Generation of Rooted Trees and Free Trees

by

Gang Li
B.S., People's University of China, 1990
M.S., University of Calgary, 1994

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

Dr. Frank Ruskey, Co-supervisor (Department of Computer Science)

Dr. Dominique Roelants, Co-supervisor (Department of Computer Science)

Dr. John Ellis, Department Member (Department of Computer Science)

Dr. G. MacGillivray, External Examiner (Department of Mathematics and Statistics)

University of Victoria

Supervisor: Dr. Frank Ruskey

ABSTRACT

In this thesis we present two new recursive algorithms for generating unlabeled rooted trees and unlabeled free trees. We first simplify Beyer-Hedetniemi's [5] iterative algorithm for generating rooted trees by using the more natural parent array representation. We then develop a new recursive algorithm which is much simpler, more flexible and easier to analyze. With some simple modifications, our algorithm will generate rooted trees of size n with height lying in a given range and/or with the number of children of each node bounded by a given integer. Our recursive algorithm for generating rooted trees is then extended to generate free trees. We also show how to make some simple modifications to generate free trees with some height constraints and/or with a degree restriction. Both algorithms and their modifications run in constant amortized time which is the best possible.

Examiners:

Dr. Frank Ruskey, Co-supervisor (Department of Computer Science)

Dr. Dominique Roelants, Co-supervisor (Department of Computer Science)

Dr. John Ellis, Department Member (Department of Computer Science)

Dr. G. MacGillivray, External Examiner(Department of Mathematics and Statistics)

Contents

List of Tables

List of Figures

List of Symbols

$\mathbf{B}_n$ .... The set of all binary rooted trees with n nodes

b_n .... The number of binary rooted trees with n nodes

$\mathcal{R}_n$ ..... The computation tree of our recursive rooted tree algorithm with n nodes

$\mathcal{R}_{<h}$. The computation tree of our recursive rooted tree algorithm with heights less than h

$\mathcal{R}_{=h}$ ..... The computation tree of our recursive rooted tree algorithm with heights exactly h

$\mathcal{R}_{>h}$ ..... The computation tree of our recursive rooted tree algorithm with heights greater than h

$\mathbf{R}_n$ .... The set of all unlabeled rooted trees with n nodes

r_n .... The number of unlabeled rooted trees with n nodes

$\mathcal{F}_n$ .... The computation tree of our recursive free tree algorithm with n nodes

$\mathbf{F}_n$ .... The set of all unlabeled free trees with n nodes

f_n .... The number of all unlabeled free trees with n nodes

T_r .... The rooted version of free tree T rooted at r

$\mathbf{L}_n$ .... The set of all labeled free trees with n nodes

lev_T .... The level sequence of the rooted tree T

par_T .... The parent array of the rooted tree T

$\langle lev_T[1..n]\rangle$ .... The level sequence of the rooted tree T with n nodes

$\langle par_T[1..n]\rangle$ .... The parent array of the rooted tree T with n nodes

$par[p]$ .... Parent of node p

$\prec$ .... Lexicographically smaller than

$\preceq$ .... Lexicographically smaller than or equal to

$\succ$ .... Lexicographically larger than

$\succeq$ .... Lexicographically larger than or equal to

$|T|$ .... The number of nodes in the tree T

$T(r)$ .... The subtree of tree T rooted at r

$succ(T)$ .... The successor of the rooted tree T in relex order

Acknowledgements

My deepest thanks to my supervisor Dr. Frank Ruskey and Dr. Dominique Roelants for their academic supervision and financial support. Without them, this thesis will not be possible.

My thanks to Department of Computer Science for providing the computing facilities and financial support.

I also thank my wife, my parents for every thing they did to help me and encourage me to complete my graduate program here in University of Victoria.

Chapter 1

Introduction

1.1 What are Trees and What is Tree Generation?

A *(free) tree* is a undirected, acyclic, connected graph. A *rooted tree* is a free tree in which one of the vertices is distinguished from the others. The distinguished node is called the *root* of the tree (we always draw the root at the top, see Figure 1.1). An *ordered tree* is a rooted tree in which the children of each node are ordered. A *labeled rooted tree* is an ordered tree in which every node is uniquely labeled by an integer from $\{1, 2, \ldots, n\}$, where n is the size of the tree (see Figure 1.1). For a tree T, by $|T|$ we denote the size of the tree T, i.e., the number of nodes in T. There are many different labeling schemes. A preorder labeling is obtained by labeling the nodes in the order that they are encountered in a preorder traversal of the tree. We will be using this labeling scheme throughout the thesis.

The tree is a widely used data structure in computer science. Various kinds of trees have been developed for different purposes. For example, quadtrees and octrees are hierarchical data structures for efficiently storing image data (see [31]); AVL trees are height balanced trees used for fast key searching in database systems.

Early mathematical research on tree was done by Cayley in the 1850's [20]. He found recursive formulas for counting the number of trees , or rooted trees, of finitely many nodes, where the degree of each node was not limited. An illustration of these

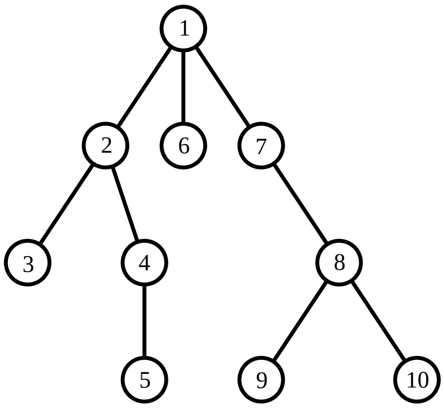

Figure 1.1: A preorder labeled rooted tree with root 1.

formulas can be also found in Knuth's book[6] and Otter's paper [12].

Tree enumeration was possibly first found useful by chemists in the study of structurally isomeric, aliphatic hydrocarbons. Cayley was the first to introduce quartic trees (every node has degree one or four) which represents the structure of the hydrocarbons C_nH_{2n+2}[4].

1.2 Representations of Trees

There are many different ways to represent a tree. One of the very popular ways is label representation sequences. We first label the tree, and then form a sequence of the nodes using the labels assigned to them, which defines the structure of the tree. For some trees, additional information has to be put into the sequence, such as the color of each node in red-black trees. Usually, for rooted trees and free trees, we use positive numbers $1, 2 \ldots, n$ as labels, so the sequences which represent these trees are of positive integers.

Let $s = \langle s_1, s_2, \ldots, s_m \rangle$ be a sequence of positive integers, and $t = \langle t_1, t_2, \ldots, t_n \rangle$ be another sequence of positive integers. Let $k = min(m, n)$. We say s is lexicographically smaller than t, or $s \prec t$, if the following conditions are satisfied:

1) there exists an integer $j > 0$ and $j \leq k$ so that $s_i = t_i$ for $0 < i < j$, and $s_j < t_j$, or

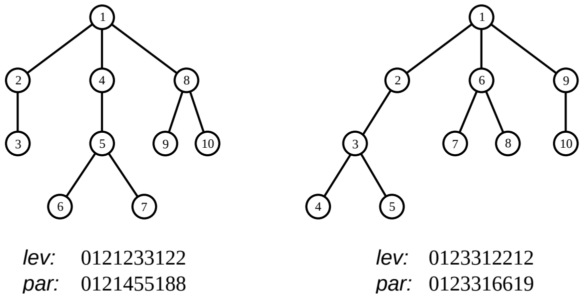

Figure 1.2: Two equivalent rooted trees with their level sequences and parent arrays. The tree and its representations on the right are canonic.

2) $s_i = t_i$ for $0 < i \leq k$ and $n > m$.

If sequence s is the representation of tree S and t is the representation of tree T, then we say S is lexicographically smaller than T, written $S \prec T$, if $s \prec t$.

1.2.1 Rooted Trees

Among all of the representations for rooted trees, the *level sequence* and the *parent array* will be the most frequently used in this thesis.

Let $\mathbf{R}_n$ be the set of all unlabeled rooted trees with n vertices. Let tree $T \in \mathbf{R}_n$ be arbitrarily ordered and then preorder labeled. The encoding sequence $\langle l_1, l_2, \ldots, l_n \rangle$ is the *level sequence* of T if l_i is the level of node i in T. The encoding sequence $\langle p_1, p_2, \ldots, p_n \rangle$ is the *parent array* of T if p_i is the parent of node i in T. The parent of node 1 is 0. By level, we mean the length of the path from the node to the root of the tree. We also use $\langle lev_T[1..n] \rangle$, lev_T for short, to represent the level sequence of T of size n, and $\langle par_T[1..n] \rangle$, or par_T, to represent the parent array of T.

Given a rooted tree T, we use $T(r)$ to denote a subtree rooted at r which includes r and all descendants of r in T. A node d in a rooted tree T is a *descendant* of a node e in T if and only if e is on the path from d to the root of T (A node is not descendant of itself).

For the rooted tree shown in Figure 1.1, its level sequence is 0122311233 and its parent array is 0122411788. See also Figure 1.2 for some other examples.

Two ordered trees are *equivalent* if one can be transformed to the other by recur-

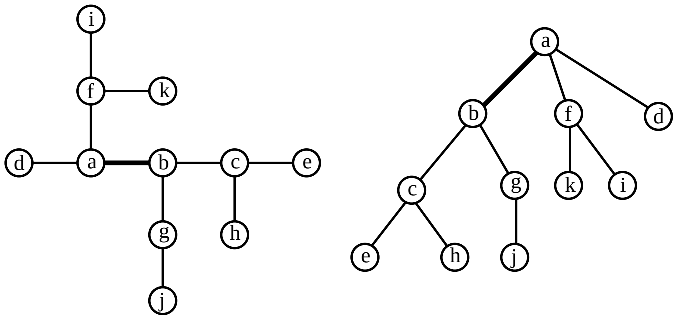

Figure 1.3: A free tree with two centers a and b. The tree on the right is its corresponding rooted version.

sively reordering the subtrees. The two trees shown in Figure 1.2 are equivalent. This equivalence relation partitions the set $\mathbf{O}_n$ of ordered trees into equivalence classes, and $\mathbf{R}_n$ is the set of the representatives of these equivalence classes. Now the problem is to pick a representative for each class. We say an ordered tree T in $\mathbf{O}_n$ is *canonic* if its level sequence lev_T is lexicographically greatest in its class. In Figure 1.2, the tree on the right is canonic. If an ordered tree is canonic, we also say that its level sequence and parent array are canonic. If the level sequence of a canonic rooted tree T is lexicographiclly greater than that for another canonic rooted tree T', then we also say T is (lexicographically) greater than T'. In this thesis, unlabeled rooted trees will be represented by their canonic level sequences or their canonic parent arrays, unless otherwise indicated.

1.2.2 Free Trees

It is easy to represent a rooted tree because of its distinguishable root. For free trees, things are different.

Let $\mathbf{L}_n$ be the set of all labeled free trees of size n, and $\mathbf{F}_n$ be the set of all unlabeled free trees of size n. A *center* of a free tree is a node whose maximal distance to all other nodes in the tree is minimal. It is well known [32] that every free tree has either one or two adjacent centers (see Figure 1.3). Let T be a labeled free tree in $\mathbf{L}_n$. If T has one center r, then T_r is the *rooted version* of T by rooting T at node r. If T has two centers, q and r, then they must be adjacent. By removing the edge between these two nodes, we obtain two rooted subtrees $Q(q)$ and $R(r)$. Let Q be the canonic rooted tree equivalent to $Q(q)$ and R be the canonic rooted tree for $R(r)$. If lev_Q is lexicographically less than or equal to lev_R, then we pick q as the root for T which gives the rooted version T_q of free tree T; otherwise we pick r which gives the rooted version T_r of T. In either case (one center or two), this root in the rooted version of a free tree is called the *canonic center* of the free tree, and the canonic parent array or level sequence of this rooted version is called the *canonic representation* of the free tree T.

As long as we transform a free tree into its corresponding rooted tree, we can represent the free tree by the level sequence or the parent array of the corresponding canonic rooted tree. And clearly, the above mapping between unlabeled free trees and unlabeled rooted trees is one to one. Now, we can generate all unlabeled free trees in $\mathbf{F}_n$ by listing all the canonic representations of corresponding rooted trees of length n (see Figure 1.4).

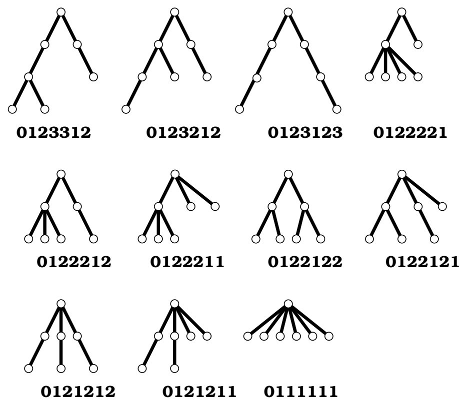

Figure 1.4: Free trees with 7 nodes and their canonic representation in level sequences

Similarly to the rooted tree case, when we say free tree T is greater than free tree T', we mean that the representation, either in level sequence or in parent array, of T is lexicographically greater than that of T'.

1.3 About the Generation of Trees

There are two different ways of generating trees: random generation and listing. Random generation is also called random selection which one constructs or chooses a tree of a given size uniformly at random (see Wilf [17]). By listing, we mean that all trees of a certain size will be produced in some designated order.

In this thesis, we will only discuss problems on listing trees. Hence generating means listing in this thesis.

Usually, algorithms for generating trees will produce the trees in some specific order. There are two kinds of orderings that are commonly used: lexicographic order, and Gray code order. As for lexicographic order, there are some variants (see Figure 1.5 for examples): *relex order* which is reversed lexicographical order, and *colex order* which is lex order if we reverse each sequence. *Gray code* is an ordering in which two consecutive sequences are very close in terms of a certain closeness relation. For example, two sequences differ only in one position.

If the cost of generating a tree by an algorithm is bounded by a constant, amortized over all trees, we say the algorithm is CAT(Constant Amortized Time).

Generation algorithms fall into one of two classes: iterative or recursive. By iterative generation, we mean the next tree is generated by a function call which works on the currently generated or initialized tree. By recursive generation, we mean the main generation function is recursive and each tree is generated by a series of recursive calls instead of only one function call as in iterative algorithms.

In Chapter 2 of this thesis, you will be introduced to some previously known algorithms for generating rooted trees. Among those, the Beyer-Hedetniemi algorithm will be given detailed attention. Chapter 3 will present a new recursive algorithm for generating unlabeled rooted trees, and show that, compared with the Beyer-Hedetmieni algorithm, this new algorithm is much simpler, flexible, more efficient, and easier to analyze. We will then extend this algorithm to unlabeled free trees by first introducing some previous algorithms for generating free trees, in Chapter 4, and then presenting the new recursive algorithm, in Chapter 5, for generating free

relex order	Gray code	lex order	colex order
11110000	11001010	10101010	11110000
11101000	11001100	10101100	11101000
11100100	11011000	10110010	11011000
11100010	11010100	10110100	10111000
11011000	11010010	10111000	11100100
11010100	11110000	11001010	11010100
11010010	11101000	11001100	10110100
11001100	11100100	11010010	11001100
11001010	11100010	11010100	10101100
10111000	10110010	11011000	11100010
10110100	10110100	11100010	11010010
10110010	10111000	11100100	10110010
10101100	10101100	11101000	11001010
10101010	10101010	11110000	10101010

Figure 1.5: The same set of strings (all well formed parenthesis of length 8) listed in different orders.

trees. The new recursive algorithm for free trees is CAT, uses linear space, and is more efficient (even though it is recursive) than the old algorithms.

It is our convention that we use *Propositions* to describe straightforward facts which will help to understand the following discussion, or to assist the proof of a lemma or a theorem. A *Lemma* is a statement which is not trivial and hence needs a proof. A *Theorem* is a major result which either comes from a reference paper or needs a proof.

Chapter 2

Previous Algorithms for Generating Rooted Trees

2.1 An Introduction

Rooted trees occur throughout computer science. For example, they are used in data structures for disjoint sets, and in mathematics, where they are studied in conjunction with bracketing systems, and even in biology, where they are used for the evolutionary classification of species. This chapter first discusses how to count the number of rooted trees, then briefly introduces some previous algorithms for generating rooted trees. The algorithm due to Beyer and Hedetniemi will be presented in detail.

2.2 Counting the unlabeled rooted trees

In order to design algorithms to generate all unlabeled rooted trees, it is useful to know the number of them with fixed size n. We follow Knuth's [6] discussion on counting rooted trees.

For small trees, we can just draw them to figure out this number. Figure 2.1 gives all possible unlabeled rooted trees of size 4.

For any given size n, let $r_n = |\mathbf{R}_n|$ be the number of unlabeled rooted trees. Obviously $r_1 = 1$. If $n > 1$, the tree has a root and various subtrees. Let j_k be the

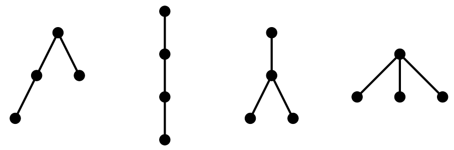

Figure 2.1: All unlabeled rooted trees of size 4.

number of subtrees with k nodes. Then we have

$$\binom{r_k + j_k - 1}{j_k}$$

ways to choose (with repetition) j_k of the r_k possible $k-$vertex rooted trees, which gives us

$$r_n = \sum_{j_1+2j_2+\cdots+(n-1)j_{n-1}=n-1} \prod_{k=1}^{n-1} \binom{r_k + j_k - 1}{j_k} \tag{2.1}$$

Cayley[20] further found that the generating function for r_n satisfies:

$$A(x) = x/(1-x)^{r_1}(1-x^2)^{r_2}(1-x^3)^{r_3}\cdots.$$

Using equation 2.1, we can compute a table of r_n for $n = 1 \ldots 14$ (see Table 2.1).

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14
r_n	1	1	2	4	9	20	48	115	286	719	1842	4766	12486	32973

Table 2.1: The number r_n of rooted trees with n nodes

2.3 Previous Algorithms for Generating Rooted Trees

Several algorithms have been developed for generating (i.e., listing) rooted trees.

The earliest algorithm seems to be that of Scions [16]. He introduced the level sequence representation of the rooted trees. Both recursive and iterative approaches were applied in his algorithm. He claimed his algorithm was CAT (no proof was given).

Another method of generating rooted trees was developed by Kozina [19] whose algorithm ran in time $O(n)$ per tree (not CAT).

Pallo [15] introduced a new encoding, called the weight sequences, of binary rooted trees, and algorithms for generating, ranking and unranking binary rooted trees were given. His algorithm for generating binary rooted trees is CAT.

An algorithm for generating binary trees in lexicographic order was presented by Ruskey and Hu [9]. Ruskey [7] later extended this algorithm to k-ary trees which generate k-ary trees in $O(k)$ per tree.

Very recently, Vajnovszki [25] presented a new encoding for binary unordered trees and used this coding scheme to generate binary unordered rooted trees. This algorithm is CAT.

However, the most wellknown algorithm is due to Beyer and Hedetniemi [5] (It is also described in the book of Wilf [17]). They generalized Ruskey's [7] result by presenting a very simple iterative CAT algorithm for generating unlabeled rooted trees.

Kubicka and Kubicki[14] later extended Beyer and Hederniemi's [5] algorithm to generate binary rooted trees in constant amortized time.

We will discuss Beyer and Hedetniemi's algorithm in detail because it is related to our new recursive algorithm which will be presented in Chapter 3.

2.4 The Beyer-Hedetniemi Iterative Algorithm

Beyer and Hedetniemi's algorithm iteratively generates all canonic rooted trees represented by level sequences. They were the first to show an algorithm for generating rooted trees in constant time per tree, amortized over all trees, i.e., a CAT algorithm.

The algorithm lists these canonic level sequences in relex (reversed lexicographic) order. Let $lev_T = \langle l_1, l_2, \ldots, l_n \rangle$ be the level sequence of a rooted tree T, let $succ_T = \langle s_1, s_2, \ldots, s_n \rangle$ denote the relex successor of lev_T and let

$$p = max\{i : l_i > 1\} \text{ and } q = max\{i : i < p, l_i = l_p - 1\}.$$

In other words, p is the position of the rightmost element larger than 1, and q is the position of the parent of node p. For example, in the level sequence $\langle 01233211 \rangle$, $p = 6$ and $q = 2$. Note that p must be a leaf. Then s_i is determined by

$$s_i = \begin{cases} l_i & \text{for } 1 \leq i < p \\ s_{i-p+q} & \text{for } p \leq i \leq n. \end{cases} \tag{2.2}$$

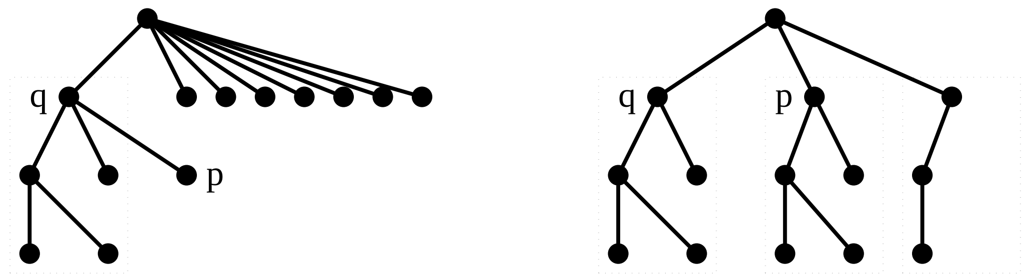

Figure 2.2: Copy Strategy: The tree on the right is the successor of the tree on the left.

The subsequence $l_q, l_{q+1}, \ldots, l_{p-1}$ represents the subtree rooted at q. The update of s_i's repeatedly copies this subsequence (or this subtree rooted at q). For example, the successor of the level sequence $\langle 01233221111111 \rangle$ is $\langle 01233212332123 \rangle$. See Figure 2.2.

We will extend this idea later to recursively generating rooted trees, and we call it the *Copy Strategy* (see Chapter 3).

The algorithm starts with the lexicographically largest canonic level sequence of length n, which is $\langle 012 \ldots n-1 \rangle$. It then repeatedly applies the successor function (2.2) to the current canonic level sequence to obtain its successor, until the lexicographically smallest canonic level sequence, $\langle 0111 \ldots 1 \rangle$, is obtained. The output of the Beyer-Hedetniemi's algorithm for $n = 6$ is shown in Figure 2.3.

Given a canonic level sequence $\langle l_1, l_2, \ldots, l_n \rangle$, the procedure in Figure 2.4 will find the successor of the level sequence:

The direct implementation of the above algorithm will not be CAT, since Step 1, to find p will take $n - p + 1$ actions (an action is a comparison or an assignment which takes constant amount of time independent of n) , and Step 2, to find q will take another $p - q$ actions. Together with Step 3, there will be $2 * (n - p + 1) + p - q$

012345	012321	012345	012321
012344	012315	012344	012312
012343	012311	012343	012311
012342	012222	012342	012222
012341	012221	012341	012221
012333	012215	012333	012212
012332	012211	012332	012211
012331	012141	012331	012121
012325	012111	012323	012111
012322	011111	012322	011111
(a)		(b)	

Figure 2.3: (a) The parent arrays of canonic rooted trees of size 6 in relex order. (b) The corresponding level sequences.

procedure NEXT();
  Step 1: find the biggest index p so that $l_p > 1$.
  Step 2: find the parent position q of node p.
  Step 3: copy the subsequence $l_q, \ldots, l_{p-1}$ repeatedly into subsequence $l_p, \ldots, l_n$.
end; {of NEXT}

Figure 2.4: Beyer and Hedetniemi's algorithm

actions. A clever implementation of the algorithm will reduce $p - q$ to 1, so that the total actions taken by NEXT() will only be $2(n - p + 1) + 1$.

We use an array $L[1..n]$ to store the level sequences of rooted trees, and $par[1..n]$ to store the position of the parent of each node i. So, $q = par[p]$. By updating array $par[]$ when updating array $L[]$, we can reduce $p - q$ to 1. So we only need to amortize $n - p + 1$ over all rooted trees.

The simplified Pascal implementation of the Beyer-Hedetniemi algorithm is shown in Figure 2.5.

The average number of steps s required to generate single tree is given by the following formula:

$$s = \frac{1}{r_n} \sum_{T \in \mathcal{R}_n} (n - p(T) + 1) \tag{2.3}$$

where $p(T)$ is the p, defined above, associated with rooted tree T.

In [5], it was shown that $s \leq 2$. But Kubicka[11] refined the bound to 1.5113.

```
procedure NEXT
var
  i,q : integer
begin
{r1}   while L[p] = 1 do p:= p-1
{r2}   if p = 1 then done:= true else
{r3}   if (L[p] = 2) and (L[p-1] = 1) then begin
{r4}      L[p] := 1; par[p] := par[p-1]; printit
{r5}   end else begin
{r6}      q := p - par[p]
{r7}      for i:=p to n do begin
{r8}         L[i] := L[i-q]
{r9}         if par[i-q] < p-q then
{r10}            par[i] := par[i-q]
{r11}        else
{r12}            par[i] := q + par[i-q]
{r13}     end
{r14}     p := n
{r15}     printit     { print out the tree }
{r16}  end
end;
```

Figure 2.5: Simplified Pascal implementation of the Beyer-Hedetmieni algorithm.

Chapter 3

A New Recursive Algorithm for Generating Rooted Trees

3.1 A Recursive Algorithm

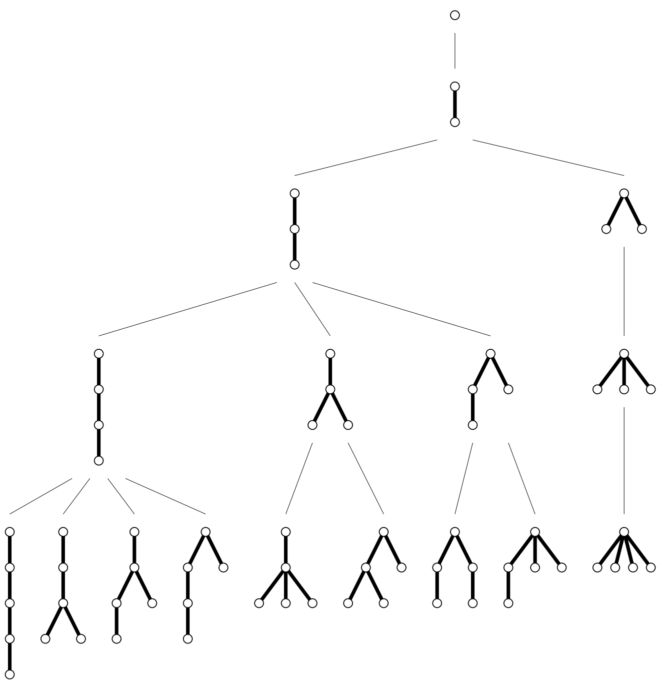

Figure 3.1: The tree of rooted trees(up to 5 nodes) $\mathcal{R}_5$

Before we present our recursive algorithm, let us first consider Figure 3.1. Figure 3.1 is a tree, we call it $\mathcal{R}_5$, formed by all unlabeled rooted trees of at most 5 nodes. The rooted trees at the same level in $\mathcal{R}_5$ are of the same size, and organized in relex order (if represented by their canonic level sequences). Let $\mathcal{R}_n$ be such tree of rooted trees with at most n nodes. In such a tree-of-trees $\mathcal{R}_n$, the rooted tree T of size $m < n$ is a parent of rooted tree T' of size $m+1$ if and only if T is obtained by removing node $m+1$ from T'. So, the only rooted tree of size 1 will be the root of $\mathcal{R}_n$.

Observe that, given any canonic rooted tree T of size m, we can obtain its parent by removing the last node m, and obtain its children by adding node $m+1$ as the rightmost child of some node on the rightmost path of T. We call these node in T *parent candidates* of new node $m+1$. We call children of T in $\mathcal{R}_n$ the *canonic extensions* of T. It seems that some nodes on the rightmost path of T can not be the parent candidates of node $m+1$. For example, the node 4 in the second rooted tree at the fourth level of $\mathcal{R}_5$ in Figure 3.1 can not be the parent candidate of node 5. To generate children of T in relex order, we can simply add node $m+1$ as the rightmost child of the lower level parent candidates first.

Based on above observations, we have the recursive algorithm *GenRooted()* (see Figure 3.2).

procedure GenRooted(T)
{L1}**if** $|T| \geq n$ **then** output the tree
{L2}**else**
{L3}   **for** each parent candidate p of node $|T|$ in T **do begin**
{L4} $T := T$ by adding node $|T|+1$ as the child of p;
{L5}   GenRooted(T)
{L6}   **end**
end; { of GenRooted}

Figure 3.2: Pseudocode of the recursive algorithm for generating rooted trees

The algorithm *GenRooted*() starts from the smallest rooted tree with only one node. It can generate rooted trees in lexicographic order or relex order depending on the order of parent candidates in line {L3}. We choose to follow the Beyer-Hedetniemi algorithm to generate rooted trees in relex order. $\mathcal{R}_5$ becomes the computation tree of the algorithm GenRooted() when $n = 5$. GenRooted() actually traverses the computation tree $\mathcal{R}_n$ in preorder.

Now, the problem becomes how to choose those parent candidates so that the resulting rooted tree after adding the new node preserves the canonic property. Observe that if p is a parent candidate, then the parent of p is also a parent candidate (see the next section for more details). Then we only need to know, for a canonic rooted tree T with n nodes, how to add a new node to get its biggest canonic extension.

To solve this problem, we adopt the idea from [5] (see also Chapter 2 and 4) – we apply the so called *copy strategy* (see next section for more details) to help find such a parent candidate which leads to the biggest canonic extension of the rooted tree T. To implement the copy strategy, we have to know which subtree to copy, i.e., the root and the size of that subtree.

Let s be the root of the subtree to copy, and cL be the size of the subtree. The algorithm GEN() in Figure 3.3 implements the copy strategy: initially, s and cL are set to 0. This helps us to initialize the biggest canonic rooted tree of size n, which is a chain of length $n-1$: when $s = 0$, we add a new node to the very end of the chain (see line {R3}). After the chain of length $n-1$ is built, s and cL will get their first non-zero values in lines {R9} and {R11}, respectively.

Given any rooted tree T of size $p-1$, to generate the biggest extension of T, we follow the current s and cL to continue copying the previously chosen subtree (see line {R4-R7}), s and cL will not be changed. To generate all other extensions of T, we add node p as a child of its current grandparent $par[par[p]]$ (see line {R10}). we set $s = par[p]$ and $cL = p - par[par[p]]$ (see lines {R9} and {R11}, respectively) for further use.

3.2 Proof of Correctness

First, we define some terms. Nodes with the same parent are *siblings*. If $c_1 < c_2 < \cdots < c_m$ are all the siblings of node p, and i is the largest index for which $c_i < p$, we say the c_1 is the *leftmost sibling* of p, and c_i the *rightmost younger sibling* of p. We say tree T is a *prefix* of tree T' if lev_T is a prefix of $lev_{T'}$.

```
procedure Gen( p, s, cL : integer);
begin
{R1}  if p>n (*or ((par[p-1] = 1) and (par[p-2] = 1))*) then PrintIt
{R2}  else begin
{R3}    if cL = 0 then par[p] := p-1 else <- initialize first tree
{R4}    if par[p-cL] < s                  <- gen biggest extension
{R5}       then par[p] := par[s]
{R6}       else par[p] := cL + par[p-cL];
{R7}    Gen( p+1, s, cL );
{R8}    while par[p] > 1 do begin         <- gen other extensions
{R9}       s := par[p];                   <- find new subtree to copy
{R10}      par[p] := par[s];
{R11}      Gen( p+1, s, p-s )
{R12}   end
{R13} end
end; {of Gen}
```

Figure 3.3: Recursive CAT algorithm for generating rooted trees.

Given two sequences T_1 and T_2, we write $T_1 \prec T_2$ if and only if T_1 is lexicographically less than T_2. We show in Lemma 3.2.1 that the level sequence and the parent array are equivalent in terms of lexicographic ordering. For canonic rooted trees T and S, we write $T \prec S$ if $par_T \prec par_S$; we write $T \equiv S$ if $par_T = par_S$.

Lemma **3.2.1** *For canonic trees T and S of the same size, $par_T \preceq par_S$ if and only if $lev_T \preceq lev_S$.*

Proof:By induction on the size. It is trivial to check the lemma holds for trees of size 1.

Assume that the lemma holds for all canonic rooted trees of size $n-1$. Suppose T and S are canonic rooted trees of size n. Let $\langle par_T[1..n]\rangle$ and $\langle par_S[1..n]\rangle$ be the parent arrays of trees T and S respectively; $\langle lev_T[1..n]\rangle$ and $\langle lev_S[1..n]\rangle$ be the level sequences of T and S respectively. If $par_T \prec par_S$ then either $\langle par_T[1..n-1]\rangle \prec \langle par_S[1..n-1]\rangle$ or $\langle par_T[1..n-1]\rangle = \langle par_S[1..n-1]\rangle$ and $par_T[n] < par_S[n]$. In the first case, we have $\langle lev_T[1..n-1]\rangle \prec \langle lev_S[1..n-1]\rangle$ which implies that $lev_T \prec lev_S$ by definition of lexicographic order. The second case implies that T and S are the same if node n is removed from both trees, and the parent of node n in T is a proper ancestor of the parent of n in S. So, we have $lev_T[n] < lev_S[n]$ which implies that $lev_T \prec lev_S$. Similarly, we can prove that if $lev_T[n] \preceq lev_S[n]$ then $par_T[n] \preceq par_S[n]$. $\square$

We say that subtrees, T_1 and T_2, of T are *comparable subtrees* if the roots of these two trees are siblings. Recall that $T(p)$ denotes the subtree of T rooted at node p.

Since the trees will be generated in relex order, we shall produce the lexicographically biggest tree first in the process of constructing a tree with n nodes based on a tree with $n-1$ nodes already generated.

PROPOSITION **3.2.1** *Given a canonic tree T, if $p_1 < p_2 < \ldots < p_k$ is a sequence of consecutive children of a node p in T, then $T(p_i) \succeq T(p_{i+1})$ for $1 \le i \le k$.*

Now suppose that we have a tree T with $n-1$ nodes and want to extend to a tree with n nodes. Because of the preorder labeling, node n must be the child of some ancestor of node $n-1$ (a node is considered to be an ancestor of itself). That is to say, node n should be added as a rightmost child of some node on the rightmost path of the current tree.

PROPOSITION **3.2.2** *If $\langle par[1..n-1], par[n]\rangle$ is canonic and $par[n] \neq 1$, then $\langle par[1..n-1], par[par[n]]\rangle$ is also canonic.*

Suppose $root = p_1, p_2, \ldots, p_k = n-1$ is the rightmost path of a canonic tree T of size $n-1$. We say that p_i is a *valid position* if we add a child to p_i, and the new tree is still canonic. If p is a valid position, then by Proposition 3.2.2, all the ancestors are valid positions too.

LEMMA **3.2.2** *If T is a canonic tree with $n-1$ nodes, then the valid positions of T will produce all possible canonic trees with n nodes and T as their prefix*

PROOF:Any canonic tree of size n with T as its prefix must have its node n added as the child of an ancestor p of node $n-1$. Hence p is a valid position because it is on the rightmost path of T, and the new tree with n nodes is canonic. $\square$

Obviously, the lexicographically largest extension of a tree T is created by adding a child to the valid position with biggest level(farthest from the root). Now, we wish to know how to determine this position. As a matter of fact, all nodes on the rightmost path are candidates for valid positions, and if node $n-1$ is valid, then it is the one with the greatest level.

Let p_i be as defined above, and c_i is rightmost younger sibling of p_i (a node is not a sibling of itself), then by Proposition 3.2.1, $T(c_i) \succeq T(p_i)$.

Proposition **3.2.3** *T' is a canonic extension of T by adding node n as $par[n] = p_i$ if and only if $T'(p_j) \preceq T'(c_j)$ for all $j \geq i$.*

Proposition **3.2.4** *Let T be a canonic tree, and let $a_1, a_2, \ldots, a_m$ be consecutive children of the root of T. Suppose i is the smallest index such that $T(a_i) = T(a_{i+1}) = \cdots = T(a_{m-1})$ and $T(a_m)$ is a prefix of $T(a_{m-1})$. Let L_j be the subsequence in lev_T associating with $T(a_j)$ for $1 \leq j < i$, L for $T(a_k)$ where $i \leq k < m$, and L' for $T(a_m)$. An extension T' of T, $\langle 0, L_1, \ldots, L_{i-1}, L, \ldots, L, L', b\rangle$, is canonic and lexicographically largest if $\langle L', b\rangle$ is a prefix of L when $T(a_{m-1})$ is bigger in size than $T(a_m)$, or $\langle b\rangle$ is a prefix of L when $T(a_{m-1}) = T(a_m)$.*

The above proposition tells that we can *copy* the rightmost comparable subtree to expand to a greatest tree in lex order if the subtree that contains node $n-1$ is a prefix of its rightmost comparable subtree. But such expansion will not always preserve the canonicity if there is more than one subtree containing $n-1$ which are prefixes of their rightmost comparable subtrees. The following definition is important in finding the right comparable subtree (if this tree has been repeatedly copied more than once, we choose the first one) to produce the greatest in lex order and preserve the canonicity. We call the root of this subtree the *critical node*.

Definition **3.2.1** *Let q be the smallest ancestor (a node is an ancestor of itself) of $n-1$ such that $T(q)$ is a prefix of $T(c)$, where c is the rightmost proper sibling of q. Then the critical node s is the leftmost sibling of c such that $T(s) \equiv T(c)$. $cL = |T(s)|$ is the size of $T(s)$.*

The Critical node s is well-defined for all rooted trees except the greatest tree $\langle 0, 1, \ldots, n-1 \rangle$, since the siblings of the first ancestor with at least one sibling could be the candidates for the critical node. For the exceptional case, i.e. the greatest canonic tree (which is actually a chain), we let $s = 0$ and $cL = 0$. We will not do any copying, we just add node n as the child of node $n-1$.

Observe that a tree is canonic if and only if all its subtrees are canonic.

Lemma **3.2.3** *The tree T' of size n extended, by copying subtree $T(s)$, where s is the critical node, from T of size $n-1$ is canonic and lexicographically greatest among all rooted trees of size n.*

Proof:Let s be as defined in the Definition 3.2.1. Since we are actually copying subtree $T(s)$, the subtree $T'(par(s))$ is canonic and lex largest with prefix $T(par(s))$ by Proposition 3.2.4. Let c be rightmost younger sibling of $par(s)$, then we know that $T(par(s)) \prec T(c)$ and $T(par(s))$ is not a prefix of $T(c)$ because of the minimality of s. So, $T'(par(s)) \prec T'(c)$ where $T'(c) = T(c)$. Similarly, we can prove that all subtrees rooted at ancestors of s,which are all valid positions, are canonic (including the root of T'), hence T' is canonic.

Since $T'(par(s))$ is the lex largest extension of $T(par(s))$, any larger extension of tree T will definitely enlarge $T(par(s))$ further because the newly added node n in T' must be a descendant of $par(s)$ on the rightmost path. This is contrary to the fact that $T'(par(s))$ is the lexicographically largest extension of $T(par(s))$. Hence T' is the lex largest extension of T. □

Suppose we are given any prefix $\langle par[1..k] \rangle$, then to generate the parent arrays of all rooted trees with n nodes, we divide the situation into two cases:
case 1: no trees with this prefix have been generated yet;
case 2: all trees with this prefix have been generated.

Copying Strategy:

{Rule1} For case 1, use the current s and cL to continuously copy subtree $T(s)$ of size cL.

{Rule2} For case 2, set $s = par[k]$, and $par[k] = par[s]$

if $par[k]$ is not the root of the tree T.

Note that the first tree $\langle 0, 1, 2, \ldots, n-1 \rangle$ will be initialized by the program. There is no doubt that it is the lexicographically greatest tree of size n. For this case, critical node s remains the same as in the initial call, $s = 0$ and $cL = 0$.

LEMMA **3.2.4** {Rule2} *in the Copying Strategy correctly updates the critical node s for any given prefix.*

PROOF:For the given prefix(which is a tree T) $\langle par[1..k-1], par[k] \rangle$, either there exists an ancestor $p \neq k$ of k such that $T(p)$ is a prefix of its rightmost sibling, or there exists no such p. Let $a = par[k]$ and $b = par[par[k]]$.

The second case implies that for any ancestor q of k, $T(q) \prec T(c)$ where c is the rightmost sibling of q. After the updating $par[k] = par[par[k]]$, we get T',$\langle par[1..k-1], par[par[k]] \rangle$. For any ancestor q of a, $T'(q) \prec T(q)$, hence there exists no q such that $T'(q)$ is a prefix of $T'(c)$ where c is its rightmost sibling. So, only $T'(k)$, the tree with only one node k, is a prefix of $T(a)$, where a is its rightmost sibling. And there is no sibling c' of a such that $T'(a) = T'(c')$ since a is a valid position for $\langle par[1..k-1] \rangle$. So $s = a$ for prefix $\langle par[1..k-1], par[par[k]] \rangle$.

For the first case, any ancestor p of node k in T' has no sibling c such that $T'(p)$ is a prefix of $T'(c)$, even though $T(p)$ might be the prefix of the tree rooted at p's rightmost sibling, because $T'(p) \prec T(p)$. Similarly, only $T'(k)$ is the prefix of the subtree $T'(a)$, and $s = a$. □

Pascal code is shown in Figure 3.3 as the procedure *Gen(p,s,cL)*. The procedure produces all parent arrays with prefix $\langle par[1..p-1] \rangle$. The initial call is *Gen(1,0,0)*; no initialization is necessary.

THEOREM **3.2.1** *If $\langle par[1..p-1] \rangle$ is the parent array of a canonic tree T, the critical node of T is s, and $cL = |T(s)|$, then the call* Gen(p,s,cL) *generates all canonic trees of size n whose parent array has prefix $\langle par[1..p-1] \rangle$.*

Proof: We shall prove the theorem by induction on decreasing values of p for fixed n. The theorem is true if $p > n$ because $\langle par[1..n]\rangle$ is the only tree of size n with the prefix of itself. Line $\{R1\}$ in procedure *Gen()* will simply output the tree.

Assume for any $p+1 \leq n$, the theorem holds. We want to show that that *Gen(p,s,cL)* will generate all canonic trees with prefix $\langle par[1..p-1]\rangle$. If the tree is the first tree in the list, i.e., $cL = 0$, we just append node p as child of $p-1$. This is the lex largest tree. Otherwise, by Lemma 3.2.4, s and cL have been correctly updated by previous calls at line $\{R9\}$ and $\{R11\}$ respectively, and Lemma 3.2.3 guarantees the valid position with the biggest level will be generated first at lines $\{R4 - R7\}$ by applying $\{Rule1\}$. By Lemma 3.2.2, all possible extensions to trees of size p will be generated at lines $\{R8 - R12\}$ by applying $\{Rule2\}$. Then, by our assumption, all these extensions will be wholly expanded to canonic trees of size n. $\square$

3.3 Complexity Analysis

We now argue that the algorithm is CAT. Observe that every iteration of the while loop results in a recursive call, and that the total amount of computation is proportional to the number of recursive calls. Let r_n denote the number of rooted trees with n nodes. The number of recursive calls is $r_1 + r_2 + \cdots + r_n$.

From Knuth [6], pg. 396:

$$r_n \sim 0.43992...(2.95576...)^n n^{-3/2} \tag{3.1}$$

Let $T(x) = \sum_{i\geq 0} r_i x^i$ be the ordinary generating function of the sequence $\{r_n\}$. It's radius of convergence is $\rho \approx 0.3383219$. Kubicka [11] shows that asymptotically

$$\frac{1}{r_n}\sum_{i=0}^{n} r_i \sim \frac{1}{1-\rho} \approx 1.5113.$$

This constant ratio implies that the algorithm is CAT.

The preceding proof relied on the asymptotic expression (3.1). By slightly modifying the algorithm, we can prove, by completely elementary means, that the algorithm is CAT. In the computation tree $\mathcal{C}_n$, nodes with one child only occur if the corresponding rooted trees have two successive nodes whose parent is the root; i.e., if $par[p-1] = par[p-2] = 1$. Removing the comment delimiters *(** and **)* at line $\{R1\}$ eliminates nodes with only one child. The computation tree now has more leaves than internal nodes and so clearly the underlying algorithm is CAT, since only a constant amount of computation is done at each node. Ruskey [8] calls this the Path Elimination Technique (PET).

3.4 Generating Rooted Trees with Height Restrictions

By slightly modifying the procedure *Gen()* of Figure 3.3, we can also generate trees with height restrictions, without losing the CAT property. The computation trees of these modified algorithms are certain subtrees of $\mathcal{R}_n$, the computation tree of the original recursive rooted tree algorithm in Figure 3.3.

To generate all trees of height at least h, initialize *par[1..h+1]* to be $0, 1, \ldots, h$ and then call *Gen(h+2,0,0)*. To generate all trees of height exactly h, initialize *par[1..h+1]* to be $0, 1, \ldots, h$ and then call *Gen(h+2,h+1,1)*. To generate all trees of height at most h change the test $cL = 0$ at line $\{R3\}$ to

```
(cL = 0) and (p <= h+2)
```

and ignore the first tree generated (which has height $h+1$ and acts as the initialization of the recursive construction).

In all three cases the resulting algorithms are CAT. Recall that $\mathcal{R}_n$ is the computation tree for the algorithm in Figure 3.3. Let $\mathcal{R}_{>h}$ be computation tree for the algorithm of generating trees with height at least h, and $\mathcal{R}_{=h}$ for the algorithm of generating trees with height h, and $\mathcal{R}_{<h}$ for the algorithm of generating trees with height at most h. Since each node in the computation trees are associated with a recursive call *Gen(p,s,cL)*, we label each node of $\mathcal{R}_n$ by (p, s, cL). So the root of $\mathcal{R}_n$ will be $(1, 0, 0)$.

h	1	2	3	4	5	6	7	8	9	10	11	12	13
$n = 9$	1	21	76	93	61	26	7	1					
$n = 10$	1	29	147	225	180	94	34	8	1				
$n = 11$	1	41	277	528	498	308	136	43	9	1			
$n = 12$	1	55	509	1198	1323	941	487	188	53	10	1		
$n = 13$	1	76	924	2666	3405	2744	1615	728	251	64	11	1	

Table 3.1: The numbers of rooted trees of various heights.

The computation tree $\mathcal{R}_{>h}$ is the subtree of $\mathcal{R}_n$ rooted at $(h+2, 0, 0)$; $\mathcal{R}_{=h}$ is the subtree of $\mathcal{R}_n$ rooted at $(h+1, 0, 0)$ minus the subtree of $\mathcal{R}_n$ rooted at $(h+2, 0, 0)$; $\mathcal{R}_{<h}$ is the subtree of $\mathcal{R}_n$ obtained by deleting the subtree of $\mathcal{R}_n$ rooted at $(h+2, 0, 0)$.

By applying the PET technique as above to eliminate those nodes with one child, the resulting computation trees, $\mathcal{R}_{>h}, \mathcal{R}_{=h}, \mathcal{R}_{<h}$, will have more leaves than internal nodes. This is because the subtree deletion of $\mathcal{R}_{=h}$ and $\mathcal{R}_{<h}$ can introduce at most one node with a single child.

3.5 Generating Rooted Trees with Parenthood Restrictions

In this section we consider rooted tree where each node has at most k children. Algorithms for the case of $k = 2$ has been developed previously by Pallo in [15], by Ruskey in [9] and by Kubicka and Kubicki [14]. Our strategy is the same as in the rooted tree case; that is, we develop a recursive algorithm whose computation tree is that subtree of $\mathcal{R}_n$, call it $\mathcal{B}_{n,k}$, containing only rooted tree T so that $|T| \leq n$ and the number of children of each node in T is less than or equal to k. Figure 3.4 shows $\mathcal{B}_{6,2}$. An important difference from $\mathcal{R}_n$ is that in $\mathcal{B}_{n,k}$ there can be rooted trees at levels less than n with no children. The three nodes in $\mathcal{B}_{6,2}$ boxed with dotted lines in Figure 3.4 are wasted steps, where two of them have no children.

Let b_n be the number of binary unordered trees. These numbers satisfy the recurrence relation $b_0 = b_1 = 1$ and for even $n = 2k$,

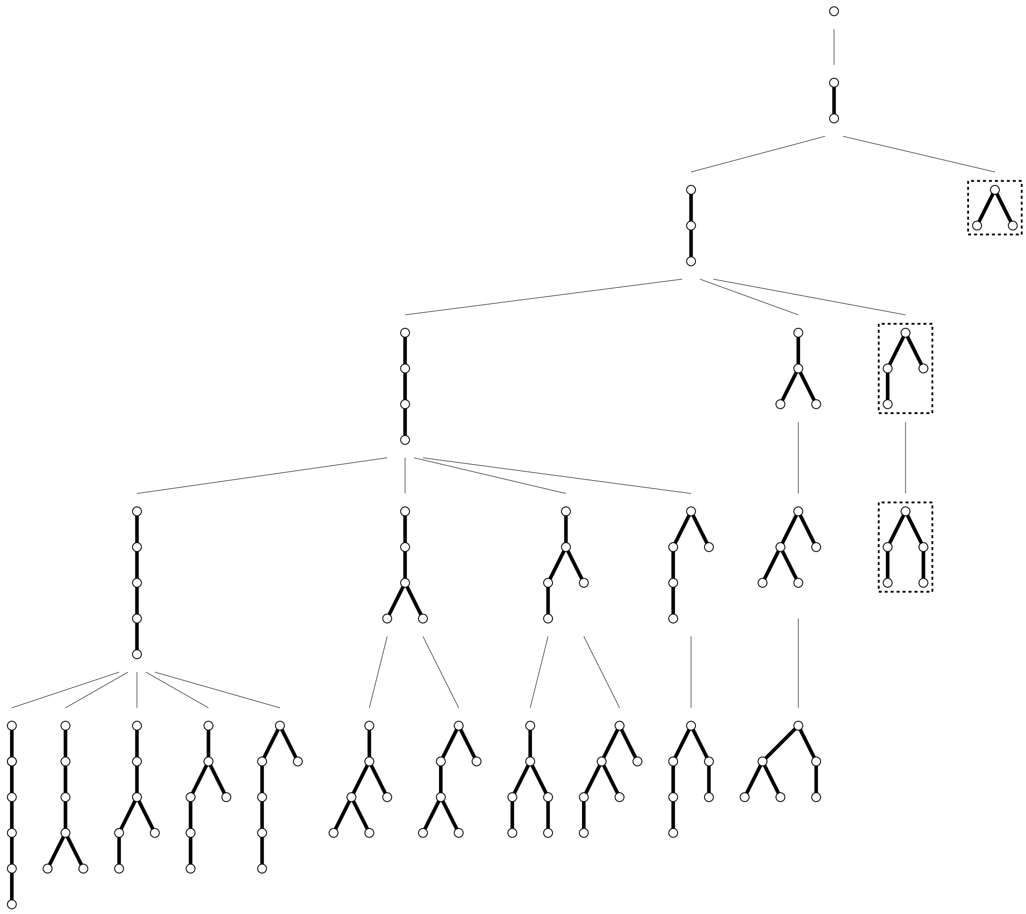

Figure 3.4: The tree of rooted "binary" trees (up to 6 nodes).

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
r_n	1	1	1	2	4	9	20	48	115	286	719	1842	4766	12486	32973
$b_{n,2}$	1	1	1	2	3	6	11	23	46	98	207	451	983	2179	4850
$b_{n,3}$	1	1	1	2	4	8	17	39	89	211	507	1238	3057	7639	19241
$b_{n,4}$	1	1	1	2	4	9	19	45	106	260	643	1624	4138	10683	27790

Table 3.2: Number $b_{n,m}$ of rooted trees of size n with at most m children.

$$b_{2k} = b_0 b_{2k-1} + b_1 b_{2k-2} + \cdots + b_{k-1} b_k,$$

and for odd $n = 2k+1$,

$$b_{2k+1} = b_0 b_{2k} + b_1 b_{2k-1} + \cdots + b_{k-1} b_{k+1} + \binom{b_k + 1}{2},$$

from which we may compute the numbers in Table 3.2.

Sloane's sequence database refers to these as the Wedderburn-Etherington numbers [34], [35]. Their ordinary generating function satisfies the functional equation $2B(x) = 2 + x[B^2(x) + B(x^2)]$. Asymptotically (from Comtet [36], pg. 55),

$$b_n \sim 0.7916...(2.48325354...)^n n^{-3/2}.$$

From the preceding expression it follows that

$$\sum_{i=1}^{n} b_i = O(b_n).$$

A naive approach is to introduce an array *chi[1..n]* whose ith entry is the number of children of node i. Replace each of the recursive calls *Gen(p+1,s,cL)* at lines $\{R7\}$ and $\{R11\}$ by the three lines

```
chi[par[p]] := chi[par[p]] + 1;
if chi[par[p]] <= k then Gen(p+1,s,cL);
chi[par[p]] := chi[par[p]] - 1;
```

The result is an algorithm whose running times are CAT for realistic values of n (i.e., on average there is at most 3 iterations of the loop for $n \leq 25$). There is some redundancy in the algorithm: because if any node have k or more children, the algorithm will still check them to see if it is possible to attach a new child to the node.

We may obtain an algorithm that is provably CAT by maintaining an array *jump[1..n]* where *jump[i]* is the closest ancestor of node i with less than k children, and an array *rchild[1..n]* where *rchild[i]* is the rightmost child of node i.. Array *jump* is the parent array of a certain "subtree" of T generated by the algorithm. See Appendix B for a Pascal implementation of *jump[]*.

3.6 Final Remarks

The recursive algorithm for generating rooted trees we presented in this chapter is not only much simpler than the Beyer-Hedetniemi iterative algorithm, it is also much more flexible in the sense that it can be easily modified to generate rooted trees so that they are of size between n_1 and n_2, each node has at most m children, and the height of the trees is between lb and ub (see Appendix B for its Pascal implementation). We have proved that this recursive algorithm is a CAT algorithm.

We did a small experiment on the actual running time of the simplified Beyer-Hedetniemi's algorithm and our recursive algorithm. The results were in Table 3.3. Both algorithms were implemented in Pascal and tested on the same machine. It shows that our recursive algorithm is faster than the iterative one.

n	trees	Iterative	time/tree	Recursive	time/tree
9	286	1940	6.783	1490	5.210
10	719	4880	6.787	3660	5.090
11	1842	12700	6.894	9300	5.049
12	4766	32200	6.756	23300	4.888
13	12486	84800	6.791	60900	4.877
14	32973	221000	6.702	157000	4.761
15	87811	591000	6.730	415000	4.726
16	235381	1560000	6.628	1100000	4.673
17	634847	4260000	6.710	2950000	4.647
18	1721159	11550000	6.711	7980000	4.636
19	4688676	31960000	6.816	21760000	4.641

Table 3.3: The running time(in microseconds) comparison of BH's iterative and our new recursive algorithm.

Chapter 4

Previous Algorithms for Generating Free Trees

4.1 An Introduction

A *free tree* is a connected graph without cycles. The generation of unlabeled free trees is more complicated than that of rooted trees due mainly to the absence of the root. In this chapter, we will first count the free trees. We will then give a brief discussion of previous algorithms for generating free trees. We will provide a detailed description of Wright, Richmond, Odlyzko and McKay's [18] algorithm. It is related to our new recursive algorithm which will be presented in the next chapter.

4.2 Counting the Free Trees

We discussed in Chapter 2 that free trees can be easily represented as a rooted tree by picking a node as the root. Enumeration of free trees is also related to rooted trees.

The number r_n of unlabeled rooted trees is given by the formula (2.1).

Let F be any free tree with n nodes, and r a node in F. Recall that T_r is the corresponding rooted version of T with r as root. Suppose there are k subtrees of the root r in T_r, with $s_1, s_2, \ldots, s_k$ nodes in these respective subtrees. So,

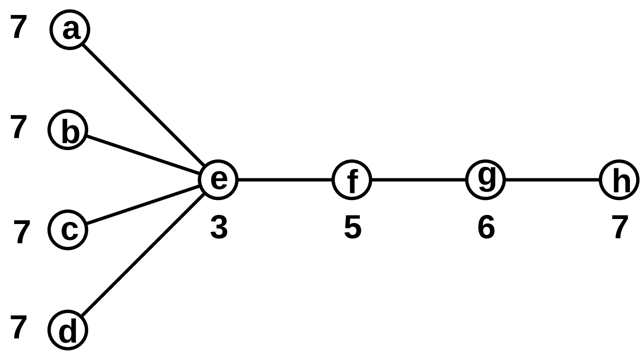

Figure 4.1: Weights of the nodes in a free tree.

$\sum_{1 \leq i \leq k} s_i = n - 1$. In such circumstances, we say that the *weight* of r, $weight(r)$, in F is $max(s_1, s_2, \ldots, s_k)$. Thus in the tree in Figure 4.1, the node e has weight 3, and node f has weight $max(5, 2) = 5$.

A node in a free tree with minimum weight is called a *centroid* of the free tree. Note that the centroid of a free tree is not necessarily a center of the free tree. For example, in Figure 4.1, the centroid is e, and the center is f.

Let r and $s_1, s_2, \ldots, s_k$ be as above, and let $t_1, t_2, \ldots, t_k$ be the roots of the subtrees emanating from r. Obviously, the weight of t_1 is at least $n - s_1 = 1 + s_2 + \cdots + s_k$.

If there is a centroid c, we have

$$weight(r) = max(s_1, s_2, \ldots, s_k) \geq weight(c) \geq 1 + s_2 + \cdots s_k,$$

and this implies $s_1 > s_2 + \cdots + s_k$. A similar result can be derived if we replace t_1 by t_i in the above discussion. So at most one of the subtrees can contain a centroid. This condition implies the following proposition:

PROPOSITION **4.2.1** *Any free tree F has either one or two adjacent centroids.*

Conversely, if $s_1 > s_2 + \cdots + s_k$, there is a centroid in the subtree $T(t_1)$, since

$$weight(t_1) \leq max(s_1 - 1, 1 + s_2 + \cdots + s_k) \leq s_1 = weight(r),$$

and the weight of all nodes in the subtrees $T(t_2), T(t_3), \ldots, T(t_k)$ is at least $s_1 + 1$. Now we have the following proposition:

PROPOSITION **4.2.2** *A node r in a free tree F is the only centroid if and only if*

$$s_j \leq (\sum_{1 \leq i \leq k} s_i) - s_j, \ \textit{for } 1 \leq j \leq k. \tag{4.1}$$

We now have that the free trees with one centroid is the number of rooted trees minus those rooted trees that violate the condition in equation (4.1). The number of free trees with one centroid therefore comes to

$$r_n - \sum_{\substack{1 \leq i \leq j \\ i+j=n}} r_i r_j.$$

Now, what is the number of free trees with two centroids? Since the weights of the two centroids is equal, n must be even and they each are weighted at $n/2$. Let $n = 2m$. To form a bicentroidal free tree, we could just choose two rooted trees of size m with repetition and connect two roots by an edge. This gives the number of bicentroidal free trees:

$$\binom{r_m + 1}{2}.$$

Thus the total number of unlabeled free trees is

$$f_n = r_n - \sum_{\substack{i \leq j \\ i+j=n}} r_i r_j + \frac{(1 - (-1)^{n-1})}{2} \binom{r_{n/2} + 1}{2}. \tag{4.2}$$

The equation (4.2) above suggest a simple generating function for the number of unlabeled free trees (see [6]):

$$\begin{aligned} F(x) &= A(x) - \tfrac{1}{2}A(x)^2 + \tfrac{1}{2}A(x^2) \\ &= x + x^2 + x^3 + 2x^4 + 3x^5 + 6x^6 + 11x^7 + 23x^8 + \cdots \end{aligned}$$

where $A(x)$ is the generation function for r_n.

We can also use equation (2.1) to compute the number f_n of unlabeled free trees of size n (see Table 4.1).

Equation (4.2) gives no clue how to generate free trees, but it does show the relation between rooted trees and free trees. Some algorithms were developed based

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
f_n	1	1	1	2	3	6	11	23	47	106	235	551	1301	3159	7741

Table 4.1: The number f_n of free trees with n nodes

on the idea that the set of unlabeled free trees is the subset of unlabeled rooted trees if free trees are represented by rooted trees as discussed before.

4.3 Generating Free Trees

Algorithms for generating various labeled or unlabeled rooted trees have been extensively investigated in the past, but to our knowledge few results have been published on generating unlabeled free trees (even though a significant amount of work has been done on the generation of labeled free trees). It is easier to generate bicentral and unicentral free trees separately. In [16], Scions wrote: "However, it has proved impossible so far to generate the interleaved set of unicentral and bi-central [free] trees except by carrying out a comparison between the next central tree and the next bi-central tree, which is essentially a sorting operation which we have previously avoided."

Read [1] did some early work in 1970. He formulated an algorithm for generating all free trees of size n. Unfortunately, since his algorithm must process and store all trees with $n-1$ nodes, the space required grew exponentially. Kozina [19] introduced a coding method using linear space and derived from it an algorithm for generating rooted trees and free trees. His free tree algorithm generated unicentral free trees and bicentral free trees separately. The running time is $O(nr_n)$ for rooted trees, $O(n^2r_n)$ for free trees since after generating each rooted tree, a checking procedure is needed to output only valid free trees. In 1981, Wilf [17] gave an algorithm for random generation of unlabeled free trees. But no systematic ways of generation all free trees was provided.

In 1981, Liu [24] published a paper on generating rooted trees and free trees lexicographically. He first introduced an algorithm for generating rooted trees. Then this algorithm was extended to generate free trees.

He represented a rooted tree by a sequence of non-negative integers when the integer at position i gave the number of children it had. The nodes were labeled by a preorder traversal of the tree as we discussed in Chapter One. Two trees were equivalent if one can be obtained by rearranging the order of the subtrees of the other one. The representative of each equivalence class was the lexicographically largest sequence in this class. The algorithm was iterative. Given a sequence, it scanned the sequence from right to left to find the first node p with more than one child, reduced the number of children of p by one, and then made the rest of the sequence lexicographically largest. The algorithm outputed all valid representations of unlabeled rooted trees lexicographically from largest one, $(n-1, 0, 0, \ldots, 0)$, to smallest one, $(1, 1, \ldots, 1, 0)$.

A free tree was represented by a rooted tree where the root of the rooted tree was a center of the free tree. For bicentral free trees, the root was picked so that the resulting rooted tree was lexicographically bigger. Note that the representation of rooted tree mentioned above was always referred to as the lexicographically biggest one in its equivalence class.

The algorithm introduced by Liu for generating free trees was an extension of the one for rooted trees. The algorithm will generate all rooted trees. Only those valid representation of free trees will be output.

The complexity of Liu's rooted tree algorithm was $O(nr_n)$, where r_n is the number of unlabeled rooted trees. The complexity of his free tree algorithm was $O(n^2 r_n)$ since all rooted trees would actually been generated in $O(nr_n)$ time and examined in $O(n)$ time. Note that $r_n/f_n = O(n)$ where f_n is the number of unlabeled free trees, so the average running time for each tree is $O(n^2)$.

In 1984, Tinhofer and Schreck [26] presented an algorithm for generating all unlabeled free trees as an application of their new method of coding unlabeled free trees. Their algorithm for coding the free trees is $O(n)$ where n is the size of the free trees.

The average length of their coding sequence is $0.84n$, the algorithm ran in $O(nf_n)$ time where f_n is the number of free trees with n nodes, and the resulting list of coding sequences were not in lexicographic order. The average running time for each tree is $O(n)$.

In 1985, Wright, Richmond, Odlyzko and Mckay [18] extended Beyer and Hedetniemi's [5] algorithm for generating rooted trees to generating unlabeled free trees. The algorithm generates each tree in constant average time, independent of the size of the trees. We will discuss their algorithm in detail in the following section.

4.4 An Iterative Algorithm

An algorithm of Beyer and Hedetniemi[5] for generating rooted unlabeled trees was extended to generate unlabeled free trees by Wright,Richmond, Odlyzko and McKay [18]. We therefor call their algorithm the *WROM Algorithm*. All nonisomorphic trees of a given size are generated, without repetition, in time proportional to the number of trees.

The WROM algorithm adopts canonic level sequences for rooted trees as the representation of unlabeled free trees (see also the discussion in Chapter 1 on representation of free trees). They differ from our representation only on the selection of the root for bicentral free trees. They call the center they pick the *primary root*, and the canonic level sequence of the resulting rooted tree the *primary level sequence*.

For unicentral free trees, our canonic center is their primary root, and our canonic representations of free trees represent the exactly same free trees as their primary level sequences.

For bicentral trees, let c, d be two centers in free tree T, and $Q(c)$ and $R(d)$ be two rooted trees obtained by deleting the edge between c and d from T. Let $|Q(c)|$ and $|R(d)|$ be the number of nodes in Q and R respectively.

For any free tree T, the WROM algorithm's selection of primary root is defined by the following rules:

(R1) if T is unicentral, then c is the root if c is the center of T, or

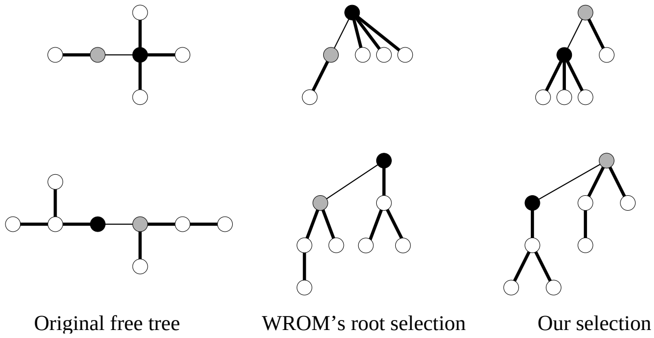

Figure 4.2: Two different ways to select the root.

0 1 2 3 4 1 2 3	0 1 2 2 2 2 1 2
0 1 2 3 3 1 2 3	0 1 2 2 2 1 2 2
0 1 2 3 3 1 2 2	0 1 2 2 2 1 2 1
0 1 2 3 2 1 2 3	0 1 2 2 2 1 1 1
0 1 2 3 2 1 2 2	0 1 2 2 1 2 2 1
0 1 2 3 2 1 2 1	0 1 2 2 1 2 1 2
0 1 2 3 1 2 3 1	0 1 2 2 1 2 1 1
0 1 2 3 1 2 2 2	0 1 2 2 1 1 1 1
0 1 2 3 1 2 2 1	0 1 2 1 2 1 2 1
0 1 2 3 1 2 1 2	0 1 2 1 2 1 1 1
0 1 2 3 1 2 1 1	0 1 2 1 1 1 1 1
	0 1 1 1 1 1 1 1

Figure 4.3: Primary canonical level sequences with $n = 8$.

(R2) if T is bicentral with two centers, c, d, then c is the root if

(R2.1) $|Q(c)| \geq |R(d)|$ and

(R2.2) if $|Q(c)| = |R(d)|$, then $lev_{Q(c)} \succeq lev_{R(d)}$.

This representation is different from ours (see Chapter 1). Our representation is more natural in the sense that we always choose the lexicographically greatest canonic rooted tree as the representation of the free tree. The WROM algorithm's representation is inconsistent since for unicentral free trees they followed this rule, but for bicentral ones they didn't.

From now on in this section, we will always look at the rooted tree representation defined above for any given free tree.

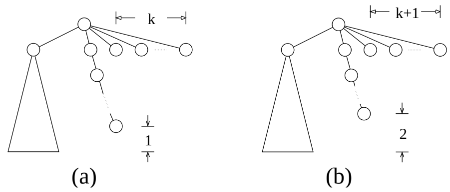

Figure 4.4: (R1) fails: NEXT() generate (b) from (a).

The WROM algorithm trys to use the successor function, NEXT(), in Beyer and Hedetmieni's algorithm (see Figure 2.4) to generate all unlabeled free trees. Since the set of all primary canonic level sequences of free trees is a subset of all canonic level sequences of rooted trees (see [18] for details), only a filter system is needed to be built to skip over those canonic level sequences which are not primary.

Beyer-Hedetmieni's successor function NEXT() will always generate a canonic level sequence. The only problem raised when applying it to the free tree case is that the resulting canonic level sequence may not be primary, i.e., the resulting root may not be a primary root, if we treat the resulting tree as a free tree.

It turns out that there are only three classes of primary level sequences which will be transferred to a non-primary ones by Beyer and Hedetmieni's successor function NEXT():

Class 1: The transformation of primary level sequences in this class will violate only (R1) above. These trees must be a bicentral free tree with two subtrees $Q(c)$ and $R(d)$ (as above) where c is the root, and the leftmost subtree of c in $Q(c)$ is a chain, and the rest of subtrees are one-node subtrees. See Figure 4.4.

Class 2: The transformation of primary level sequences in this class will violate only (R2.1) above. Such a tree must be unicentral free tree, with c as primary root, in which the leftmost subtree of c has more than half of the nodes in the whole tree. See Figure 4.5.

Class 3: The transformation of primary level sequences in this class will violate only (R2.2) above. Such a tree must be a bicentral free tree, with c as primary root, d as the other center, and $|Q(c)| = |R(d)|$, $lev_{Q(c)} = lev_{R(d)}$(for Q, R defined as before). See Figure 4.6.

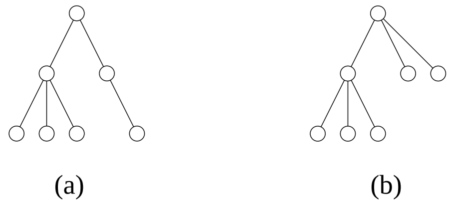

Figure 4.5: (R2.1) fails: NEXT() generate (b) from (a).

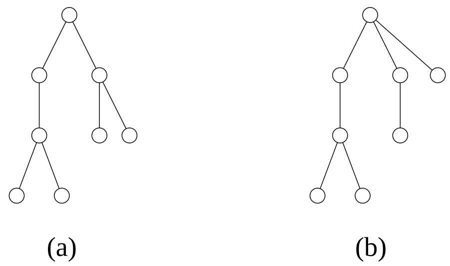

Figure 4.6: (R2.2) fails: NEXT() generate (b) from (a).

The WROM algorithm then modifies the results generated by NEXT(). Figure 4.8 shows some examples of such modifications.

Let T be a free tree with n nodes in one of the classes above, and NEXT(L) denotes the level sequence generated by NEXT() from L. Let $L[i]$ be the level of node i, and $T_1, T_2, \ldots, T_m$ be the subtrees of primary root of T. Let S be the set of all primary level sequences of length n in Class 1, 2 and 3. The WROM algorithm for generating free trees is shown in Figure 4.7 based on above notations.

The WROM algorithm was shown to be very efficient. Table 4.2 shows that the average number of times the WROM algorithm access the positions in the level sequences $L[1..n]$ where n is the size of the rooted trees.

THEOREM **4.4.1** *[18] WROM's algorithm generates unlabeled free trees in Constant Amortized Time (CAT).*

if $L \in S$ **then begin**
$L := next(L)$ where p initialized to $|T_1| + 1$;
if $L[|T_1| + 1] > 1$ **then** $L[n - h + 1 \ldots n] := 1, 2, \ldots, h$;
else $L := next(L)$;

Figure 4.7: The WROM algorithm

n	trees	cost	cost/tree	n	trees	cost	cost/tree
4	2	7	3.500	13	1301	4558	3.503
5	3	11	3.667	14	3159	10709	3.390
6	6	26	4.333	15	7741	25469	3.290
7	11	44	4.000	16	19320	61729	3.195
8	23	97	4.217	17	48629	151897	3.124
9	47	189	4.021	18	123867	377951	3.051
10	106	416	3.925	19	317955	953876	3.000
11	235	887	3.774	20	823065	2423668	2.945
12	551	2006	3.641	21	2144505	6235148	2.907

Table 4.2: Average number of positions accessed by the WROM algorithm.

As discussed in Chapter 2 and 3, we simplified and introduced a recursive version of Beyer Hedetniemi's algorithm for generation of rooted trees. That idea can also be extended to simplify the WROM algorithm. In the next chapter, we present our new recursive algorithm for generating free trees.

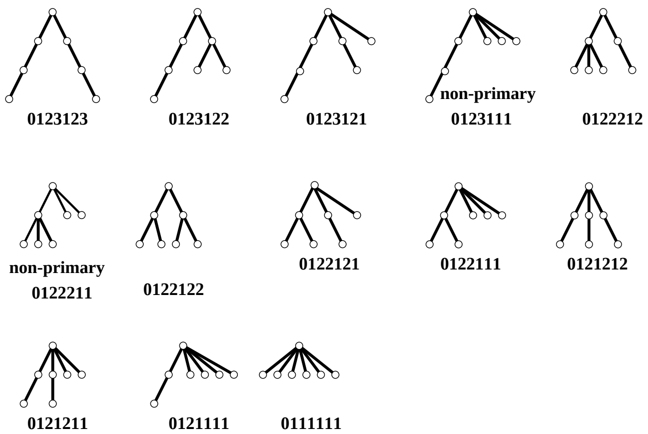

Figure 4.8: Free trees generated by the WROM algorithm.

Chapter 5

A New Recursive Algorithm for Generating Free Trees

5.1 An Introduction

In this chapter, we first introduce our new recursive algorithm for generating unlabeled free trees. We then present a proof of its correctness and a complexity analysis. Finally, we modify our algorithm to generate free trees under some height and/or degree constraints.

5.2 The Recursive Algorithm

Our recursive algorithm for free trees is an extension of our recursive algorithm for rooted trees, since the set of rooted versions of all free trees is a subset of the set of all canonic rooted trees, i.e., the rooted version of a free tree is actually a rooted tree in which the root is always a center of the original free tree.

5.2.1 How to approach the problem

There are many approaches to the problem of generating unlabeled free trees.

One approach is to generate free trees of size n by examining and extending free trees of size $n-1$. Figure 5.1 presents all unlabeled free trees of at most seven nodes.

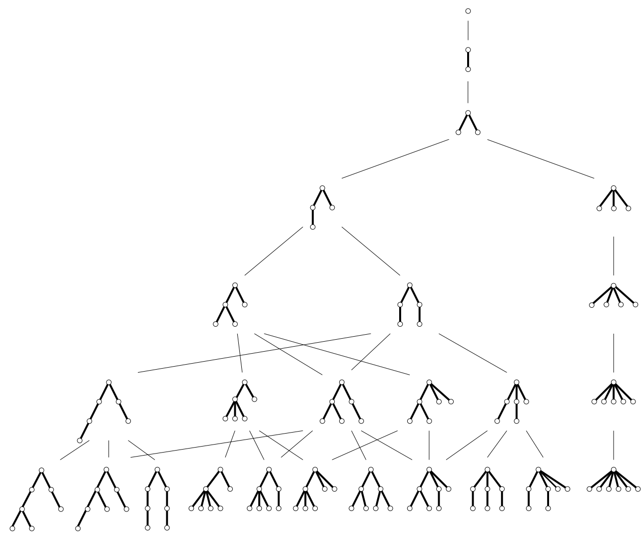

Figure 5.1: The Hasse diagram of the poset of free trees with at most 7 nodes.

The free trees at the same level have the same number of nodes, and are organized in relex (reversed lexicographic) order. We define a partial order $\prec$ among all free trees with at most n nodes. Define $T \prec T'$ if $|T| < |T'|$ and T can be obtained by recursively remove some leaves from T', i.e., T' is a "supertree" of T. In Figure 5.1 we show the Hasse diagram of the poset of all free trees with at most 7 nodes.

The Hasse diagram in Figure 5.1 illustrates that it is possible to generate free trees of size n from free trees of size $n-1$ by adding a leaf to a free tree of size $n-1$ as a child of some leaf. Given a free tree T of size $n-1$, to generate all T' so that $T \prec T'$ and $|T'| = n$, we need to solve two problems: one is how to pick the right leaf to add a new node as its child; the other is how to avoid duplicates (since the Hasse diagram in Figure 5.1 is not a tree structure, there are "loops" in it). Read [1] developed an algorithm which could generate all free trees of size n from all free trees of size $n-1$, but his algorithm had to examine and store all free trees of size $n-1$, and the space was exponential with respect to n. Another problem with this approach is that it is hard to generate all free trees of size n lexicographically in terms of the level sequence representation. The lexicographically biggest free tree of size $n-1$ may not always be used to generate the lexicographically biggest free tree of size n by adding a leaf. See, for example, the leftmost tree at level $n = 6$ in Figure

5.1 can not be generated from the leftmost tree at level $n = 5$ by adding a leaf.

Another approach is the WROM iterative algorithm. The algorithm adopted the idea from the Beyer-Hedetniemi algorithm for generating rooted trees. It used unlabeled rooted trees to represent free trees as we discussed in Chapter 1. It first initialized the lexicographically biggest free tree. It then applied Beyer-Hedetniemi's iterative algorithm to generate its successor. There are some cases where Beyer-Hedetniemi's algorithm generates some rooted trees which can not be used as a representation of a free tree (see [5] for definition of the representation). The WROM algorithm uses some techniques to detect and fix these failure cases (see Chapter 2 for details). For easy detection and repair, the WROM algorithm uses an unnatural way to represent free trees. Recall that a free tree is transferred into a rooted tree by picking a root for it. For unicentral free trees, the WROM algorithm picks the unique center as the root and arranges the subtrees of the resulting rooted tree recursively so that the subtrees on the left are always lexicographically bigger or equal to the ones on the right. But for bicentral free trees, there are two centers. By deleting the edges between these two centers we have two rooted subtrees (see Chapter 1 for more explanation). The WROM algorithm picks the center not purely according to the lexicographic order of the two rooted subtrees. It always arranges the bigger sized subtree on the right side of the root, and if they are of the same size, arranges the lexicographically bigger subtree on the right side of the root instead of the left as in unicentral case (see Figure 4.2 for some examples).

Compared to the WROM algorithm, we pick the root in a very natural and consistent way: we always pick the center which lexicographically maximizes the canonic level sequence of the resulting rooted tree T_r.

Our goals are, first, to maintain the same natural canonic representation of free trees extended from that of rooted trees (see Chapter 1), and second, to develop a simpler, more flexible recursive algorithm for generating free trees in relex order.

Recall from Chapter 1 that we represent a free tree T by picking a root r and then transferring it to its rooted version, a canonic rooted tree, T_r. The level sequence

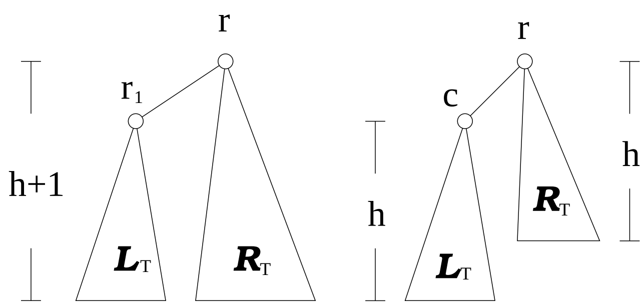

Figure 5.2: Two subtrees of unicentral and bicentral free trees.

lev_{T_r} and parent array par_{T_r} of this canonic rooted tree T_r becomes the representation of the free tree T.

If $|T| > 2$, we further divide the rooted version of the free tree T into two subtrees, L_T and R_T whose heights differ by at most one. See Figure 5.2. The definitions of L_T and R_T are given below:

If T is unicentral with center r, let $r_1, r_2, \ldots, r_m$ be the children of root r in the rooted version T_r of T, and let $T(r_i)$ be the subtree of T_r rooted at r_i for $i = 1 \ldots m$. From the definition of T_r we know that $T_i \preceq T_j$ if $i > j$. Now, we delete the edge between r and r_1. We have two rooted subtrees: L_T and R_T. The left subtree L_T is $T(r_1)$ and the right subtree R_T is T_r after removing subtree $T(r_1)$, denoted by $T_r \backslash T(r_1)$. See Figure 5.2.

If T is bicentral, let r be the center which lexicographically maximizes the canonic level sequence of the resulting rooted tree T_r, i.e., the rooted version of T, and c be the other center of T. After deleting the edge between r and c, similarly, we have two rooted subtrees of T_r: L_T and R_T. The left subtree L_T is $T(c)$ and the right subtree R_T is T_r after removing $T(c)$, denoted by $T_r \backslash T(c)$. See Figure 5.2.

Let r be the canonic center we choose, and $r_1 < r_2 < \cdots < r_m$ be the children of r. Then node r_1 is the root of L_T by definition. Recall that $T_r(r_i)$ is the subtree, rooted at r_i, of rooted tree T_r. Observe that by the definition of the rooted version T_r of T, and the way we choose L_T and R_T, we have the following proposition.

Proposition 5.2.1 *If T is unicentral, then $L_T \succeq T_r(r_2) \succeq \cdots \succeq T_r(r_m)$. If T is* *bicentral, then* $L_T \succeq R_T$*.*

Note that if T is a free tree in the WROM algorithm, we can define the same L_T and R_T, but Proposition 5.2.1 is not always true for the bicentral case: it is not true when $|L_T| = |R_T|$ and L_T not equivalent to R_T; sometimes true, sometimes not true otherwise (see Chapter 4 for details).

Since L_T and R_T are both canonic rooted trees, Figure 5.2 illustrates a way to construct the rooted version of a free tree T: first build a rooted tree L_T, then build a rooted tree R_T so that $|T| = |L_T| + |R_T|$ and Proposition 5.2.1 is satisfied. By connecting the root of L_T and the root of R_T, and making the root r of R_T as the new root, we form the rooted version T_r of the desired free tree T.

Now, the problem becomes how to generate free trees in relex order. Here is our strategy: We first label the root of R_T as 1, the root of L_T as 2, and let parent of 2 be 1, parent of 1 be 0. We then generate all those L_T's in relex order by using our recursive rooted tree algorithm with initial call $Gen(3, 0, 0)$, and then for fixed L_T, we generate corresponding R_T's in relex order. The parent array of T_r will be

$$\langle p_1, \underbrace{p_2, \ldots, p_{sL}}_{L_T}, \underbrace{p_{sL+1}, \ldots, p_N}_{R_T} \rangle$$

where p_i is the parent of node i in T_r. Note that it is always true that $p_1 = 0$ and $p_2 = 1$.

5.2.2 How to generate L_T's in relex order

We have a recursive algorithm $Gen()$ for generating rooted trees of fixed size (see Chapter 3). Suppose we want to generate all free trees of fixed size n. Observe that the size of L_T is not fixed even though the size of T is fixed. This makes the problem harder as we can not directly use $Gen()$.

The reason that the size of L_T is not fixed is that the difference between the height of L_T and R_T must not exceed 1, due to the root selection method for the rooted version of the free tree T.

Proposition 5.2.2 *For a rooted version T_r of a free tree T of size N and height h, the biggest possible size of L_T is obtained when the R_T is a chain of length $h-1$ for the bicentral case, and height h for the unicentral case.*

Proposition 5.2.2 implies the following proposition.

Proposition 5.2.3 *The lexicographically biggest possible L_T for a free tree T of size N and height h is the lexicographically biggest rooted tree of size $N-h$ and height $h-1$ for bicentral case, or of size $N-h-1$ and height $h-1$ for unicentral case.*

Figure 5.3 (a) and (b) show an example for $N = 10, h = 3$.

Proposition 5.2.4 *Let T be a free tree with N nodes and height h. The smallest value of $|L_T|$ is obtained when L_T has the following form.*

(a) If T is unicentral, then L_T is a chain of length $h-1$.

(b) If T is bicentral and not an odd length chain, then L_T is a chain C of length $h-1$ together with a node attached at a non-leaf of C.

(c) If T is an odd length chain (and so $h = N/2$), then L_T is a chain of length $h-1$.

See Figure 5.3 (c) and (d) for example.

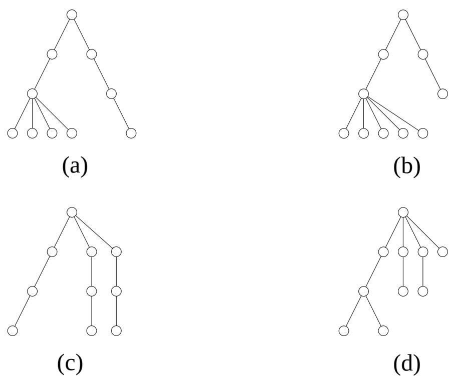

Figure 5.3: For $N = 10, h = 3$, (a) Biggest L_T for unicentral case, (b) Biggest L_T for bicentral case, (c) Smallest L_T for unicentral case, (d) Smallest L_T for bicentral case.

So, to generate all L_T's, we have to consider these constraints related to the height of T_r. Now two things need to be taken care of: generating all possible L_T's under these size constraints and generating them in relex order.

Let $N = |T|$ be the size of T, $sL = |L_T| + 1$ be the label of the last node in L_T, and h be the height of the rooted version T_r of T. Proposition 5.2.5 tells us the lexicographically biggest free tree of size N.

Proposition **5.2.5** *Let T_r be the canonic rooted version of free tree T, with L_T and R_T defined as before. Then T_r is the lexicographically biggest among all free tree T of size N if*

1). $sL = |L_T| + 1 = \lceil N/2 \rceil + 1$ and $h = \lfloor N/2 \rfloor$, and

2). If N is even, then both L_T and R_T are chains of length $h-1$, or

3). If N is odd, then L_T is the lexicographically biggest rooted tree of size $sL-1 = |L_T|$ and height $h-1$, and R_T is a chain of length $h-1$.

The leftmost tree at the bottom level (level 7) of Figure 5.1 shows an example of case 3) with $N = 7$ and the leftmost tree at level 6 shows an example of case 2) with $N = 6$.

Recall that our recursive algorithm *Gen(p,s,cL)* for generating rooted trees in Figure 3.3 generates rooted trees of fixed size n in relex order: the greater the height of a rooted tree, the earlier it gets generated. To generate L_T, we first initialize n to be sL, the size of L_T plus one (since the label of L_T starts with node 2 instead of 1), and then run the algorithm $Gen(p, s, cL)$. When the size of L_T needs to be changed, we change the value of n in the algorithm, and continue the execution of the recursive call $Gen(p, s, cL)$.

There are two instances when sL changes. One is after the node $|L_T|+1$, the last node in L_T, is added as the rightmost child of the root of L_T, which is 2 according to our preorder labeling; the other is when the height of L_T is reduced (by one).

After the node $|L_T| + 1$ being added as the rightmost child of the root of L_T, we generate the successor of L_T by throwing away the node $|L_T| + 1$. This node will be used in R_T. So, the size of the successor of L_T is one less than the size of L_T.

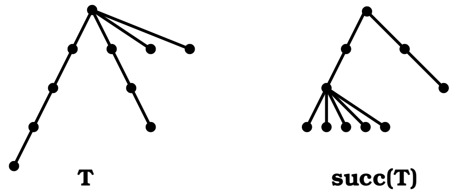

Figure 5.4: The successor of the smallest free tree of height h.

Note that this is a modification to the original rooted tree algorithm $Gen(p, s, cL)$. The original algorithm will not generate rooted trees of varied size. To implement this modification, we need only reduce sL by one and output the new L_T, represented by the sequence $L[2..sL]$, after the while loop at line R8-R12. We call this modification as **Modification L1**. We call the modified version of $Gen()$ for generating L_T $GenLT()$.

Observe that

PROPOSITION **5.2.6** *If the last node, $|L_T|+1$, in L_T is the rightmost child of the root 2 of L_T, then by removing this node, we obtain the lexicographically biggest rooted tree among those rooted trees of size less than or equal to N and lexicographically smaller than L_T.*

When the height of L_T is going to be reduced by one, what is the successor of the current free tree T? Observe that by Proposition 5.2.6, we have

LEMMA **5.2.1** *Only when rooted tree L_T is a chain will the modified recursive algorithm $GenLT()$ reduce the height of L_T by one (and only one) when generating the successor of L_T. This happens when $s = 0$ in the while loop at line R8 in $Gen(p, s, cL)$.*

PROOF:Since the successor of L_T can be of various size, if L_T is not a chain, $GenLT()$ will not have to move the last node on the leftmost path of L_T in order to reduce the height of L_T. Simply removing a leaf not on the leftmost path of L_T will give a rooted tree which has the same height as L_T, but is lexicographically smaller. $\square$

To generate the successor, $succ(L_T)$, of L_T mentioned in Lemma 5.2.1, we need to make the left subtree $L_{succ(T)}$ in the successor, $succ(T)$, of T as large as possible. By Proposition 5.2.3, $succ(T)$ must be a bicentral free tree such that $R_{succ(T)}$ is just a chain and $L_{succ(T)}$ is the lexicographically biggest in its size and height. Suppose p is the last node of L_T mentioned above. To generate $succ(T)$, we only need to reset h, which is currently $p-1$ since L_T is a chain, to $p-2$, then reset sL to $N-h+1$, and then continue the execution of the algorithm $GenLT()$. The algorithm will then generate the successor $L_{succ(T)}$ of L_T with new size $N-h+1$ instead of h (see Figure 5.4. We call this modification of the algorithm $GenLT()$ **Modification L2**. We add this modification into $GenLT()$, and still call the modified algorithm $GenLT()$.

PROPOSITION **5.2.7** *Given the rooted version T_r of a free tree T with L_T a chain of length $h-1$, where h is the height of T_r, Modification L2 will result in generating the lexicographically greatest successor $succ(T)$ of T, in the corresponding rooted version, such that $R_{succ(T)}$ is a chain of height $h-2$ and $L_{succ(T)}$ is the lexicographic biggest rooted tree of size $N-h+2$ and height $h-2$.*

Now, the prototype of our new algorithm for generating left subtree L_T will be $GenLT(p, s, cL, h, sL)$, where p, s, cL are inherited from $Gen(p, s, cL)$ with the same meaning (see Chapter 3). We add sL, the label of the last node in L_T, as a parameter to trace the current size of L_T, and h, the height of T, as another new parameter. Recall that $GenLT$ is obtained by implementing Modification L1 and L2 in $Gen(p, s, cL)$.

We first initialize root of R_T as 1, root of L_T as 2, and $par[2] = 1$. The initial call will be $GenLT(3, 0, 0, h, sL)$ (see Figure 5.2.2 for the algorithm $GenLT$) with h and sL as set in Proposition 5.2.5. To implement Modification L2, we just insert the following lines after {R8} in $Gen(p, s, cL)$:

```
if s=0 then begin
   h := p - 2;
   sL := N - h + 1;
end;
```

Since the node pp on the leftmost path of L_T is added into L_T by recursive call $GenLT(pp, 0, 0, h, sL)$ as in $Gen(pp, 0, 0)$, we use $s = 0$ to identify the situation when the last node p of L_T is on the leftmost path of L_T, i.e., L_T is a chain.

For Modification L1, we simply add the line

```
GenRT(p,2,p-2,h,sL-1);
```

after the while loop (line {R8-R12}) in $Gen(p, s, cL)$ to generate R_T such that the last node p in the current left subtree L_T is transferred to the right subtree in the successor of the current T. Note, $GenRT$ will be discussed in the next section.

There are some other small modifications to transfer $Gen()$ into $GenLT()$. In line {R1} of $Gen()$, n will be replaced by sL to indicate the completion of L_T, and $PrintIt$ is replaced by $GenRT()$ to generate the right subtree R_T for the current left subtree L_T. Since the root of L_T is 2 instead of 1, we replace 1 by 2 at line {R8} in $Gen()$. Algorithm $GenRT()$ will be discussed in the next section.

5.2.3 How to generate R_T's in relex order

For a fixed L_T, we will then generate R_T's for unicentral case first and then the bicentral case.

Since for a fixed L_T, the height of R_T in a bicentral T is one bigger than the height of R_T in a unicentral T. So, we have the following proposition.

PROPOSITION **5.2.8** *Given any bicentral free tree T and unicentral free tree T'. If $|T| = |T'|$ and L_T is equivalent to $L_{T'}$, then $R_T \preceq R_{T'}$.*

First we have to make sure that the number of nodes left from $GenLT$ is enough to generate R_T since we require the height of R_T to be exactly h for unicentral case, and $h - 1$ for bicentral case, where h is the height of the rooted version T_r of T. So, if $p = |L_T| + 1 = sL$ is the label of the last node of L_T (hence $N - p$ is the number of nodes left for R_T), we require that $N - p >= h$ for unicentral case, and $N - p >= h - 1$ for bicentral case.

```
procedure GenLT( p, s, cL, h, sL : integer);
begin
{T1}  if p > sL then GenRT(p,2,p-2,h,sL) {n->sL; PrintIt->GenRT}
{T2}  else begin
{T3}    if cL = 0 then par[p] := p-1 else
{T4}    if par[p-cL] < s
{T5}       then par[p] := par[s]
{T6}       else par[p] := cL + par[p-cL];
{T7}    GenLT( p+1, s, cL, h, sL );
{T8}    while par[p] > 2 do begin      {change 1 to 2}
{T9}       if s = 0 then begin         {Modification L2}
{T10}         h = p-2;                 {      ''       }
{T11}         sL = N - h + 1           {      ''       }
{T12}      end;                        {      ''       }
{T13}      s := par[p];
{T14}      par[p] := par[s];
{T15}      GenLT( p+1, s, p-s, h, sL )
{T16}   end;
{T17}   GenRT(p,2,p-2,h,sL-1)       {Modification L1, add GenRT}
{T18} end
end; {of GenLT};
```

LEMMA **5.2.2** *For N, p, h as above, $N - p >= h - 1$.*

PROOF:Initially, we set sL so that $N - p = N - sL = h - 1$ (see Proposition 5.2.5). sL will only be modified in two cases, Modification L1 and L2. Modification L1 will reduce sL by one, hence $N - p$ will be greater than $h - 1$. Modification L2 set sL so that $N - p = h - 1$ (see Proposition 5.2.7). □

We then have to check whether Proposition 5.2.1 will be satisfied if we adopt $Gen()$ as $GenRT()$ to generate R_T.

For unicentral case, if we have at least $h + 1$ nodes, including the root 1, for R_T, we can always do the following without violating Proposition 5.2.1: (Otherwise, no unicentral T will be generated with the current L_T.)

PROPOSITION **5.2.9** *For fixed L_T, to generate the lexicographically biggest T_r of a unicentral free tree T, we can call $Gen(p, 2, sL)$ to copy subtree L_T repeatedly.*

For bicentral case, from Lemma 5.2.2, we know we have at least enough nodes to build a chain of length $h - 1$, and this is a valid rooted version of a bicentral free tree. What happens when we have more nodes? We have to make sure Proposition 5.2.1 is preserved.

LEMMA **5.2.3** *If h is the current designated height of the rooted version T_r of a bicentral free tree T, $N = |T|$, and p is the last node in L_T, then R_T can be built without violating Proposition 5.2.1 only when one of the following conditions is satisfied:*

(D1) $(p - 1) * 2 \geq N$, *or*

(D2) $(p - h - 1 = 1)$ *and* $par[p] > 2$, *or*

(D3) $(p - h - 1 \geq 2)$ *and* $((par[h + 2] > 2)$ *or* $(par[h + 3] > 2))$.

PROOF:When condition (D1) is satisfied, $|L_T| \geq |R_T|$, so we can just make a (partial) copy of L_T to generate R_T. L_T will be lexicographically greater than or equal to R_T.

When condition (D2) is satisfied, we know that L_T is just a chain C of length $h - 1$ plus the last node p attached to a non-leaf node on the chain C which is not the root 2 of L_T. Since the last node p is not the child of the root of L_T, L_T has a successor of the same size or of larger size. So, if $|L_T| < |R_T|$, we can generate R_T by finding the successor of L_T.

When condition (D3) is satisfied, L_T is not a chain. It has at least two more nodes than a chain of length $h-1$. So as long as L_T is not the lexicographically smallest of height $h-1$, we are done. We know by the recursive definition of canonic rooted trees, if there are two consecutive nodes at level one, i.e. children of root, then all the following nodes have to be at level one. We know that node $h+1$ is the node in L_T with highest level. If the two following nodes, $h+2$ and $h+3$ are children of node 2, the root of L_T, then L_T is the lexicographically smallest of its size and its height. Otherwise we can always find a successor of L_T with the same height, $h-1$, so that R_T is the successor of the current L_T. $\square$

Lemma 5.2.3 will guarantee that if one of the conditions is satisfied, we can always expect a good and lexicographically biggest, with respect to the fixed L_T, R_T to be generated according to Proposition 5.2.1. We now apply $Gen(p, s, cL)$ to generate R_T's in order to build those bicentral free trees with fixed L_T.

Is it possible that all the conditions in Lemma 5.2.3 fail, i.e., no R_T will be generated for a L_T generated by $GenLT()$? Fortunately, this is impossible.

LEMMA **5.2.4** *If L_T is generated by $GenLT()$, then there exists at least one R_T so that Proposition 5.2.1 is satisfied.*

PROOF:The height h of the rooted version T_r of a free tree T plays an important role here. Observe that $h = \lfloor N/2 \rfloor$ is the biggest possible height of all free trees with N nodes. With such height, R_T will always exist. It is a chain of length $h-1$ for bicentral case, and h for unicentral case. See the first few trees at each level in Figure 5.1 for examples. If $h < \lfloor N/2 \rfloor$, then by Lemma 5.2.2 we have at least $h-1$ nodes left for R_T. We can always generate R_T by copying L_T to form a unicentral T if there are more than $h-1$ nodes left for R_T (otherwise, we can simply generate a R_T as a chain of length $h-1$ to form a bicentral T without violating Proposition 5.2.1). □

For a fixed L_T, The first R_T to be generated is the lexicographically biggest one. Since the root of R_T has been set, we can use $Gen(p, 2, p-2)$ to copy L_T except the root of L_T (note that $p-2 = |L_T| - 1$). But if the size of R_T is greater than the size of L_T, and we continuously make copies of L_T, R_T will be lexicographically bigger than L_T. In such a case, we need to find the lexicographically biggest R_T of height $h-1$ which is smaller than L_T (if $|R_T| = |L_T|$, we can just make a copy). The following lemma provides the solution to finding such an R_T.

LEMMA **5.2.5** *If* $(sL-1)*2 < N$*, i.e.,* $|L_T| < |R_T|$*, then we can find the successor* $R = succ(L_T)$ *of* L_T *with height* $h-1$ *and size* $N - sL + 1$ *by doing the following:*

Step1. Scan the array $par[2..sL]$ *backward to find the first position (i.e. the first node)* q *so that* q *is not a child of node 2 (root of* L_T*). If such* q *does not exist, then* R *does not exist, otherwise,*

Step2. If $par[q] = q-1$*, i.e.,* q *is the node on the leftmost path of* L_T *at level* h *of rooted version* T_r *of* T*, then no such* R *exists. Otherwise,*

Step3. Reset $s = par[q], par[q] = par[s]$*, and start to copy the subtree rooted at* s *until reaching the node* $|R|$*, the last node in* R*.*

PROOF:This is just an extension of our algorithm for rooted trees, except that we now generate successors with different sizes and the same height, which will not affect the correctness of our algorithm for rooted trees. For Step2, before we start to generate R_T, we already check (see Lemma 5.2.3) to make sure such a situation will not happen, i.e., we won't be required to generate R in such a situation. □

Lemma 5.2.5 shows how to modify $Gen(p, s, cL)$ to generate the lexicographically biggest R_T with height $h-1$ which is lexicographically less than L_T when $|L_T| < |R_T|$.

When doing the copy using $Gen(p, 2, p-2)$ to generate R_T, we let pp be the node in R_T supposed to copy node q defined in Step1 of Lemma 5.2.5. To modify $Gen()$ in order to identify pp, we do the following:

```
procedure modRT;
if ((sL-1)*2 < N) and (pp-cL<=sL) and (
       ((pp-cL+1<sL) and (par[pp-cL+1]=2) and
       (par[pp-cL+2]=2))                          {Case1}
    or ((pp-cL+1=1) and (par[pp-cL+1]=2))         {Case2}
    or (pp-cL+1)) then begin                      {Case3}
    s:= par[pp]; cL:= pp-s;
    par[pp]:=par[s];
end else if par[p-cL]=2 then par[p]:= 1;
end {of modRT}.
```

Figure 5.5: The code to modify $Gen(p, s, cL)$ to make the successor of L_T.

Observe that pp must be encountered before we finish the first copy of L_T in the generation of R_T (actually, there won't be second copy after the modification–we will be copying a new subtree). So we have $pp - cL <= sL$. Since Lemma 5.2.3 guarantees that L_T is not the lexicographically smallest one, such a pp exists, and it is not on the leftmost path of L_T. Then the corresponding node $q = pp - cL$ in L_T of pp must fall into one of the following three cases:

Case1. $q = pp - cL$ is followed by at least two nodes of level 2, i.e., they are the children of the root of L_T. So, we can use $pp - cl + 1 < sL$ to indicate there are at least two nodes beyond $pp - cL$ in L_T, and they are children of node 2, i.e., $par[pp - cL + 1] = 2$ and $par[pp - cL + 2] = 2$.

Case2. $q = pp - cL$ is followed by only one node at level 2 which is the only one beyond $pp - cL$ in L_T, i.e., $pp - cL + 1 = sL$ and $par[pp - cL + 1] = 2$.

Case3. $pp - cL$ is the last node in L_T which is not at level 2, i.e., $pp - cL + 1 > sL$. $pp - cL$ must be at level bigger than 2 because L_T is not the lexicographic smallest, and Case1 and Case2 has failed.

After identifying pp, we reduce the level of pp by setting the parent of pp be to the current grandparent of pp, then start to copy the subtree rooted at new rightmost sibling (used to be the parent of pp) of pp. Figure 5.2.3 shows a piece of code, called *modRT*, which will identify pp and make the corresponding modification.

Since $Gen(p, 2, p-2)$ (see Figure 5.2.2) will only copy the parenthood relation within L_T except the parenthood relation between the root 2 of L_T and the children of node 2, we have to make special arrangement. To make a copy of all nodes which are children of node 2, root of L_T, we simply add the following code to the end of $modRT$.

```
else if par[p-cL]=2 then par[p]:=1;
```

After creating the first R_T for fixed L_T, $Gen(p, s, cL)$ will do the job to generate all successors of R_T. One more thing to be aware of is that we don't want the height of R_T get reduced for bicentral case (also for unicentral case). To avoid this, we only need to check at the beginning of the while loop in line {R8} of $Gen(p, s, cL)$ if $p > sL + h$ for unicentral case, and $p > sL + h + 1$ for bicentral case If these conditions are satisfied, we go on. Otherwise, we skip the while loop.

After making all the modifications to $Gen(p, s, cL)$, we have the algorithm $GenRT$ for generating R_T in relex order (see Figure 5.6).

5.2.4 Generating free trees in relex order

Recall in the above section, before we generate R_T for a fixed L_T we have to check if there exists a valid R_T. We call this procedure $expand()$ to check if R_T is obtainable.

Observe that $GenLT()$ and $GenRT()$ has many thing shared with $Gen()$. We implement the algorithm by merging these two recursive procedures together to form a new recursive procedure called $GenFree()$. We introduce parameter f in $GenFree()$ where $f = 0$ means we are now generating L_T, i.e., we are in the $GenLT()$ part of $GenFree()$ and $f = 1$ means we are now generating R_T. Another parameter g will also be added so that $g = 0$ means we are generating R_T for unicentral T and $g = 1$ means we are generating R_T for bicentral T.

Our recursive algorithm for generating rooted version of free trees is shown in Figure 5.7.

```
procedure GenRT( p, s, cL, h, sL : integer);
begin
{W1}  if p > N then PrintIt    {We can now print the whole tree}
{W2}  else begin
{W3}    if cL = 0 then par[p] := p-1 else
{W4}    if par[p-cL] < s
{W5}       then par[p] := par[s]
{W6}       else begin
{W7}            par[p] := cL + par[p-cL];
{W8}            if bicentral then modRT;
{W9}      end;
{W10}   GenRT( p+1, s, cL, h, sL);
{W11}   while (par[p] > 2) and (not-reducing-height-of-RT) do begin
{W12}      s := par[p];
{W13}      par[p] := par[s];
{W14}      GenRT( p+1, s, p-s, h, sL);
{W15}   end;
{W16} end;
end {of GenRT};
```

Figure 5.6: An algorithm for generating R_T.

```
procedure expand(p, h, n: integer);
begin
{E1} if N-p >= h then GenRT(p+1,2,p-1,h,n,N,1,0);
{E2} if ((p-1)*2 >= N) or ((p-h-1=1) and (par[p]>2)) or
{E3}    ((p-h-1>=2) and ((par[h+2]>2) or (par[h+3]>2))) then
{E4}    GenRT(p+1,2,p-2,h,n,N,1,1)
end;

procedure GenFree( p, s, cL,h,sL, n, f,g : integer);
begin
{F1}  if p > n then begin
{F2}    if f = 0 then expand(p-1,h,n) else PrintIt
{F3}  end else begin
{F4}    if cL = 0 then par[p] := p-1 else
{F5}    if par[p-cL] < s then par[p]:=par[s]
{F6}    else begin
{F7}      par[p] := cL + par[p-cL];
{F8}      if g=1 then
{F9}        if ((sL-1)*2 < n) and (p-cL<=sL) and (
{F10}             ((p-cL+1<sL) and  (par[p-cL+1]=2)
{F11}             and (par[p-cL+2]=2))
{F12}          or ((p-cL+1=sL) and (par[p-cL+1]=2))
{F13}          or (p-cL+1>sL)) then begin
{F14}             s:= par[p]; cL:= p-s;
{F15}             par[p] := par[s]
{F16}       end else if par[p-cL]=2 then par[p]:=1
{F17}   end;
{F18}   GenFree( p+1, s, cL,h,sL,n,f,g );
{F19}   while (par[p] > 2) and ((f=0) or (p>sL+h-g)) do begin
{F20}        if s=0 then h:= p-2;
{F21}        s := par[p]; par[p] := par[s];
{F22}        if f=0 then GenFree(p+1,s,p-s,h,0,N-h+1,f,g)
{F23}        else GenFree(p+1,s,p-s,h,sL,n,f,g)
{F24}   end;
{F25}   if f=0 then expand(p-1,h,p-1)
{F26} end
end; {of GenFree};
```

Figure 5.7: A recursive algorithm for generating unlabeled free trees

The Figure 5.8 shows the output of our algorithm in parent arrays and level sequences for $N = 8$.

5.3 Proof of Correctness

THEOREM **5.3.1** *Algorithm GenFree() in Figure 5.7 generates all canonic representatives of unlabeled free trees in relex order.*

PROOF:

First of all, the algorithm *GenFree* correctly inherits the Copying Strategy introduced in *Gen* since *GenLT* and *GenRT* do not modify the copying techniques implemented in *Gen*.

1) The algorithm generates the lexicographically largest correctly. To generate the lexicographically biggest free tree T, we first set $sL = \lceil N/2 \rceil$ and $h = \lfloor N/2 \rfloor$. Then call the recursive $GenFree(3, 0, 0, h, 0, sL, 0, 0)$. It will first generate L_T in exactly the same way as *Gen*() with the height restriction. The correctness of *Gen* will guarantee that T described in Proposition 5.2.5.

2) For any given canonic representative of a free tree T of height h, the algorithm will produce its correct successor $succ(T)$.

{Case 1}. If the right subtree T_R is not the lexicographically smallest one of its size and its height, *GenRT* will correctly apply the Copying Strategy to generate the correct successor $succ(T_R)$ (See Chapter 3.2) of T_R. Since L_T is fixed and unchanged, the resulting rooted tree T' by connecting L_T and R_T is the correct successor of T_r: since $succ(R_T) \prec R_T$ implies that Proposition 5.2.1 is satisfied for the resulting rooted tree; secondly, if there is a free tree F so that the rooted version of F is lexicographically smaller than T_r and bigger than T'_r, then we have that $succ(T_R)$ is not the correct successor of T_R.

{Case 2}. If the right subtree T_R is the lexicographically smallest one of its size and its height, then it must be in one of two forms shown in Figure 5.9, and *GenRT* will stop at line {W11} since the algorithm is trying to reduce the height of R_T.

parent array	level sequence
0 1 2 3 4 1 6 7	0 1 2 3 4 1 2 3
0 1 2 3 3 3 1 7	0 1 2 3 3 3 1 2
0 1 2 3 3 2 1 7	0 1 2 3 3 2 1 2
0 1 2 3 3 1 6 7	0 1 2 3 3 1 2 3
0 1 2 3 3 1 6 6	0 1 2 3 3 1 2 2
0 1 2 3 3 1 6 1	0 1 2 3 3 1 2 1
0 1 2 3 2 5 1 7	0 1 2 3 2 3 1 2
0 1 2 3 2 2 1 7	0 1 2 3 2 2 1 2
0 1 2 3 2 1 6 7	0 1 2 3 2 1 2 3
0 1 2 3 2 1 6 1	0 1 2 3 2 1 2 1
0 1 2 3 1 5 6 1	0 1 2 3 1 2 3 1
0 1 2 2 2 2 2 1	0 1 2 2 2 2 2 1
0 1 2 2 2 2 1 7	0 1 2 2 2 2 1 2
0 1 2 2 2 2 1 1	0 1 2 2 2 2 1 1
0 1 2 2 2 1 6 6	0 1 2 2 2 1 2 2
0 1 2 2 2 1 6 1	0 1 2 2 2 1 2 1
0 1 2 2 2 1 1 1	0 1 2 2 2 1 1 1
0 1 2 2 1 5 5 1	0 1 2 2 1 2 2 1
0 1 2 2 1 5 1 7	0 1 2 2 1 2 1 2
0 1 2 2 1 5 1 1	0 1 2 2 1 2 1 1
0 1 2 1 4 1 6 1	0 1 2 1 2 1 2 1
0 1 2 1 4 1 1 1	0 1 2 1 2 1 1 1
0 1 1 1 1 1 1 1	0 1 1 1 1 1 1 1

Figure 5.8: Output of our free tree recursive algorithm for $N = 8$.

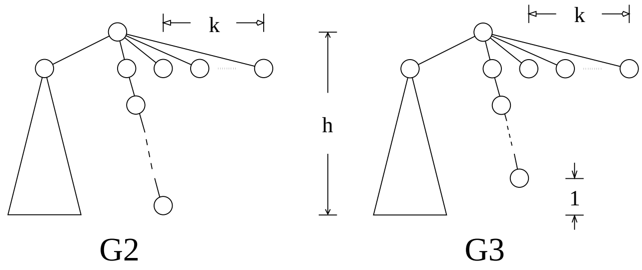

Figure 5.9: Two forms of the lexicographically smallest R_T.

Now, all recursive calls issued by *GenRT* will exit, and program returns to the line in *GenLT* where *GenRT* is called. *GenLT* will resume its execution by generating the successor of the current L_T.

To generate correct successor of the current L_T, *GenLT* will do the following:

{Case 2.1}. If L_T is not the lexicographically smallest of its size and its height, the Copying Strategy in *GenLT* inherited from *Gen* will correctly generate the successor $succ(L_T)$ of L_T. Then *GenRT* will be called to generate a new right subtree R for $succ(L_T)$ and R will be the lexicographically biggest for the fixed left subtree $succ(L_T)$. Clearly the free tree formed by $succ(L_T)$ and R is the correct successor of the current free tree T . By Proposition 5.2.8, we will generate unicentral free trees first if there are enough nodes (this is implemented in subroutine *expand*() in algorithm *GenFree*: we first let $g = 0$ to generate unicentral free trees; then let $g = 1$ to generate bicentral free trees) and then generate bicentral free trees. By Lemma 5.2.4, we can generate at least one R_T for the current L_T without violating Proposition 5.2.1.

{Case 2.1.1}. To generate the lexicographically biggest right subtree for unicentral free tree F, we simply use the Copying Strategy in *GenRT* to copy the current $L_F = succ(L_T)$ repeatedly. This will generate lexicographically biggest R_F without violating the Proposition 5.2.1. The resulting free tree F is the correct successor of T.

{Case 2.1.2}. To generate the lexicographically biggest right subtree for bicentral free tree F with a fixed left subtree $L_F = succ(L_T)$, we need to consider two cases: if

$|L_F| \geq |R_F|$, then we just copy (partially) the subtree L_F to be R_F without violating Proposition 5.2.1; if $|L_F| < |R_F|$, then we need to find the successor of L_F of size $|R_F|$. If the left subtree L_F satisfies the conditions in Lemma 5.2.3, then we can build a lexicographically biggest R_F for bicentral F with the fixed L_F by Lemma 5.2.5.

{Case 2.2}. If L_T is the lexicographically smallest of its size and its height, then L_T is either a chain or a chain plus some nodes added as children of the root of the L_T.

If L_T is a chain, *GenLT* will apply Modification L2 to reduce the height of L_T. Proposition 5.2.7 shows that *GenLT* will generate the correct successor $succ(L_T)$ of L_T of size $N - h + 2$ and height $h - 2$. $succ(L_T)$ has the biggest possible size of its height by Proposition 5.2.2, and it is the lexicographically biggest in its size and height by Proposition 5.2.3. After *GenRT* generates a new right subtree R (also the lexicographically biggest of its height and its size as stated in Proposition 5.2.2), the resulting rooted tree obtained by combining $succ(L_T)$ and R together is the correct successor of the current T_r.

If L_T is a chain with some nodes attached to the root of L_T, we can apply Modification L1 to generate the correct successor of L_T by Proposition 5.2.6.

□

5.4 Complexity Analysis

By examining the algorithm, we know that between each consecutive pair of recursive calls *GenFree()* there is only constant amount of work. There is only one while-loop in the algorithm. But for each loop executed, one recursive call will be made. So, we need only to count the number of the calls to *GenFree()* to estimate the time complexity of the algorithm. The results of an experimental test of the algorithm are shown in Table 5.1

Table 5.1 suggests that there exists a bound, $b = 5$, of average number of calls per tree generated, which would imply that the algorithm is CAT.

n	trees	calls	calls/trees	n	trees	calls	calls/trees
4	2	7	3.500	14	3159	8974	2.841
5	3	15	3.750	15	7741	20725	2.677
6	6	25	4.167	16	19320	49021	2.537
7	11	47	3.917	17	48629	117298	2.412
8	23	92	4.000	18	123867	285547	2.305
9	47	181	3.771	19	317955	703119	2.211
10	106	383	3.613	20	823065	1754073	2.131
11	235	803	3.403	21	2144505	4420303	2.061
12	551	1772	3.216	22	5623756	11253413	2.001
13	1301	3920	3.011	23	14828074	28895101	1.949

Table 5.1: Average number of recursive calls for generating free trees

Let us first take a look at the computation tree $\mathcal{F}_n$ of our algorithm(see Figure 5.10).

By the algorithm in Figure 5.7, we observe that the recursive call $GenFree()$ always occurs after an assignment of $par[p]$. This assignment represents the current status of the partially constructed free tree. We thus use the partially constructed free trees, represented by the parent arrays $par[1..p]$ for $1 \leq p \leq n$, to be the nodes in the computation tree $\mathcal{F}_n$ in Figure 5.10. We further divide these recursive calls into two classes:

{C1} the calls which will lead to valid free trees (in their rooted versions) at level n,

{C2} the calls which will not lead to any valid free trees at level n.

The calls in {C2} are wasted, they occur when $p > n$ and $f = 0$ (node p will get reassigned in the next call) at line {F1-F2}.

The execution of the algorithm $GenFree()$ is just a preorder traversal of the computation tree. To obtain a child of a node, we need only add a new node to a proper position in the currently half-built free tree. The nodes resulted from recursive calls in {C2} is represented by a symbol $\otimes$.

Observe that there are some single-branched paths in the computation tree. And furthermore, some of these paths occur in the middle of the paths from the root of

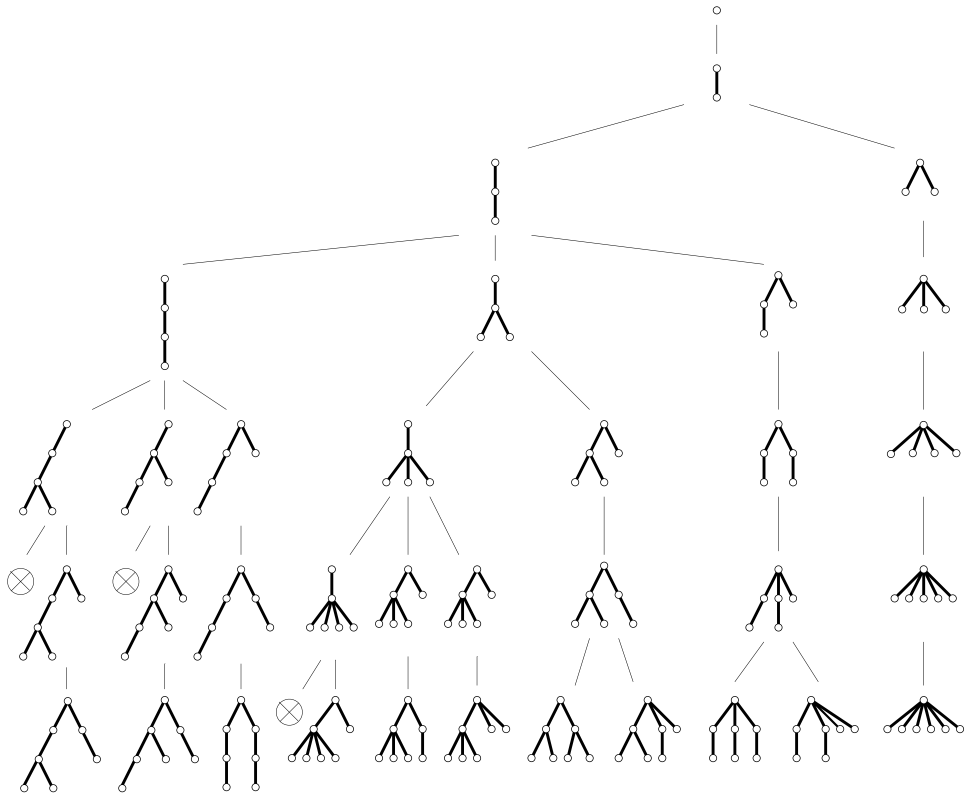

Figure 5.10: The computation tree $\mathcal{F}_n$ of our recursive free tree generation algorithm.

$\mathcal{F}_n$ to its leaves. This makes it difficult to use Path Elimination Techniques(PET) [8] to eliminate those single-branched paths as we did for rooted trees (see also section 3.3, 3.4).

Let us investigate closely those paths from leaves up to the root in the computation tree. For leaves x and y at level n in $\mathcal{F}_n$, let $d(x, y)$ be the distance to their youngest common ancestor in the computation tree. Let $succ(T)$ be the successor of the free tree T in relex order. Let $succ(T) = \mathcal{E}$ when T is the lexicographically smallest, and $d(T, \mathcal{E}) = n$. Let ε_n be the total number of wasted calls ($\otimes$ in computation tree $\mathcal{F}_n$). Then clearly, we have

$$\mid \mathcal{F}_n \mid = \varepsilon_n + \sum_{T \in \mathbf{F}_n} d(T, succ(T)) \tag{5.1}$$

where $\mathbf{F}_n$ is the set of all unlabeled free trees, and $\mid \mathcal{F}_n \mid$ is the number of nodes in $\mathcal{F}_n$.

We now partition the set $\mathbf{F}_n$ of free trees into three disjoint subsets: $G1 \oplus G2 \oplus G3$.

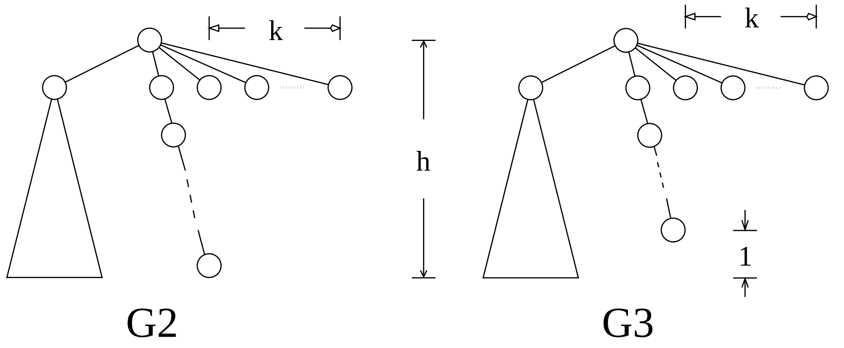

Figure 5.11: Free trees in $G2$ and $G3$.

Let $\langle e_1, e_2, \ldots, e_n \rangle$ be the canonic level sequence of the rooted version T_r of a free tree $T \in \mathbf{F}_n$, then

$T \in G2$ if and only if there is a $h \geq 1$ and a $k \geq 0$, such that

$$e_i = \begin{cases} 1 & \text{for } n-k < i \leq n \\ i-(n-k-h) & \text{for } n-k-h < i \leq n-k \text{ and } T \text{ is unicentral} \\ 1 & \text{only for } i = 2 \text{ if } 1 \leq i \leq n-k-h \end{cases}$$

$T \in G3$ if and only if there is a $h \geq 1$, and a $k \geq 0$, such that

$$e_i = \begin{cases} 1 & \text{for } n-k < i \leq n \\ i-(n-k-h+1) & \text{for } n-k-h+1 < i \leq n-k \text{ and } T \text{ is bicentral} \\ 1 & \text{only for } i = 2 \text{ if } 1 \leq i \leq n-k-h \end{cases}$$

$T \in G1$ if and only if $T \notin G2$, and $T \notin G3$.

The free trees in $G2$ and $G3$ are illustrated in Figure 5.11.

We now have that

Lemma **5.4.1**

$$d(T, succ(T)) = \begin{cases} 1 + k_T & \textit{for } T \in G1 \\ h_T + k_T + 1 & \textit{for } T \in G2 \\ h_T + k_T & \textit{for } T \in G3 \end{cases}$$

where k_T is the number of leaves in T adjacent to canonic center of T, i.e., the number of leaves at level 1, and h_T is the height of the canonic representative, which is a rooted tree, of T.

Proof:For any free tree T in $G1$, we only need to remove the k_T leaves attached to the canonic center of T, and then remove one more (which will be lifted up one level to get the successor of T later), which comes up to $1 + k_T$. For any free tree T in $G2$ or $G3$, we have to remove k_T leaves attached the canonic center as well as those nodes on the path from root to node $n - k_T$ in the right subtree, plus one more (as in case G1), in order to get the successor of T. This gives $h_T + k_T$ for bicentral case, and $h_T + k_T + 1$ for unicentral case. □

Now,

$$\begin{aligned}
\sum_{T\in\mathbf{F}_n} d(T, succ(T)) &= \sum_{T\in G1} d(T, succ(T)) + \sum_{T\in G2} d(T, succ(T)) + \sum_{T\in G3} d(T, succ(T)) \\
&= \sum_{T\in\mathbf{F}_n} (k_T + 1) + \sum_{T\in G2} h_T + \sum_{T\in G3} (h_T - 1)
\end{aligned}$$

Let $\alpha_n = \sum_{T\in\mathbf{F}_n}(k_T + 1)$, and $\beta_n = \sum_{T\in G2} h_T$, and $\gamma_n = \sum_{T\in G3}(h_T - 1)$. So,

$$\sum_{T\in\mathbf{F}_n} d(T, succ(T)) = \alpha_n + \beta_n + \gamma_n$$

Let $f_{n,k}$ denote the number of unlabeled free trees of size n with exactly k leaves attached to their canonic centers. By the definition of canonic representation of the free tree, in Chapter 1, we know that $f_{n,k} \le f_{n-k}$. Clearly, $\alpha_n = \sum_{T\in\mathbf{F}_n}(k_T + 1) = \sum_{k=0}^{n-1}(k+1)f_{n,k} \le \sum_{k=0}^{n-1}(k+1)f_{n-k}$.

Asymptotically from [18], we have

$$\alpha_n \le \sum_{k=0}^{n-1}(k+1)f_{n-k} \le 4f_n \tag{5.2}$$

We will then prove that β_n and γ_n are also $O(f_n)$.

A fundamental tool in our analysis is the following results from [13] and [12].

Theorem **5.4.1** *[13, 12] Let r_n be the number of unlabeled rooted tree of size n, and f_n be the unlabeled free tree of size n, we have*

$$r_n \sim \frac{C_1\rho^{-n}}{n^{3/2}} \tag{5.3}$$

$$f_n \sim \frac{C_2\rho^{-n}}{n^{5/2}} \tag{5.4}$$

where $C_1 \approx 0.4399, C_2 \approx 0.5349$ and $\rho \approx 0.3383$.

Let S_n^h be the number of unlabeled rooted trees of size n and height at most h. It was shown [18] that

$$S_n^h \leq \rho^{-n}(1+\frac{C_2}{h^2})^{-n} \leq C_3 r_n n^{3/2} exp(-\delta n/h^2) \tag{5.5}$$

for some constant $C_2 > 0, C_3 > 0, \delta > 0$ and $1 \leq h < n$.

Let r_n^h be the number of rooted trees of size n with height exactly h. Then, $r_n^h = S_n^h - S_n^{h-1}$. Observe that the number of free tree T in $G2$ with height h_T is the number of rooted trees of height $h_T - 1$ and of size $n - h_T - k_T - 1$ where k_T is the number of leaves attached to the canonic center of T. So, the size of $G2$ is equal to the number of rooted trees of height $h_T - 1$ with $n - h_T - k_T - 1$ nodes, where $1 \leq h_T \leq n/2$ and $0 \leq k_T \leq n - 2h_T - 1$, i.e., $|G2| = \sum_{h=1}^{n/2}\sum_{k=0}^{n-2h-1} r_{n-h-k-1}^{h-1}$.

Lemma **5.4.2**

$$\beta_n = O(f_n). \tag{5.6}$$

Proof:

$$\begin{aligned}
\beta_n &= \textstyle\sum_{T\in G2} h_T \\
&= \textstyle\sum_{h=1}^{n/2}\sum_{k=0}^{n-2h-1} h r_{n-h-k-1}^{h-1} \\
&= \textstyle\sum_{h=1}^{n/2}\sum_{k=0}^{n-2h-1} h(S_{n-h-k-1}^{h-1} - S_{n-h-k-1}^{h-2}) \\
&\leq \textstyle\sum_{h=1}^{n/2}\sum_{k=0}^{n-2h-1} h S_{n-h-k-1}^{h-1} \\
&\leq \textstyle\sum_{h=1}^{n/2}\sum_{k=0}^{n-2h-1} C_3 h r_{n-h-k-1}(n-h-k-1)^{3/2} e^{-\delta(n-h-k-1)/(h-1)^2} && \text{(eq(5.5))} \\
&\sim \textstyle\sum_{h=1}^{n/2}\sum_{k=0}^{n-2h-1} h C_1 C_3 \rho^{-(n-h-k-1)} e^{-\delta(n-h-k-1)/(h-1)^2} && \text{(eq(5.3))}
\end{aligned}$$

Now divide by f_n to get

$$\begin{aligned}
\frac{\beta_n}{f_n} &\leq \frac{\sum_{h=1}^{n/2}\sum_{k=0}^{n-2h-1} h C_1 C_3 \rho^{-(n-h-k-1)} e^{-\delta(n-h-k-1)/(h-1)^2}}{f_n} \\
&\sim \frac{\sum_{h=1}^{n/2}\sum_{k=0}^{n-2h-1} h C_1 C_3 \rho^{-(n-h-k-1)} e^{-\delta(n-h-k-1)/(h-1)^2}}{C_2 n^{-5/2}\rho^{-n}} && \text{(by eq. (5.4))} \\
&= \textstyle\sum_{h=1}^{n/2}\sum_{k=0}^{n-2h-1} \frac{h C_1 C_3 e^{-\delta(n-h-k-1)/(h-1)^2}}{C_2 n^{-5/2}\rho^{-h-k-1}} \\
&\leq \textstyle\sum_{h=1}^{n/2}\sum_{k=0}^{n-2h-1} \frac{C_1 C_3 n^{7/2}}{C_2 \rho^{-h-k-1} e^{\delta(n-h-k-1)/(h-1)^2}} && \text{(replace } h \text{ with } n) \\
&\leq \textstyle\sum_{h=1}^{n/2}\sum_{k=0}^{n-2h-1} \frac{C_1 C_3 n^{7/2}}{C_2 e^{(h+k+1)+\delta(n-h-k-1)/(h-1)^2}} && \text{(since } 1/p > e)
\end{aligned}$$

From Lemma 5.4.3, we have $(h+k+1)+\delta(n-h-k-1)/(h-1)^2 \geq C_4 n^{C_5}$ for some constant $C_4 > 0, C_5 > 0$ when $1 < h \leq n/2$ and $0 \leq k \leq n-2h-1$. Hence, we have

$$\begin{aligned} \frac{\beta_n}{f_n} &\leq \sum_{h=1}^{n/2} \sum_{k=0}^{n-2h-1} \frac{C_1 C_3 n^{7/2}}{C_2 e^{C_4 n^{C_5}}} \\ &\leq \frac{C_1 C_3 n^{11/2}}{C_2 e^{C_4 n^{C_5}}} \longrightarrow 0 \text{ when } n \longrightarrow \infty \end{aligned}$$

□

Lemma **5.4.3**

$$(h+k+1)+\delta(n-h-k-1)/(h-1)^2 \geq C_4 n^{C_5}$$

for some constants $C_4 > 0, C_5 > 0$ where n, h, k, δ as in Lemma 5.4.2.

Proof:Let $\phi(h,k) = (h+k+1)+\delta(n-h-k-1)/(h-1)^2$. For $1 \leq h < \sqrt{\delta}+1$, $\phi(h,k)$ will decrease when k is increased. So, $k = n-2h-1$ will minimize the value of function $\phi(h,k)$, and $\phi(h,k) \geq (n-h)+\delta h/(h-1)^2$. Since h is bounded by $\sqrt{\delta}+1$, there exists a constant d_1 such that $\phi(h,k) \geq n+d_1$.

For $\sqrt{\delta}+1 \leq h \leq n/2$, $\phi(h,k)$ will not decrease when k is increasing. So $k=0$ will minimize the value of $\phi(h,k)$, and $\phi(h,k) \geq (h+1)+\delta(n-h-1)/(h-1)^2 \geq C_4 n^{C_5}$ for some constants $C_4 > 0$ and $C_5 > 0$ since the value of the function ϕ will reach its lowest point when $h = O(n^{C_5})$ for some constant $C_5 > 0$. □

Lemma **5.4.4**

$$\gamma_n = O(\sum_{h=1}^{n/2} \sum_{k=0}^{n-2h} (h-1) S_{n-h-k}^{h-1}) = O(f_n). \tag{5.7}$$

Proof:We can use a similar arguement as in Lemma 5.4.2 to prove this lemma since the constant 1 in equation 5.6 does not matter. □

Lemma **5.4.5**

$$\varepsilon_n = O(f_n) \tag{5.8}$$

n	trees	WROM	Time/trees	New Alg	Time/trees
16	19320	0.35	0.0000181	0.394	0.0000204
17	48629	0.88	0.0000181	0.91	0.0000187
18	123867	2.2	0.0000178	2.08	0.0000168
19	317955	5.64	0.0000177	4.89	0.0000154
20	823065	14.3	0.0000174	11.8	0.0000143
21	2144505	36.9	0.0000172	28.9	0.0000135
22	5623765	95.9	0.0000171	72.3	0.0000128
23	14828074	261.1	0.0000176	186.8	0.0000126

Table 5.2: The running time (in seconds) comparison of our algorithm with the WROMalgorithm.

PROOF:

$\varepsilon_n < f_n$ since node $\otimes$ always has a right sibling x that is not a $\otimes$ node. And such case happens only when the program switches from generating left subtree L to generating the right subtree R, so it occurs at most once in the generation of a leaf node(a free tree). □

From equations (5.1), (5.2), (5.6), (5.7), (5.8), we have

THEOREM **5.4.2** $\mid \mathcal{F}_n \mid = O(f_n)$ *which implies that the algorithm in Figure 5.7 is CAT.*

We implement the algorithm (see appendix) in Pascal. The algorithm(even though recursive algorithm usually requires extra time) turned out to be faster (see Table 5.4) compared to the WROM algorithm(both algorithm are CAT).

5.5 Generating Free Trees with Diameter Restrictions

This algorithm can also be easily modified to generate free trees with diameter constraints.

To generate free trees with diameter at most U, you may let $ub = (U+1)/2$, and call

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$h \leq 4$	1	1	1	2	3	5	8	12	18	26	37	51	71	95	128
$h \leq 5$	1	1	1	2	3	6	10	19	32	58	95	161	258	417	647

Table 5.3: The number of free trees with height restrictions

$$GenFree(3, 0, 0, ub, n - ub + 1, 0, 0).$$

To generate free trees with diameter at least L, you may set a global variable to check the parameter p in the call $GenFree(p, s, cL, h, l, n, f, g)$. Stop the program when $p \leq (L + 3)/2$ and $s \neq 0$.

Properly combining the above two will produce the free trees with diameter exactly d.

Clearly, these simple modification will not affect the CAT property of the original algorithm since we only add constant amount of work into each recursive call.

5.6 Generating Free Trees with Bounded Degree

Similarly, the algorithm can be modified to generate free trees with bounded degree d by replacing each recursive call $GenFree(p, s, cL, h, l, n, f, g)$ with

```
chi[par[p]] := chi[par[p]] + 1;
if par[p] = root then k := d else k := d-1;
if chi[par[p]] <= k then GenFree(p,s,cL,h,l,n,f,g);
chi[par[p]] := chi[par[p]] - 1;
```

where $chi[p]$ is the number of children of node p, $par[p]$ is the parent node of p.

To make the modified algorithm CAT, we can also implement an array called $jump[]$ as for rooted tree algorithm which maintains the current number of children of each nodes.

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$m = 3$	1	1	1	2	2	4	6	11	18	37	66	135	265	552	1132
$m = 4$	1	1	1	2	3	5	9	18	35	75	159	355	802	1858	4347
$m = 5$	1	1	1	2	3	6	10	21	42	94	204	473	1098	2633	6353

Table 5.4: The number of Cayley m-free trees with n nodes

Chapter 6

Conclusions

In this thesis, we presented two recursive algorithms for generating unlabeled rooted trees and free trees. Both algorithms are proved to be CAT. These recursive algorithms are very simple and flexible compared to those old algorithms. They can be simply modified to generate trees under parenthood constraints (or degree constraints), or height restrictions.

Bibliography

[1] R.C. Read, *How to grow trees,* in Combinatorial Structures and their Applications, Gordon and Breach, New York, 1970.

[2] S. Zaks, *Lexicographic generation of ordered trees*, Theoretic Computer Science, 10 (1980), pp63-82.

[3] S. Zaks, D. Richards, *Generating trees and other combinatorial objects lexicographically*, SIAM J. Comp. 8. 73 (1979).

[4] R.W. Robinson, F. Harary, and A.T. Balaban, *The Number of Chiral and Achiral Alkanes and Monosubstitued Alkanes*, Tetrahedron, Vol. 32, pp.355-361, Pergamon Press, 1976.

[5] T. Beyer and S.M. Hedetniemi, *Constant Time Generation of Rooted Trees*, SIAM J. Computing, 9 (1980) 706-712.

[6] D.E. Knuth, *Fundamental Algorithms, The Art of Computer Programming*, 1968, Addison-Wesley.

[7] F. Ruskey, *Generating t-ary trees lexicographically*, SIAM Journal of Computing, 7 (1978), pp424-439.

[8] F. Ruskey, *Combinatorial Generation*, in preparation, 1996.

[9] F. Ruskey and T. Hu, *Generating binary trees lexicographically*, SIAM J. Computing, **6** (1977), pp745-758.

[10] F. Ruskey and A. Proskurowski, *Generating binary trees by transpositions*, Journal of Algorithms **11** (1990) pp 68-84.

[11] E. Kubicka, *An Efficient Method of Examining All Trees*, unpublished manuscript, 1991.

[12] R. Otter, *The number of Trees*, Annals of Mathematics, Vol. 49, No. 3, July, 1948.

[13] G. Pólya, *Kombinatorische Anzahlbestimmungen für Gruppen, Graphen und chemische Verbindungen*, Acta Math., 68 (1937), pp. 145-254.

[14] E. Kubicka and G. Kubicki, *Constant Time Algorithm for Generating Binary Rooted Trees*, Congressus Numerantium, 90 (1992) 57-64.

[15] J. Pallo, *Lexicographic generation of binary unordered trees*, Pattern Recognition Letters, 10 (1989) 217-221.

[16] H.I. Scions, *Placing Trees in Lexicographic Order*, Machine Intelligence, 3 (1969) 43-60.

[17] H.S. Wilf, *Combinatorial Algorithms: An Update*, SIAM, CBMS 55, 1989.

[18] R.A. Wright, B. Richmond, A. Odlyzko, and B.D. McKay, *Constant Time Generation of Free Trees*, SIAM J. Computing, 15 (1986) 540-548.

[19] A. V. Kozina, *Coding and generation of nonisomorphic trees*, Cybernetics (Kibernetica), vol. 15 (5), 1975 (1979), pg. 645-651 (38-43).

[20] A. Cayley, *Collected Mathematical Papers*, Cambridge, 1889-1897; **3**, 242; **9**, 202, 427; **11**, 365; **13**, 26.

[21] T. Beyer and F. Ruskey, *Constant Average Time Generation of Subtrees of Bounded Size,* unpublished manuscript, May, 1989.

[22] J. Fill and E.M. Reingold, *Solutions Manual for Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, 1977.

[23] T. Hikita, *Listing and Counting Subtrees or Equal Size of a Binary Tree,* Information Processing Letters, 17 (1983) 225-229.

[24] J. Liu, *Lexicographic generation of rooted trees and trees*, Kexue Tongbao, Vol. 28 No. 4 pp448-451, 1983.

[25] V. Vajnovszki, *Constant time generation of binary unordered trees*,Bulletin of the European Association for Theoretical Computer Science, Number 57 (1995), pp221-229.

[26] G. Tinhofer and H. Schreck, *Linear time tree codes*, Computing, 33, pp211-225, 1984.

[27] T.C. Hu and F. Ruskey, *Circular cuts in a network,* Mathematics of Operations Research, 5 (1980) pp422-434.

[28] A. Meir and J.W. Moon, *On Subtrees of Certain Families of Rooted Trees,* Ars Combinatoria, 16-B (1983) pp305-318.

[29] A. Nijenhuis and H. S. Wilf, *Combinatorial Algorithms*, Second Edition, Academic Press, 1978.

[30] F. Ruskey, *Listing and Counting Subtrees of a Tree,* SIAM J. Computing, 10 (1981) pp141-150.

[31] L.K. Swift, T.Johnson, and P.E. Livadas, *Parallel Creation of Linear Octrees from Quadtree Slices*, Parallel Processing Letters, World Scientific Publishing Company, 1994.

[32] Sabra S. Anderson, *Graph Theory and Finite Combinatorics*, p34, Markham Publishing Company, 1970.

[33] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, *Introduction to Algorithms*, The MIT Press, 1992.

[34] Etherington, *Non-associative powers and a functional equation*, Mathematical Gazette, 21 (1937) pp36-39.

[35] Wedderburn, *The functional equation* $g(x^2) = 2\alpha x + g^2(x)$, Annals of Mathematics, 24 (1922) pp121-140.

[36] Louis Comtet, *Advanced Combinatorics*, D. Reidel Publishing Company, Dordrecht, Holland, 1974.

Appendix A

Wright,Richmond,Odlyzko and McKay's Free Tree Program

The following Pascal program is translated from original authors' algorithm.

```
procedure nexttree;
var
  fixit, needr, needc, needh2 : boolean;
  oldp,oldq,oldwq,delta : integer;
begin
  fixit := false;
  if (c=n+1) or (p=h2) and ((L[h1]=L[h2]+1) and (n-h2>r-h1)
    or (L[h1]=L[h2]) and (n-h2+1<r-h1)) then begin
    if L[r]>3 then begin
      p:= r; q:= W[r];
      if h1=r then h1:=h1-1;
      fixit := true;
    end else begin
      p:=r; r:=r-1;  q:=2;
    end;
  end;
```

```
needr:= false; needc:= false; needh2:= false;
if p<=h1 then h1:= p-1;
if p<=r  then needr:= true  else
 if p<=h2 then needh2:= true else
   if (L[h2]=L[h1]-1) and (n-h2=r-h1) then begin
     if p<=c then needc:= true; end
   else c:= MAX;

oldp:=p; delta:=q-p; oldq:= L[q]; oldwq:=W[q]; p:=MAX;
for i:=oldp to n do begin
  L[i]:=L[i+delta];
  if L[i]=2 then W[i]:=1 else
  begin
    p:=i;
    if L[i]=oldq then q:= oldwq else
      q:= W[i+delta] - delta;
    W[i]:= q;
  end;
  if (needr) and (L[i]=2) then begin
    needr:=false; needh2:=true; r:= i-1;
  end;
  if (needh2) and (L[i]<=L[i-1]) and (i>r+1) then begin
    needh2:=false; h2:=i-1;
    if (L[h2] = L[h1]-1) and (n-h2 = r-h1) then needc:= true
    else c:= MAX;
  end;
  if needc then begin
   if L[i]<>L[h1-h2+i]-1 then begin
      needc:=false; c:= i;
   end else
      c:= i+1;
```

```
  end;
end;
```

Appendix B

Implementation of $Jump[]$

The following code is our PASCAL implementation of the array $Jump[]$ and $Chi[]$, as mentioned in section 2.2, to generate canonic trees in which each node has at most k children.

```
procedure Gen( p, s, cL : integer);
var
  entry : integer;       { first good pos in jump[] }
  temp  : integer;
begin
  numgen := 1 + numgen;
  if (p > n) then PrintIt
  else begin
     if cL = 0 then begin    {<---- first tree}
         par[p] := p-1;
     end else
        if par[p-cL] < s
           then par[p] := par[s]
           else par[p] := cL + par[p-cL];
     chi[par[p]] := chi[par[p]] + 1;
     temp := rChi[par[p]]; rChi[par[p]] := p;
     if chi[par[p]] <= k then begin
         if chi[par[p]] < k then jump[p]:= par[p]
```

```
 else jump[p] := jump[par[p]];
        Gen( p+1, s, cL );
    end;
    chi[par[p]] := chi[par[p]] - 1;
    rChi[par[p]] := temp;
    jump[p] := jump[par[p]];
    entry := jump[p];
    while entry >= 1 do begin
        par[p]:= entry;
        chi[entry] := chi[entry] + 1;
        temp := rChi[par[p]]; rChi[par[p]] := p;
        if (chi[entry] >= k)  then jump[p] := jump[entry];
        Gen( p+1, temp, p-temp );
        chi[entry] := chi[entry] - 1;
        rChi[par[p]]:= temp;
        entry:= jump[entry];
        jump[p] := entry;
    end;
  end;
end {of Gen};
```

Appendix C

Pascal code for recursive generation of free trees

```
program genTree( input, output);

const MaxSize  = 50;  { max size of the tree }

var
 N     : integer;                           { number of nodes in a tree }
 par     : array [1.. MaxSize] of integer;     { parent position of i }
 num     : integer;                               { total number of trees }
 ub      : integer;    {upper bound }

procedure PrintIt;
var i : integer;
begin
  num := num+1;
  write( '[',num:3,']' );
  for i:=1 to N do write(par[i]:3); writeln;
end {of PrintRight};

procedure Gen( p, s, cL,h,l, n, f,g : integer); forward;
```

```
procedure expand(p, h, n, N: integer);
begin
 if N - p >= h then Gen(p+1,2,p-1,h,n,N,1,0);
 if ((p-1)*2 >= N) or ((p-h-1=1) and (par[p]>2)) or
    ((p-h-1>=2) and ((par[h+2]>2) or (par[h+3]>2))) then
    Gen(p+1,2,p-2,h,n,N,1,1);
end;

procedure Gen( p, s, cL,h,l, n, f,g : integer);
begin
  if (p > n) then begin
   if (f = 0) then expand(p-1,h,n,N) else PrintIt; end
  else begin
    if (cL = 0) and (p<=ub+2) then par[p] := p-1 else
    if par[p-cL] < s then par[p]:=par[p-cL]
    else begin
      par[p] := cL + par[p-cL];
      if (g=1) then
        if ((l-1)*2 < n)
           and (p-cL<=l) and (
               ((p-cL+1<l) and  (par[p-cL+1]=2)
               and (p-cL+2<=l) and (par[p-cL+2]=2))   {case 1}
           or ((p-cL+1=l) and (par[p-cL+1]=2))        {case 2}
           or (p-cL+1>l)) then begin                  {case 3}
              s:= par[p]; cL:= p-s;
              par[p] := par[par[p]];
        end else if (par[p-cL]=2) then par[p]:=1;
    end;
    Gen( p+1, s, cL,h,l,n,f,g );
    while (par[p] > 2) and ((f=0) or (p>l+h-g)) do begin
         if (s=0) then h:= p-2;
```

```
        s := par[p]; par[p] := par[s];
        if (f=0) then Gen(p+1,s,p-s,h,0,N-h+1,f,g)
        else Gen(p+1,s,p-s,h,l,n,f,g);
    end;
    if (f = 0) then expand(p-1,h,p-1,N);
  end;
end {of Gen};

begin {------------------ main --------------------}
  write('Input N ='); readln(N);
  ub :=N div 2;
  par[1] := 0; par[2] := 1;
  Gen( 3, 0, 0,ub,0,(N+3)div 2,0,0);
  writeln('total = ',num:3);
end.  {------------------ main --------------------}
```

VITA

Surname:	Li	Given Names:	Gang

Place of Birth: Kuitun, Xingjiang, China

Educational Institutions Attended:

University of Victoria	1994 to 1996
University of Calgary	1992 to 1994
People's University of China	1986 to 1990

Degrees Awarded:

B.Sc.	Poeple's University of China	1990
M.Sc.	University of Calgary	1994

Honours and Awards:

University of Victoria Fellowship	1994-96

Publications: