经典计算机科学著作最新版

计算机程序设计艺术

第4卷 第4册（双语版）

生成所有树
组合生成的历史

The Art of Computer
Programming, Volume 4
Generating All Trees
History of Combinatorial
Generation

Fascicle 4

苏运霖 译

（美）Donald E. Knuth 著

机械工业出版社
China Machine Press

计算机程序设计艺术

第4卷　第4册
生成所有树
组合生成的历史

The Art of Computer Programming
Volume 4, Fascicle 4
Generating All Trees
History of Combinatorial Generation

（双语版）

（美）　Donald E. Knuth　著
斯坦福大学

苏运霖　译

机械工业出版社
China Machine Press

关于算法分析的这套多卷论著已经长期被公认为经典计算机科学的定义性描述。《计算机程序设计艺术，第4卷 组合算法》是作者近期发表的部分内容。作为关于组合查找的冗长一章的一部分，这个分册讨论生成所有树和组合生成的历史。读者从本册中不仅会看到很多新内容，而且会发现与第1卷～第3卷及计算机科学和数学领域的丰富联系。一如既往，书中包括了大量的习题和富有挑战性的难题。

本书版权登记号：图字：01-2006-1931

图书在版编目（CIP）数据

计算机程序设计艺术 第4卷 第4册 生成所有树—组合生成的历史（双语版）/（美）克努特（Knuth, D. E.）编著；苏运霖译. —北京：机械工业出版社，2007.4

书名原文：The Art of Computer Programming, Volume 4, Fascicle 4：Generating All Trees, History of Combinatorial Generation

ISBN 978-7-111-20825-9

Ⅰ.计…　Ⅱ.①克…　②苏…　Ⅲ.程序设计－英、汉　Ⅳ.TP311.1

中国版本图书馆CIP数据核字（2007）第013348号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）
责任编辑：王　玉
北京瑞德印刷有限公司印刷 · 新华书店北京发行所发行
2007年4月第1版第1次印刷
170mm × 242mm · 15.75印张
定价：42.00元

PREFACE

I like to work in a variety of fields
in order to spread my mistakes more thinly.
— VICTOR KLEE (1999)

THIS BOOKLET is Fascicle 4 of *The Art of Computer Programming*, Volume 4: *Combinatorial Algorithms*. As explained in the preface to Fascicle 1 of Volume 1, I'm circulating the material in this preliminary form because I know that the task of completing Volume 4 will take many years; I can't wait for people to begin reading what I've written so far and to provide valuable feedback.

To put the material in context, this fascicle contains Sections 7.2.1.6 and 7.2.1.7 of a long, long chapter on combinatorial searching. Chapter 7 will eventually fill three volumes (namely Volumes 4A, 4B, and 4C), assuming that I'm able to remain healthy. It will begin with a short review of graph theory, with emphasis on some highlights of significant graphs in the Stanford GraphBase, from which I will be drawing many examples. Then comes Section 7.1, which deals with bitwise manipulation and with algorithms relating to Boolean functions. Section 7.2 is about generating all possibilities, and it begins with Section 7.2.1: Generating Basic Combinatorial Patterns. Details about various useful ways to generate n-tuples, permutations, combinations, and partitions appear in Sections 7.2.1.1 through 7.2.1.5. That sets the stage for the main contents of the present booklet, namely Section 7.2.1.6, which completes the study of basic patterns by discussing how to generate various kinds of tree structures; and Section 7.2.1.7, which completes the story of the preceding subsections by discussing the origins of the concepts and pointing to other sources of information. Section 7.2.2 will deal with backtracking in general. And so it will go on, if all goes well; an outline of the entire Chapter 7 as currently envisaged appears on the `taocp` webpage that is cited on page ii.

I had great pleasure writing this material, akin to the thrill of excitement that I felt when writing Volume 2 many years ago. As in Volume 2, where I found to my delight that the basic principles of elementary probability theory and number theory arose naturally in the study of algorithms for random number generation and arithmetic, I learned while preparing Section 7.2.1 that the basic principles of elementary combinatorics arise naturally and in a highly motivated way when we study algorithms for combinatorial generation. Thus, I found once again that a beautiful story was "out there" waiting to be told.

In fact, I've been looking forward to writing about the generation of trees for a long time, because tree structures have a special place in the hearts of all computer scientists. Although I certainly enjoyed preparing the material about classic combinatorial structures like tuples, permutations, combinations, and partitions in Sections 7.2.1.1–7.2.1.5, the truth is that I've saved the best for last: Now it's time for the dessert course. Ever since 1994 I've been giving an annual "Christmas tree lecture" at Stanford University, to talk about the most noteworthy facts about trees that I learned during the current year, and at last I am able to put the contents of those lectures into written form. This topic, like many desserts, is extremely rich, yet immensely satisfying. The theory of trees also ties together a lot of concepts from different aspects of computer programming.

And Section 7.2.1.7, about the history of combinatorial generation, was equally satisfying to the other half of my brain, because it involves poetry, music, religion, philosophy, logic, and intellectual pastimes from many different cultures in many different parts of the world. The roots of combinatorial thinking go very deep, and I can't help but think that I learned a lot about human beings in general as I was putting the pieces of this story together.

My original intention was to devote far less space to such subjects. But when I saw how fundamental the ideas were, I knew that I could never be happy unless I covered the basics quite thoroughly. Therefore I've done my best to build a solid foundation of theoretical and practical ideas that will support many kinds of reliable superstructures.

I thank Frank Ruskey for bravely foisting an early draft of this material on college students and for telling me about his classroom experiences. Many other readers have also helped me to check the first drafts, especially in Section 7.2.1.7 where I was often operating at or beyond the limits of my ability to understand languages other than English.

I shall happily pay a finder's fee of \$2.56 for each error in this fascicle when it is first reported to me, whether that error be typographical, technical, or historical. The same reward holds for items that I forgot to put in the index. And valuable suggestions for improvements to the text are worth 32¢ each. (Furthermore, if you find a better solution to an exercise, I'll actually reward you with immortal glory instead of mere money, by publishing your name in the eventual book:–)

Cross references to yet-unwritten material sometimes appear as '00' in the following pages; this impossible value is a placeholder for the actual numbers to be supplied later.

Happy reading!

Stanford, California D. E. K.
June 2005

A note on notation. At the beginning of Chapter 7 I'll define some operations on graphs for which many different notations are presently rampant. My current plan is to say that, if G is a graph on the vertices $U = \{u_1, \ldots, u_m\}$ and if H is a graph on the vertices $V = \{v_1, \ldots, v_n\}$, then:

- $G + H$ is the sum, aka juxtaposition, of G and H: It has the $m + n$ vertices $U \cup V$ and the edges of G and H.
- $G \mathbin{\bar{+}} H$ is the cosum, aka join, of G and H, namely the complement of the juxtaposition of their complements. (Thus its edges are those of G and H, plus all $u_j \mathbin{—} v_k$.)
- $G \times H$ is the Cartesian product of G and H: It has the mn vertices $U \times V$; its edges are $(u, v) \mathbin{—} (u', v)$ when $u \mathbin{—} u'$ in G, and $(u, v) \mathbin{—} (u, v')$ when $v \mathbin{—} v'$ in H.
- $G \diamond H$ is the direct product, aka conjunction, of G and H: Again its vertices are $U \times V$, but its edges are $(u, v) \mathbin{—} (u', v')$ if and only if $u \mathbin{—} u'$ in G and $v \mathbin{—} v'$ in H.
- $G \otimes H$ is the strong product of G and H: As its symbol implies, it combines the edges of $G \times H$ and $G \diamond H$.
- There also are coproducts, analogous to the cosum.

Other notations that are used in this fascicle and not otherwise explained can be found in the Index to Notations at the end of Volumes 1, 2, or 3. Those indexes point to the places where further information is available. Of course Volume 4 will some day contain its own Index to Notations.

CONTENTS

目　　录

CHAPTER SEVEN

COMBINATORIAL SEARCHING

The opening sections of this chapter will appear in Volume 4, Fascicle 0, and Volume 4, Fascicle 1, planned for publication in 2006 and 2007.

7.2. GENERATING ALL POSSIBILITIES

7.2.1. Generating Basic Combinatorial Patterns

OUR GOAL in this section is to study methods for running through all of the possibilities in some combinatorial universe, because we often face problems in which an exhaustive examination of all cases is necessary or desirable. . . .

7.2.1.1. Generating all n-tuples. Let's start small, by considering how to run through all 2^n strings that consist of n binary digits. . . .

7.2.1.2. Generating all permutations. After n-tuples, the next most important item on nearly everybody's wish list for combinatorial generation is the task of visiting all *permutations* of some given set or multiset. . . .

The complete texts of Sections 7.2.1.1 and 7.2.1.2 can be found in Volume 4, Fascicle 2, first published in February 2005.

7.2.1.3. Generating all combinations. Combinatorial mathematics is often described as "the study of permutations, combinations, etc.," so we turn our attention now to combinations. . . .

7.2.1.4. Generating all partitions. Richard Stanley's magnificent book *Enumerative Combinatorics* (1986) begins by discussing The Twelvefold Way, a $2 \times 2 \times 3$ array of basic combinatorial problems that arise frequently in practice. . . . We've learned about n-tuples, permutations, combinations, and compositions in previous sections of this chapter; so we can complete our study of classical combinatorial mathematics by learning about the remaining entries in Stanley's table, which all involve *partitions*. . . .

The *partitions of an integer* are the ways to write it as a sum of positive integers, disregarding order. . . .

7.2.1.5. Generating all set partitions. Now let's shift gears and concentrate on a rather different kind of partition. The *partitions of a set* are the ways to regard that set as a union of nonempty, disjoint subsets. . . .

The complete texts of Sections 7.2.1.3, 7.2.1.4, and 7.2.1.5 can be found in Volume 4, Fascicle 3, first published in July 2005.

Explain the significance of the following sequence:
un, dos, tres, quatre, cinc, sis, set, vuit, nou, deu, . . .

— RICHARD P. STANLEY, *Enumerative Combinatorics* (1999)

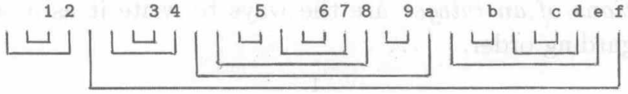
Just as in a single body there are pairs of individual members,
called by the same name but distinguished as right and left,
so when my speeches had postulated the notion of madness,
as a single generic aspect of human nature,
the speech that divided the left-hand portion
repeatedly broke it down into smaller and smaller parts.

— SOCRATES, *Phædrus* 266A (c. 370 B.C.)

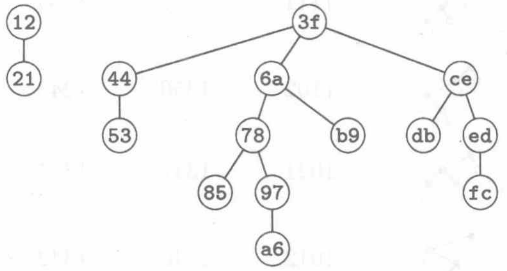
7.2.1.6. Generating all trees. We've now completed our study of the classical concepts of combinatorics: tuples, permutations, combinations, and partitions. But computer scientists have added another fundamental class of patterns to the traditional repertoire, namely the hierarchical arrangements known as trees. Trees sprout up just about everywhere in computer science, as we've seen in Section 2.3 and in nearly every subsequent section of *The Art of Computer Programming*. Therefore we turn now to the study of simple algorithms by which trees of various species can be exhaustively explored.

First let's review the basic connection between nested parentheses and forests of trees. For example,

```
1 2     3 4 5     6 7 8   9 a         b     c d   e f
( ( ) ) ( ( ( ) ) ( ( ( ) ( ( ) ) ) ( ) ) ( ( ) ( ( ) ) ) )     (1)
    1 2       3 4       5     6 7 8   9 a       b     c d e f
```

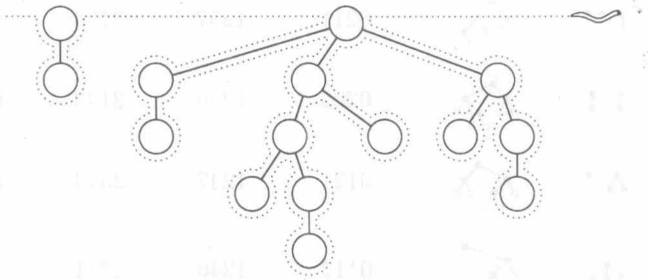


illustrates a string containing fifteen left parens '`(`' labeled `1`, `2`, ..., `f`, and fifteen right parens '`)`' also labeled `1` through `f`; gray lines beneath the string show how the parentheses match up to form fifteen pairs `12`, `21`, `3f`, `44`, `53`, `6a`, `78`, `85`, `97`, `a6`, `b9`, `ce`, `db`, `ed`, and `fc`. This string corresponds to the forest



(2)

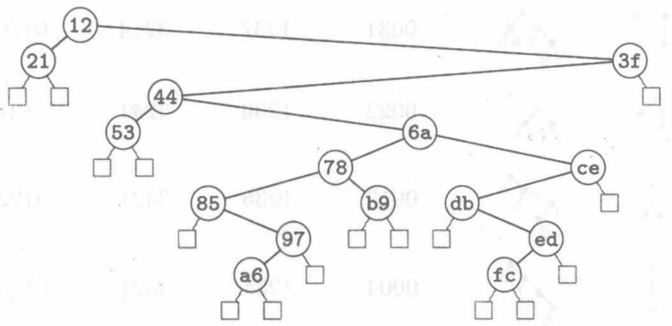
in which the nodes are $\textcircled{\texttt{12}}$, $\textcircled{\texttt{21}}$, $\textcircled{\texttt{3f}}$, ..., $\textcircled{\texttt{fc}}$ in preorder (sorted by first coordinates) and $\textcircled{\texttt{21}}$, $\textcircled{\texttt{12}}$, $\textcircled{\texttt{53}}$, ..., $\textcircled{\texttt{3f}}$ in postorder (sorted by second coordinates). If we imagine a worm that crawls around the periphery of the forest,



(3)

seeing a '`(`' whenever it passes the left edge of a node and a '`)`' whenever it passes a node's right edge, that worm will have reconstructed the original string (1).

The forest in (2) corresponds, in turn, to the binary tree

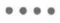
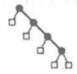
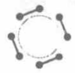
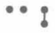
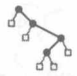
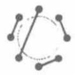
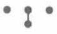
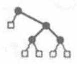
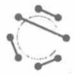
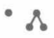
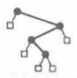
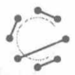
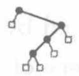
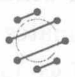
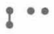
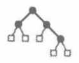
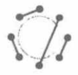
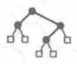
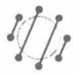
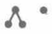
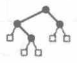
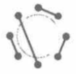
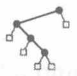
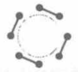
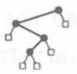
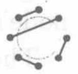
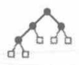
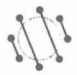
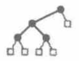
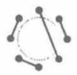
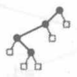
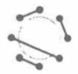
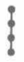
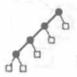
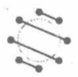


(4)

via the "natural correspondence" discussed in Section 2.3.2; here the nodes are $\textcircled{\texttt{21}}$, $\textcircled{\texttt{12}}$, $\textcircled{\texttt{53}}$, ..., $\textcircled{\texttt{3f}}$ in *symmetric* order, also known as inorder. The left subtree of node $\textcircled{x}$ in the binary tree is the leftmost child of $\textcircled{x}$ in the forest, or it is an "external node" □ if $\textcircled{x}$ is childless. The right subtree of $\textcircled{x}$ in the binary tree is its right sibling in the forest, or □ if $\textcircled{x}$ is the rightmost child in its family. Roots of the trees in the forest are considered to be siblings, and the leftmost root of the forest is the root of the binary tree.

Table 1

NESTED PARENTHESES AND RELATED OBJECTS WHEN $n = 4$

$a_1a_2\dots a_8$	forest	binary tree	$d_1d_2d_3d_4$	$z_1z_2z_3z_4$	$p_1p_2p_3p_4$	$c_1c_2c_3c_4$	matching
()()()()			1111	1357	1234	0000	
()()(())			1102	1356	1243	0001	
()(())()			1021	1347	1324	0010	
()(()())			1012	1346	1342	0011	
()((()))			1003	1345	1432	0012	
(())()()			0211	1257	2134	0100	
(())(())			0202	1256	2143	0101	
(()())()			0121	1247	2314	0110	
(()()())			0112	1246	2341	0111	
(()(()))			0103	1245	2431	0112	
((()))()			0031	1237	3214	0120	
((())())			0022	1236	3241	0121	
((()()))			0013	1235	3421	0122	
(((())))			0004	1234	4321	0123	

A string $a_1a_2\dots a_{2n}$ of parentheses is properly nested if and only if it contains n occurrences of '(' and n occurrences of ')', where the kth '(' precedes the kth ')' for $1 \le k \le n$. The easiest way to explore all strings of nested parentheses is to visit them in lexicographic order. The following algorithm, which considers ')' to be lexicographically smaller than '(', includes some refinements for efficiency suggested by I. Semba [*Inf. Processing Letters* **12** (1981), 188–192]:

Algorithm P (*Nested parentheses in lexicographic order*). Given an integer $n \geq 2$, this algorithm generates all strings $a_1 a_2 \ldots a_{2n}$ of nested parentheses.

```
P1. [Initialize.] Set $a_{2k-1} \leftarrow$ '(' and $a_{2k} \leftarrow$ ')' for $1 \leq k \leq n$; also set $a_0 \leftarrow$ ')'
    and $m \leftarrow 2n-1$.
P2. [Visit.] Visit the nested string $a_1 a_2 \ldots a_{2n}$. (At this point $a_m =$ '(', and
    $a_k =$ ')' for $m < k \leq 2n$.)
P3. [Easy case?] Set $a_m \leftarrow$ ')'. Then if $a_{m-1} =$ ')', set $a_{m-1} \leftarrow$ '(', $m \leftarrow m-1$,
    and return to P2.
P4. [Find $j$.] Set $j \leftarrow m-1$ and $k \leftarrow 2n-1$. While $a_j =$ '(', set $a_j \leftarrow$ ')',
    $a_k \leftarrow$ '(', $j \leftarrow j-1$, and $k \leftarrow k-2$.
P5. [Increase $a_j$.] Terminate the algorithm if $j = 0$. Otherwise set $a_j \leftarrow$ '(',
    $m \leftarrow 2n-1$, and go back to P2. ▍
```

We will see later that the loop in step P4 is almost always short: The operation $a_j \leftarrow$ ')' is performed only about $\frac{1}{3}$ times per nested string visited, on the average.

Why does Algorithm P work? Let A_{pq} be the sequence of all strings α that contain p left parentheses and $q \geq p$ right parentheses, where $(^{q-p}\alpha$ is properly nested, listed in lexicographic order. Then Algorithm P is supposed to generate A_{nn}, where it is easy to see that A_{pq} obeys the recursive rules

$$A_{pq} \;=\;)\,A_{p(q-1)},\ (\,A_{(p-1)q}, \quad \text{if } 0 \leq p \leq q \neq 0; \quad A_{00} = \epsilon; \tag{5}$$

also A_{pq} is empty if $p < 0$ or $p > q$. The first element of A_{pq} is $)^{q-p}() \ldots ()$, where there are p pairs '()'; the last element is $(^p)^q$. Thus the lexicographic generation process consists of scanning from the right until finding a trailing string of the form $a_j \ldots a_{2n} =)(^{p+1})^q$ and replacing it by $()^{q+1-p}() \ldots ()$. Steps P4 and P5 do this efficiently, while step P3 handles the simple case $p = 0$.

Table 1 illustrates the output of Algorithm P when $n = 4$, together with the corresponding forest and binary tree as in (2) and (4). Several other equivalent combinatorial objects also appear in Table 1: For example, a string of nested parentheses can be run-length encoded as

$$()^{d_1}()^{d_2}\ldots()^{d_n}, \tag{6}$$

where the nonnegative integers $d_1 d_2 \ldots d_n$ are characterized by the constraints

$$d_1 + d_2 + \cdots + d_k \leq k \quad \text{for } 1 \leq k < n; \qquad d_1 + d_2 + \cdots + d_n = n. \tag{7}$$

We can also represent nested parentheses by the sequence $z_1 z_2 \ldots z_n$, which specifies the indices where the left parentheses appear. In essence, $z_1 z_2 \ldots z_n$ is one of the $\binom{2n}{n}$ combinations of n things from the set $\{1, 2, \ldots, 2n\}$, subject to the special constraints

$$z_{k-1} < z_k < 2k \qquad \text{for } 1 \leq k \leq n, \tag{8}$$

if we assume that $z_0 = 0$. The z's are of course related to the d's:

$$d_k \;=\; z_{k+1} - z_k - 1 \qquad \text{for } 1 \leq k < n. \tag{9}$$

Algorithm P becomes particularly simple when it is rewritten to generate the combinations $z_1 z_2 \ldots z_n$ instead of the strings $a_1 a_2 \ldots a_{2n}$. (See exercise 2.)

A parenthesis string can also be represented by the permutation $p_1 p_2 \ldots p_n$, where the kth right parenthesis matches the p_kth left parenthesis; in other words, the kth node of the associated forest in postorder is the p_kth node in preorder. (By exercise 2.3.2–20, node j is a descendant of node k in the forest if and only if $j < k$ and $p_j > p_k$, when we label the nodes in postorder.) The inversion table $c_1 c_2 \ldots c_n$ characterizes this permutation by the rule that exactly c_k elements to the right of k are less than k (see exercise 5.1.1–7); allowable inversion tables have $c_1 = 0$ and

$$0 \le c_{k+1} \le c_k + 1 \quad \text{for } 1 \le k < n. \tag{10}$$

Moreover, exercise 3 proves that c_k is the level of the forest's kth node in preorder (the depth of the kth left parenthesis), a fact that is equivalent to the formula

$$c_k = 2k - 1 - z_k. \tag{11}$$

Table 1 and exercise 6 also illustrate a special kind of *matching*, by which $2n$ people at a circular table can simultaneously shake hands without interference.

Thus Algorithm P can be useful indeed. But if our goal is to generate all binary trees, represented by left links $l_1 l_2 \ldots l_n$ and right links $r_1 r_2 \ldots r_n$, the lexicographic sequence in Table 1 is rather awkward; the data we need to get from one tree to its successor is not readily available. Fortunately, an ingenious alternative scheme for direct generation of all linked binary trees is also available:

Algorithm B (*Binary trees*). Given $n \ge 1$, this algorithm generates all binary trees with n internal nodes, representing them via left links $l_1 l_2 \ldots l_n$ and right links $r_1 r_2 \ldots r_n$, with nodes labeled in preorder. (Thus, for example, node 1 is always the root, and l_k is either $k+1$ or 0; if $l_1 = 0$ and $n > 1$ then $r_1 = 2$.)

B1. [Initialize.] Set $l_k \leftarrow k+1$ and $r_k \leftarrow 0$ for $1 \le k < n$; also set $l_n \leftarrow r_n \leftarrow 0$, and set $l_{n+1} \leftarrow 1$ (for convenience in step B3).

B2. [Visit.] Visit the binary tree represented by $l_1 l_2 \ldots l_n$ and $r_1 r_2 \ldots r_n$.

B3. [Find j.] Set $j \leftarrow 1$. While $l_j = 0$, set $r_j \leftarrow 0$, $l_j \leftarrow j+1$, and $j \leftarrow j+1$. Then terminate the algorithm if $j > n$.

B4. [Find k and y.] Set $y \leftarrow l_j$ and $k \leftarrow 0$. While $r_y > 0$, set $k \leftarrow y$ and $y \leftarrow r_y$.

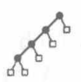
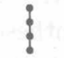
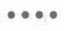
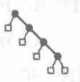
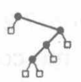
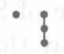
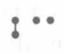
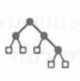
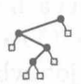
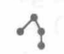
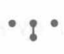
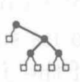
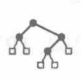
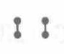
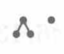
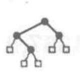
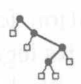
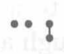
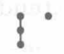
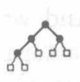
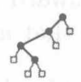
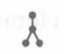
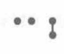
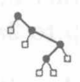
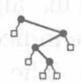
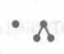
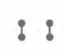
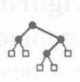
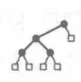
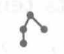
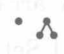
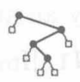
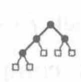
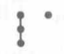
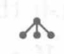
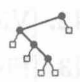
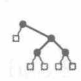
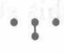
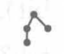
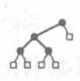
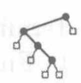
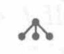
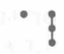
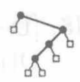
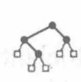
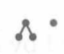
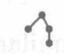
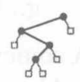
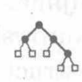
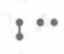
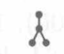
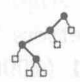
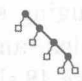
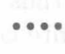
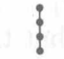
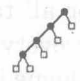
B5. [Promote y.] If $k > 0$, set $r_k \leftarrow 0$; otherwise set $l_j \leftarrow 0$. Then set $r_y \leftarrow r_j$, $r_j \leftarrow y$, and return to B2. ▌

[See W. Skarbek, *Theoretical Computer Science* **57** (1988), 153–159; step B3 uses an idea of J. Korsh.] Exercise 44 proves that the loops in steps B3 and B4 both tend to be very short. Indeed, fewer than 9 memory references are needed, on the average, to transform a linked binary tree into its successor.

Table 2 shows the fourteen binary trees that are generated when $n = 4$, together with their corresponding forests and with two related sequences: Arrays $e_1 e_2 \ldots e_n$ and $s_1 s_2 \ldots s_n$ are defined by the property that node k in preorder has e_k children and s_k descendants in the associated forest. (Thus s_k is the size of k's left subtree in the binary tree; also, $s_k + 1$ is the length of the `SCOPE` link in the sense of 2.3.3–(5).) The next column repeats the fourteen forests of Table 1 in the lexicographic ordering of Algorithm P, but mirror-reversed from left to right.

Table 2

LINKED BINARY TREES AND RELATED OBJECTS WHEN $n = 4$

$l_1l_2l_3l_4$	$r_1r_2r_3r_4$	binary tree	forest	$e_1e_2e_3e_4$	$s_1s_2s_3s_4$	colex forest	lsib/rchild
2340	0000			1110	3210		
0340	2000			0110	0210		
2040	0300			2010	3010		
2040	3000			1010	1010		
0040	2300			0010	0010		
2300	0040			1200	3200		
0300	2040			0200	0200		
2300	0400			2100	3100		
2300	4000			1100	2100		
0300	2400			0100	0100		
2000	0340			3000	3000		
2000	4300			2000	2000		
2000	3040			1000	1000		
0000	2340			0000	0000		

And the final column shows the binary tree that represents the colex forest; it also happens to represent the forest in column 4, but by links to left sibling and right child instead of to left child and right sibling. This final column provides an interesting connection between nested parentheses and binary trees, so it gives us some insight into why Algorithm B is valid (see exercise 19).

***Gray codes for trees.** Our previous experiences with other combinatorial patterns suggest that we can probably generate parentheses and trees by making only small perturbations to get from one instance to another. And indeed, there are at least three very nice ways to achieve this goal.

Consider first the case of nested parentheses, which we can represent by the sequences $z_1z_2\ldots z_n$ that satisfy condition (8). A "near-perfect" way to generate all such combinations, in the sense of Section 7.2.1.3, is one in which we run through all possibilities in such a way that some component z_j changes by ±1 or ±2 at each step; this means that we get from each string of parentheses to its successor by simply changing either `()` $\leftrightarrow$ `)(` or `())` $\leftrightarrow$ `))(` in the vicinity of the jth left parenthesis. Here's one way to do the job when $n=4$:

$$1357, 1356, 1346, 1345, 1347, 1247, 1245, 1246, 1236, 1234, 1235, 1237, 1257, 1256.$$

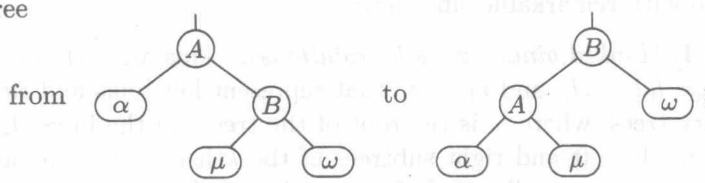
And we can extend any solution for $n-1$ to a solution for n, by taking each pattern $z_1z_2\ldots z_{n-1}$ and letting z_n run through all of its legal values using *endo-order* or its reverse as in 7.2.1.3–(45), proceeding downward from $2n-2$ and then up to $2n-1$ or vice versa, and omitting all elements that are $\le z_{n-1}$.

Algorithm N (*Near-perfect nested parentheses*). This algorithm visits all n-combinations $z_1\ldots z_n$ of $\{1,\ldots,2n\}$ that represent the indices of left parentheses in a nested string, changing only one index at a time. The process is controlled by an auxiliary array $g_1\ldots g_n$ that represents temporary goals.

```
N1. [Initialize.] Set $z_j \leftarrow 2j-1$ and $g_j \leftarrow 2j-2$ for $1 \le j \le n$.
N2. [Visit.] Visit the $n$-combination $z_1\ldots z_n$. Then set $j \leftarrow n$.
N3. [Find $j$.] If $z_j = g_j$, set $g_j \leftarrow g_j \oplus 1$ (thereby complementing the least
    significant bit), $j \leftarrow j-1$, and repeat this step.
N4. [Home stretch?] If $g_j - z_j$ is even, set $z_j \leftarrow z_j + 2$ and return to N2.
N5. [Decrease or turn.] Set $t \leftarrow z_j - 2$. If $t < 0$, terminate the algorithm.
    Otherwise, if $t \le z_{j-1}$, set $t \leftarrow t + 2[t < z_{j-1}] + 1$. Finally set $z_j \leftarrow t$ and
    go back to N2. ▌
```

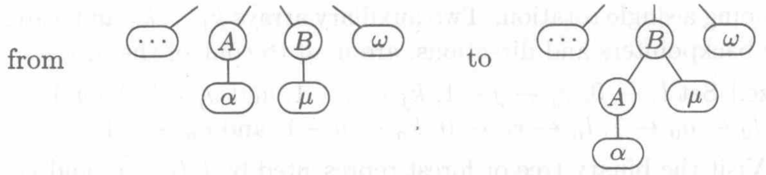
[A somewhat similar algorithm was introduced by D. Roelants van Baronaigien in *J. Algorithms* **35** (2000), 100–107; see also Xiang, Ushijima, and Tang, *Inf. Proc. Letters* **76** (2000), 169–174. F. Ruskey and A. Proskurowski, in *J. Algorithms* **11** (1990), 68–84, had previously shown how to construct *perfect* Gray codes for all tables $z_1\ldots z_n$ when $n\ge4$ is even, thus changing some z_j by only ±1 at every step; but their construction was quite complex, and no known perfect scheme is simple enough to be of practical use. Exercise 48 shows that perfection is impossible when $n\ge5$ is odd.]

If our goal is to generate linked tree structures instead of strings of parentheses, perfection of the z-index changes is not good enough, because simple swaps like `()` $\leftrightarrow$ `)(` don't necessarily correspond to simple link manipulations. A far better approach can be based on the "rotation" algorithms by which we were able to keep search trees balanced in Section 6.2.3. *Rotation to the left* changes a binary tree



(12)

thus the corresponding forest is changed



(13)

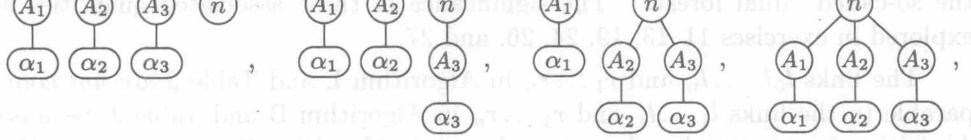
"Node $\textcircled{A}$ becomes the leftmost child of its right sibling." *Rotation to the right* is, of course, the opposite transformation: "The leftmost child of $\textcircled{B}$ becomes its left sibling." The vertical line in (12) stands for a connection to the overall context, either a left link or a right link or the pointer to the root. Any or all of the subtrees α, μ, or ω may be empty. The '$\cdots$' in (13), which represents additional siblings at the left of the family containing $\textcircled{B}$, might also be empty.

The nice thing about rotations is that only three links change: The right link from $\textcircled{A}$, the left link from $\textcircled{B}$, and the pointer from above. Rotations preserve inorder of the binary tree and postorder of the forest. (Notice also that the binary-tree form of a rotation corresponds in a natural way to an application of the *associative law*

$$(\alpha\mu)\omega = \alpha(\mu\omega) \tag{14}$$

in the midst of an algebraic formula.)

A simple scheme very much like the classical reflected Gray code for n-tuples (Algorithm 7.2.1.1H) and the method of plain changes for permutations (Algorithm 7.2.1.2P) can be used to generate all binary trees or forests via rotations. Consider any forest on $n-1$ nodes, with k roots $\textcircled{A_1}, \ldots, \textcircled{A_k}$. Then there are $k+1$ forests on n nodes that have the same postorder sequence on the first $n-1$ nodes but with node $\textcircled{n}$ last; for example, when $k=3$ they are



obtained by successively rotating $\textcircled{A_3}$, $\textcircled{A_2}$, and $\textcircled{A_1}$ to the left. Moreover, at the extremes when $\textcircled{n}$ is either at the right or at the top, we can perform any desired rotation on the other $n-1$ nodes, because node $\textcircled{n}$ isn't in the way. Therefore, as observed by J. M. Lucas, D. Roelants van Baronaigien, and F. Ruskey [*J. Algorithms* **15** (1993), 343–366], we can extend any list of the $(n-1)$-node trees to a list of all n-node trees by simply letting node $\textcircled{n}$ roam back and forth. A careful attention to low-level details makes it possible in fact to do the job with remarkable efficiency:

Algorithm L (*Linked binary trees by rotations*). This algorithm generates all pairs of arrays $l_0l_1\ldots l_n$ and $r_1\ldots r_n$ that represent left links and right links of n-node binary trees, where l_0 is the root of the tree and the links (l_k, r_k) point respectively to the left and right subtrees of the kth node in symmetric order. Equivalently, it generates all n-node forests, where l_k and r_k denote the left child and right sibling of the kth node in postorder. Each tree is obtained from its predecessor by doing a single rotation. Two auxiliary arrays $k_1\ldots k_n$ and $o_0o_1\ldots o_n$, representing backpointers and directions, are used to control the process.

```
L1. [Initialize.] Set $l_j \leftarrow 0$, $r_j \leftarrow j+1$, $k_j \leftarrow j-1$, and $o_j \leftarrow -1$ for $1 \le j < n$;
    also set $l_0 \leftarrow o_0 \leftarrow 1$, $l_n \leftarrow r_n \leftarrow 0$, $k_n \leftarrow n-1$, and $o_n \leftarrow -1$.
L2. [Visit.] Visit the binary tree or forest represented by $l_0l_1\ldots l_n$ and $r_1\ldots r_n$.
    Then set $j \leftarrow n$ and $p \leftarrow 0$.
L3. [Find $j$.] If $o_j > 0$, set $m \leftarrow l_j$ and go to L5 if $m \ne 0$. If $o_j < 0$, set $m \leftarrow k_j$;
    then go to L4 if $m \ne 0$, otherwise set $p \leftarrow j$. If $m = 0$ in either case, set
    $o_j \leftarrow -o_j$, $j \leftarrow j-1$, and repeat this step.
L4. [Rotate left.] Set $r_m \leftarrow l_j$, $l_j \leftarrow m$, $x \leftarrow k_m$, and $k_j \leftarrow x$. If $x = 0$, set
    $l_p \leftarrow j$, otherwise set $r_x \leftarrow j$. Return to L2.
L5. [Rotate right.] Terminate if $j = 0$. Otherwise set $l_j \leftarrow r_m$, $r_m \leftarrow j$, $k_j \leftarrow m$,
    $x \leftarrow k_m$. If $x = 0$, set $l_p \leftarrow m$, otherwise set $r_x \leftarrow m$. Go back to L2. ▌
```

Exercise 38 proves that Algorithm L needs only about 9 memory references per tree generated; thus it is almost as fast as Algorithm B. (In fact, two memory references per step could be saved by keeping the three quantities o_n, l_n, and k_n in registers. But of course Algorithm B can be speeded up too.)

Table 3 shows the sequence of binary trees and forests visited by Algorithm L when $n = 4$, with some auxiliary tables that shed further light on the process. The permutation $q_1q_2q_3q_4$ lists the nodes in preorder, when they have been numbered in postorder of the forest (symmetric order of the binary tree); it is the inverse of the permutation $p_1p_2p_3p_4$ in Table 1. The "coforest" is the conjugate (right-to-left reflection) of the forest; and the numbers $u_1u_2u_3u_4$ are its scope coordinates, analogous to $s_1s_2s_3s_4$ in Table 2. A final column shows the so-called "dual forest." The significance of these associated quantities is explored in exercises 11–13, 19, 24, 26, and 27.

The links $l_0l_1\ldots l_n$ and $r_1\ldots r_n$ in Algorithm L and Table 3 are *not* comparable to the links $l_1\ldots l_n$ and $r_1\ldots r_n$ in Algorithm B and Table 2, because Algorithm L preserves inorder/postorder while Algorithm B preserves preorder. Node k in Algorithm L is the kth node from left to right in the binary tree, so l_0 is needed to identify the root; but node k in Algorithm B is the kth node in preorder, so the root is always node 1 in that case.

Algorithm L has the desired property that only three links change per step; but we can actually do even better in this respect if we stick to the preorder convention of Algorithm B. Exercise 25 presents an algorithm that generates