aligned} T(n) &= \sum_{\iota=1}^{n} \iota\, \tau(n,\iota) \\ &= \sum_{\iota=1}^{n} \iota \frac{k\iota - \iota}{kn - \iota} \binom{kn-\iota}{n-\iota} \end{aligned}$$

and since we are interested in the average amount of work per tree, if we divide through by T_n we get

$$\begin{aligned} \frac{T(n)}{T_n} &= \sum_{\iota=1}^{n} \iota \frac{k\iota - \iota}{kn - \iota} \binom{kn-\iota}{n-\iota} \frac{kn+1}{\binom{kn+1}{n}} \\ &= \sum_{\iota=1}^{n} \frac{\iota\,(k\iota - \iota)(kn-\iota)!\; n!\; (kn-n+1)!\; (kn+1)}{(kn-\iota)(n-\iota)!\; ((kn-\iota)-(n-\iota))!\; (kn+1)!} \\ &= \sum_{\iota=1}^{n} \frac{\iota\,(k\iota - \iota)(kn-\iota)!\; n!\; (kn-n+1)!}{(kn-\iota)(n-\iota)!\; ((kn-n))!\; (kn)!} \\ &= \sum_{\iota=1}^{n} \frac{\iota\,(k\iota - \iota)(kn-\iota-1)!\, n!\; (kn-n+1)}{(n-\iota)!(kn)!} \\ &= \sum_{\iota=1}^{n} \frac{\iota\,(k\iota - \iota) n!\; (kn-n+1)}{(n-\iota)!(kn)(kn-1)\ldots(kn-\iota)} \end{aligned}$$

It can be observed from the above equation that $\frac{T(n)}{T_n}$ is at a maximum when $k=2$, and since we know that algorithm 4.01 is $O(1)$ per tree generated when $k=2$, the algorithm must be average constant time per tree generated for any value of $k \geq 2$ ■

The analysis of the ranking and unranking algorithms is much simpler than the analysis of the generation algorithm. As mentioned in chapter 4, both the ranking and unranking algorithms assume the existence of a precomputed array $N(n,p,m)$.

Ranking and Unranking Algorithms

We now analyse the ranking and unranking algorithms. The algorithms are no faster than algorithms that presently exist for ranking and unranking in B-order.

Theorem 5.05: Let $S=s_1,s_2,\ldots,s_{kn+1}$, and let S be a preorder enumeration of weights. Algorithm 4.02 determines the rank of S in $O(kn)$ time.

Proof. Let $T(n)$ represent the time taken by algorithm 4.02 to determine the rank of a sequence, S, with $kn+1$ elements in it and let n_i be as described in property two of lemma 4.01. Clearly the following recurrence relation describes $T(n)$.

$$T(n)=k+\sum_{i=1}^{k} T(n_i)$$

$$T(1)=1$$

$$T(0)=1$$

We will use induction on n to show that $T(n)\leq 2kn+1$. From the boundary conditions for the recurrence relation, we can see that $T(0)\leq 1$ and $T(1)\leq 2k+1$. Assume that $T(i)\leq ki+1$ for all $i<n$.

We can see from the recurrence relation that $T(n)=k+\sum_{i=1}^{k} T(n_i)$.

Since we know that all of the $T(n_i) \leq 2kn_i + 1$, and we know that $n - 1 = \sum_{i=1}^{k} n_i$ it follows that $\sum_{i=1}^{k} T(n_i) \leq k + 2k(n-1)$ and thus $T(n) \leq 2kn + 1$ Therefore algorithm 4.02 is $O(kn)$. ■

Theorem 5.06: Algorithm 4.03 determines the sequence S with rank r of size $kn + 1$ in $O((k+n)n)$ time.

Proof: Since the while loop can never produce a value greater than the current value of n, and since each time the while loop is executed, n will be decreased by one, the while loop can be executed at most n times regardless of the number of times the for loop is executed. The remainder of the instructions will take $O(k)$ time. Therefore, the procedure unrank takes time $T(n)$ where $T(n)$ is described by the following recurrence relation.

$$T(n) = k + n + \sum_{i=1}^{k} T(n_i)$$

$$T(0) = 1$$

The remainder of the proof can be done by induction on n ■

Of course, the time complexity of both the ranking and unranking algorithms is dependent on precomputing the constants $N(n, p, m)$.

CHAPTER 6

Summary and Conclusions

Summary

In Chapter 2 we discussed current methods of representing binary, k-ary and ordered trees. These methods include various types of integer sequences, bit sequences and linked representations. We then discuss various methods that have been created to generate all binary, k-ary and ordered trees of a particular size. Each of these methods defines an ordering on their respective set of trees. Zaks [25] describes two natural orderings for k-ary trees and then shows that the algorithms that were known at the time produced k-ary trees in B-order, he also stated that there was no known algorithm for generating k-ary trees in A-order.

In Chapter 4, we present an algorithm for generating all full k-ary trees in A-order. We prove the algorithm to be correct and then we show that in the case of binary trees, that the algorithm is $O(1)$ per tree generated. We also present algorithms for ranking and unranking k-ary trees. These algorithms are shown to be $O(kn)$ and $O(kn \log n)$ respectively.

Although experimental evidence suggests that

$$r(n,i)=\frac{ki-i}{kn-i}\binom{kn-i}{n-i}$$

the author is unable to prove that conjecture. Given that lemma 5.03 was correct, the author was able to show that algorithm 4.01 was $O(1)$ per tree generated regardless of the value of k.

Conclusions

In the thesis, we first surveyed the current literature on the generation of different types of ordered trees. Having summarized the results, we identified an area where no known algorithm existed, and we produced the algorithm. The algorithm lists all k-ary trees in A-order and appears to do so in $O(1)$ per tree. There has been an algorithm produced to list all binary trees in A-order, however, that algorithm was $O(n)$ per tree generated and we have shown that the new algorithm presented in this thesis is $O(1)$ in the case of binary trees.

The review done in this thesis seems to indicate that there is little work left to be done in the area of ordered tree generation. Many algorithms exist which generate trees in average constant time, and thus there is little room for improving on the existing work.

Further Research

Further research could done to show that lemma 5.03 is true and that the recurrence relation given in lemma 5.02 could be generalized to k-ary trees. Experimental evidence indicates that the recurrence would be

$$\tau(n,i) = \sum_{j=i-1}^{n-1} \binom{j+k-i-1}{k-2} \tau(n-1,j)$$

$$\tau(n,0) = 0$$

$$\tau(n,-1) = 0$$

$$\tau(n,n) = 1$$

The author was able to prove the special case of $k=2$ however the general case still requires attention.

It might also be of interest to investigate the relationship between $\tau(n,i)$ and the numbers $T(n,k,p)$ in Ruskey's paper [16].

References

[1] Arnold D B and Sleep M R , "Uniform Random Generation of Balanced Parenthesis Strings", ACM Trans on Programming Languages and Systems, v2 (1980), pp 122-128

[2] Beyer, T and Mitchell-Hedetniemi, S , "Constant Time Generation of Rooted Trees", SIAM J Comput , v9 (1980), pp 706-712

[3] de Bruijn, N and Morselt, B , "A Note on Plane Trees", J Combinatorial Theory, v2 (1967), pp 27-34

[4] Klarner, D , "Correspondences between Plane Trees and Binary Sequences", J Combinatorial Theory, v9 (1970), pp 401-411

[5] Knott, G D , "A Numbering System for Binary Trees", Comm ACM, v20 (1977), pp 113-115

[6] Knuth, D E ,"Fundamental Algorithms", v1, second ed , Addison Wesley, Reading, MA, 1973

[7] Proskurowski, A , "On the Generation of Binary Trees", J ACM, v27 (1980), pp 1-2

[8] Proskurowski, A and Laiman, E , "Fast Enumeration, Ranking, and Unranking of Binary Trees" Congressus Numerantium, v35 (1982), pp 401-413

[9] Read, R C , "The Coding of Various Kinds of Unlabeled Trees", Graph Theory and Computing, R C Read ed , Academic Press, New York, NY, 1972, pp 153-182

[10] Read, R C , "Every One a Winner" Annals of Discrete Math , v2 (1978), pp 107-120

[11] Read, R C and Corneil, D G , "The Graph Isomorphism Disease", J Graph Theory, v1 (1977), pp 339-363

[12] Read, R C , "A Survey of Graph Generation Techniques", Lecture Notes in Mathematics No 884, v8 (1982), Springer Verlag, Berlin, pp 77-89

[13] Rotem, D , "On a Correspondence Between Binary Trees and a Certain Type of Permutation", Information Processing Lett , v4 (1975), pp 58-61

[14] Rotem, D and Varol, Y L , "Generation of Binary Trees from Ballot Sequences", J ACM, v25 (1978), pp 396-404

[15] Ruskey, F and Hu, T C , "Generating Binary Trees Lexicographically", SIAM J Comput v6 (1977), pp 745-758

[16] Ruskey, F , "Generating t-ary Trees Lexicographically", SIAM J Comput v6 (1977), pp 424-439

[17] Ruskey, F and Roelants van Baronaigien, D , "Fast Recursive Algorithms for Generating Combinatorial Objects" Congressus Numerantium, v41 (1984), pp 53-62

[18] Solomon, M and Finkel, R A , "A Note on Enumerating Binary Trees", J ACM, v27 (1980), pp 3-5

[19] Standish, T , "Data Structure Techniques" Addison Wesley, Philippines, (1980)

[20] Trojanowski, A E , "Ranking and Listing Algorithms for k-ary Trees", SIAM J Comput v7 (1978), pp 492-509

[21] Wells, M , "Elements of Combinatorial Computing" Pegammon Press, (1971)

[22] Wilf, H S , "A Unified Setting for Sequencing, Ranking and Selection Algorithms for Combinatorial Objects", Adv in Math , v24 (1977), pp 281-291

[23] Williamson, S G , "On the Ordering, Ranking, and Random Generation of Basic Combinatorial Sets", Proceedings of Table Ronde, Combinatoire et Representation du Group Symmetrique, April 1976, Springer Verlag Lecture Notes in Mathematics No 579, Springer Verlag, Berlin, pp 309-339

[24] Zaks, S and Richards, D , "Generating Trees and Other Combinatorial Objects Lexicographically", SIAM J. Comput, v8 (1979), pp 73-81

[25] Zaks, S., "Lexicographic Generation of Ordered Trees", Theoret. Comput. Sci., v10 (1980), pp. 63-82.

[26] Zaks, S., "Generation and Ranking of K-ary Trees" Information Processing Lett., v14 (1982), pp. 44-48.

[27] Zerling, D., "Generating Binary Trees Using Rotations" J. ACM, v32 (1985), pp. 694-701.

APPENDIX 1

a table of $N(n, p, m)$

binary trees

n=1

m	p	1	2	3	4	5	6
1		1	1	1	1	1	1

n=2

m	p	1	2	3	4	5	6
1		2	3	4	5	6	7
2		2	2	2	2	2	2

n=3

m	p	1	2	3	4	5	6
1		5	9	14	20	27	35
2		5	7	9	11	13	15
3		5	5	5	5	5	5

n=4

m	p	1	2	3	4	5	6
1		14	28	48	75	110	154
2		14	23	34	47	62	79
3		14	19	24	29	34	39
4		14	14	14	14	14	14

n=5

m	p	1	2	3	4	5	6
1		42	90	165	275	429	637

2	42	76	123	185	264	362
3	42	66	95	129	168	212
4	42	56	70	84	98	112
5	42	42	42	42	42	42

3-ary trees

n=1

m	p	1	2	3	4	5	6	7	8	9
1		1	1	1	1	1	1	1	1	1

n=2

m	p	1	2	3	4	5	6	7	8	9
1		3	4	5	6	7	8	9	10	11
2		3	3	3	3	3	3	3	3	3

n=3

m	p	1	2	3	4	5	6	7	8	9
1		12	18	25	33	42	52	63	75	88
2		12	15	18	21	24	27	30	33	36
3		12	12	12	12	12	12	12	12	12

n=4

m	p	1	2	3	4	5	6	7	8	9
1		55	88	130	182	245	320	408	510	627
2		55	76	100	127	157	190	226	265	307
3		55	67	79	91	103	115	127	139	151
4		55	55	55	55	55	55	55	55	55

n=5

m	p	1	2	3	4	5	6	7	8	9
1		273	455	700	1020	1428	1938	2565	3325	4235
2		273	400	557	747	973	1238	1545	1897	2297
3		273	364	467	582	709	848	999	1162	1337
4		273	328	383	438	493	548	603	658	713
5		273	273	273	273	273	273	273	273	273

VITA

Surname: Roelants van Baronaigien Given Names: Dominique
Place of Birth: Duncan B.C. Date of Birth: Sept. 17, 1958

Educational Institutions Attended, with Dates of Entering and Leaving:

University of Calgary, Alberta	1976 to 1978
University of Victoria, B.C.	1978 to 1986

Degrees, Diplomas, Etc., Awarded, With Dates and Names of Institutions:

B.Sc. (Computer Science)	1980	University of Calgary
B.Sc. (Psychology)	1982	University of Victoria

Publications:

Frank Ruskey and D. Roelants van Baronaigien
"Fast Recursive Algorithms for Generating Combinatorial Objects"
Congressus Numerantium, v41 (1984), pp. 53-62.

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis (title of which is shown below) to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the university designated by me. It is understood that permission for extensive copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title: **Ordered Tree Generation Algorithms**

Author: [redacted]

Dominique Roelants van Baronaigien

Date: May 26, 1986