Efficient Enumeration of Ordered Trees with k Leaves (Extended Abstract)

Katsuhisa Yamanaka[1], Yota Otachi[2], and Shin-ichi Nakano[2]

[1] Graduate School of Information Systems, The University of Electro-Communications, 1-5-1 Chofugaoka, Chofu, Tokyo 182-8585, Japan
yamanaka@is.uec.ac.jp
[2] Department of Computer Science, Gunma University, 1-5-1 Tenjin-cho, Kiryu, Gunma 376-8515, Japan
{otachi@comp.,nakano@}cs.gunma-u.ac.jp

Abstract. In this paper, we give a simple algorithm to generate all ordered trees with exactly n vertices including exactly k leaves. The best known algorithm generates such trees in $O(n - k)$ time for each, while our algorithm generates such trees in $O(1)$ time for each in worst case.

Keywords: graph, algorithm, ordered tree, enumeration, family tree.

1 Introduction

It is useful to have the complete list of objects for a particular class. One can use such a list to search for a counter-example to some conjecture, to find the best object among all candidates, or to experimentally measure an average performance of an algorithm over all possible inputs.

Many algorithms to generate all objects in a particular class, without repetition, are already known [1,2,11,13,12,15,16,17,20,26,28]. Many excellent textbooks have been published on the subject [4,6,10,25].

Trees are the most fundamental models frequently used in many areas, including searching for keys, modeling computation, parsing a program, etc. From the point of view, a lot of enumeration algorithms for trees are proposed [2,11,15,17,18,23,26], and a great textbook has been published by Knuth [7]. Also, enumeration algorithms for some subclasses of trees are known [5].

A *rooted* tree means a tree with one designated "root" vertex. Note that there is no ordering among the children of each vertex. Beyer and Hedetniemi [2] gave an algorithm to generate all rooted trees with n vertices. Their algorithm is the first one to generate all rooted trees in $O(1)$ time per tree on average, and based on the level sequence representation. Li and Ruskey [11] also gave an algorithm to generate all such trees, and showed that it was easily modified to generate restricted classes of rooted trees. The possible restrictions are (1) upper bound on the number of children, and (2) lower and upper bounds on the height of a rooted tree.

S. Das and R. Uehara (Eds.): WALCOM 2009, LNCS 5431, pp. 141–150, 2009.
© Springer-Verlag Berlin Heidelberg 2009

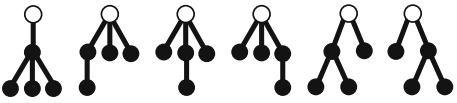

Fig. 1. All rooted ordered trees with 5 vertices including 3 leaves

A tree without the root vertex is called a *free tree*. Due to the absence of the root vertex, the generation of nonisomorphic free trees is a more difficult problem. Wright et al. [26], and Li and Ruskey [11] gave algorithms to generate all free trees in $O(1)$ time per tree on average, then Nakano and Uno [17] improved the running time to $O(1)$ time in worst case. Also they generalized the algorithm to generate all "colored" trees [18], where a colored tree is a tree in which each vertex has a color.

An *ordered* tree means a rooted tree with a left-to-right ordering specified for the children of each vertex. An algorithm to generate all ordered trees has been proposed by Nakano [15]. He also gave a method to generate non-rooted ordered trees in [15]. Sawada [23] handled enumeration problem for similar but different class of trees, called *circular*-ordered trees. A circular-ordered tree is a rooted tree with a circular ordering specified for the children of each vertex. Sawada [23] gave algorithms to generate circular-ordered trees and non-rooted ones in $O(1)$ time per tree on average.

In this paper, we wish to generate all ordered trees with exactly n vertices including exactly k leaves. See Fig. 1 for examples.

Let $S_{n,k}$ be the set of ordered trees with exactly n vertices including exactly k leaves. For instance there are six ordered trees with exactly 5 vertices including 3 leaves, as shown in Fig. 1 in which the root vertices are depicted by white circles, and $|S_{5,3}| = 6$. Such trees are one of the most natural subclasses of trees and are researched extensively, including enumeration [15,19], counting [24, p.237] and random generation [14].

The number of trees in $S_{n,k}$ is known as the Narayana number [24, p.237] as follows:

$$|S_{n,k}| = \frac{\binom{n-2}{k-1}\binom{n-1}{k-1}}{k}.$$

Two algorithms to generate all trees in $S_{n,k}$ are already known. Pallo [19] gave an algorithm to generate each tree in $S_{n,k}$ in $O(n-k)$ time on average. Also, Nakano's algorithm in [15] generates each tree in $S_{n,k}$ in $O(n-k)$ time on average.

By combining an algorithm to generate all ordered trees with specified degree sequence [8,9,22,29, etc], and a slightly modified version of an algorithm to generate all integer partitions into $(n-k)$ parts [3,21,27,30, etc], one can design an algorithm to generate all trees in $S_{n,k}$. Although such algorithm may generate each tree in $O(1)$ time in worst case, the algorithm is very complicated.

In this paper, we give a simple and efficient algorithm to generate all trees in $S_{n,k}$. Our algorithm generates each tree in $S_{n,k}$ in $O(1)$ time in worst case. The main idea of our algorithms is as follows. For some graph enumeration problems

(biconnected triangulation [12], triconnected triangulations [16], plane graphs [28] and ordered trees [15]) we can define a simple tree structure among the graphs, called the family tree, in which each vertex corresponds to each graph to be enumerated. In this paper, we design a cleverer family tree than the one in [15].

The rest of the paper is organized as follows. Section 2 gives some definitions. Section 3 defines the family tree among trees in $S_{n,k}$. Section 4 gives a simple algorithm to generate all trees in $S_{n,k}$.

2 Definitions

In this section, we give some definitions.

Let G be a connected graph with n vertices. In this paper, all graphs are unlabeled. The *degree* of a vertex v, denoted by $d(v)$, is the number of neighbors of v in G. A *tree* is a connected graph with no cycle. A *rooted* tree is a tree with one vertex r chosen as its *root*. For each vertex v in a rooted tree, let $UP(v)$ be the unique path from v to r. If $UP(v)$ has exactly k edges then we say *the depth* of v is k. *The parent* of $v \neq r$ is its neighbor on $UP(v)$, and *the ancestors* of $v \neq r$ are the vertices on $UP(v)$ except v. The parent of r and the ancestors of r are not defined. We say if v is the parent of u then u is a *child* of v, and if v is an *ancestor* of u then u is a *descendant* of v. A *leaf* is a vertex having no child. If a vertex is not a leaf, then it is called an *inner* vertex.

An *ordered tree* is a rooted tree with a left-to-right ordering specified for the children of each vertex. For an ordered tree T with the root r, let $LP(T) = (l_0(= r), l_1, l_2, \ldots, l_p)$ be the path such that l_i is the leftmost child of l_{i-1} for each i, $1 \leq i \leq p$, and l_p is a leaf of T. We call $LP(T)$ *the leftmost path* of T, and l_p *the leftmost leaf* of T. Similarly, let $RP(T) = (r_0(= r), r_1, r_2, \ldots, r_q)$ be the path such that r_i is the rightmost child of r_{i-1} for each i, $1 \leq i \leq q$, and r_q is a leaf of T. We call $RP(T)$ *the rightmost path* of T, and r_q *the rightmost leaf* of T.

3 The Family Tree

Let $S_{n,k}$ be the set of all ordered trees with exactly n vertices including exactly k leaves. In this section, we define a tree structure among the trees in $S_{n,k}$ in which each vertex corresponds to a tree in $S_{n,k}$.

We need some definitions.

The root tree, denoted by $R_{n,k}$, of $S_{n,k}$ is the tree consisting of the leftmost path $(l_0(= r), l_1, \ldots, l_{n-k})$ and $k-1$ leaves attaching at vertex l_{n-k-1}. See Fig. 2 for an example.

Then we define *the parent tree*, denoted by $P(T)$, of each tree T in $S_{n,k} \backslash \{R_{n,k}\}$ as follows. Let l_p and r_q be the leftmost leaf and rightmost leaf in T. We have two cases.

Fig. 2. The root tree $R_{7,4}$

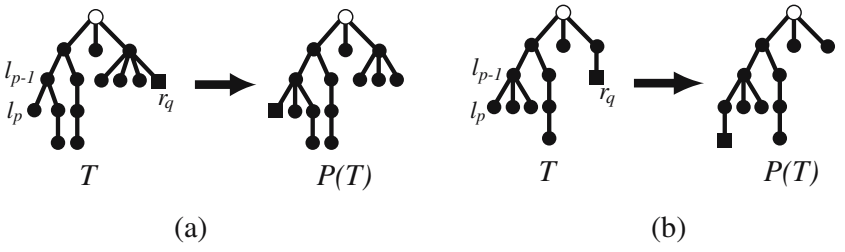

Fig. 3. Examples of the parents in (a) Case 1 and (b) Case 2

Case 1: r_{q-1} has two or more children.

$P(T)$ is the tree obtained from T by (1) removing r_q, then (2) attaching a new leaf to l_{p-1} as the leftmost child of l_{p-1}. See Fig. 3(a) for an example. The removed and attached vertices are depicted by boxes.

Case 2: r_{q-1} has only one child r_q.

$P(T)$ is the tree obtained from T by (1) removing r_q, then (2) attaching a new leaf to l_p. See Fig. 3(b) for an example.

Note that $P(T)$ is also in $S_{n,k}$.

T is called a *child tree* of $P(T)$. If T is a child tree in Case 1, then T is called *Type 1 child*, otherwise, T is *Type 2 child*.

Lemma 1. *For any* $T \in S_{n,k} \setminus \{R_{n,k}\}$, $P(T) \in S_{n,k}$ *holds.*

Given a tree T in $S_{n,k} \setminus \{R_{n,k}\}$, by repeatedly finding the parent tree of the derived tree, we can have the unique sequence $T, P(T), P(P(T)), \ldots$ of trees in $S_{n,k}$ which is called *the removing sequence* of T. See Fig. 4 for an example. in which each solid line corresponds to the relation with Case 1, and each dashed line corresponds to the relation with Case 2.

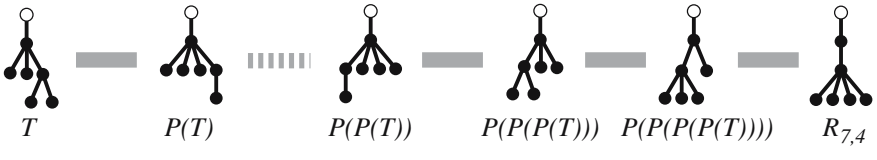

Fig. 4. The sequence of trees

Lemma 2. *The removing sequence ends up with the root tree $R_{n,k}$.*

Proof. Let T be a tree in $S_{n,k} \setminus \{R_{n,k}\}$. Let $LP(T) = (l_0, l_1, \ldots, l_p)$ be the leftmost path of T. We define two functions $f(T)$ and $g(T)$ as follows. We define that $f(T) = |LP(T)|$. Let $c_1, c_2, \ldots, c_a$ be the children of l_{p-1} from left to right. We choose the minimum i such that c_i is an inner vertex. Then we define that $g(T) = i - 1$. For convenience, if there is no such vertex, then we define that $g(T) = a$. Note that $1 \leq f(T) \leq n - k + 1$ and $1 \leq g(T) \leq k$ for any T in $S_{n,k}$.

Now we define a potential function $p(T) = (f(T), g(T))$. It is not difficult to see that $p(T) = (n - k + 1, k)$ if and only if $T = R_{n,k}$. Suppose that T_1 and T_2 are two trees in $S_{n,k}$ such that $T_1 \neq T_2$. We denote $p(T_1) < p(T_2)$ if (1) $f(T_1) < f(T_2)$ or (2) $f(T_1) = f(T_2)$ and $g(T_1) < g(T_2)$.

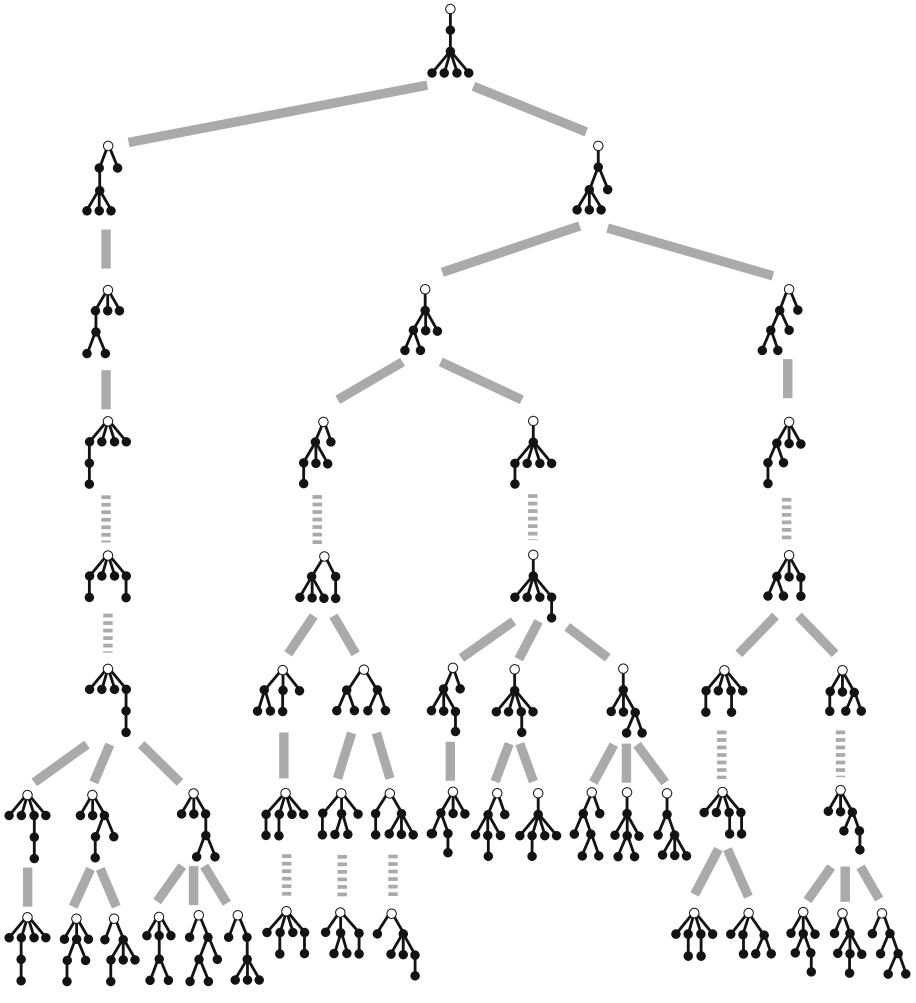

Fig. 5. The family tree $T_{7,4}$

Next we show that $p(T) < p(P(T))$. Suppose T is a Type 1 child of $P(T)$ (see Fig. 3(a)). In this case, we have $f(T) = f(P(T))$ and $g(T) + 1 = g(P(T))$. Thus $p(T) < p(P(T))$ holds. If T is a Type 2 child of $P(T)$, we always have $f(T) + 1 = f(P(T))$. Thus $p(T) < p(P(T))$ holds.

Therefore, by repeatedly finding the parent of the derived tree, we eventually obtain $R_{n,k}$ on which the potential is maximized. This completes the proof. □

By merging removing sequences we can have *the family tree* $T_{n,k}$ of $S_{n,k}$ such that the vertices of $T_{n,k}$ correspond to the trees in $S_{n,k}$ and each edge correspond to the relation between some T and $P(T)$. See Fig. 5 for an example.

4 Algorithm

Let $S_{n,k}$ be the set of ordered trees with exactly n vertices including exactly k leaves. This section gives our algorithm to generate all trees in $S_{n,k}$ by traversing $T_{n,k}$.

Given $S_{n,k}$ we can construct $T_{n,k}$ by the definition, possibly with huge space and much running time. However, how can we construct $T_{n,k}$ efficiently only given two integers n, k? Our idea [12,15,16,28] is by reversing the procedure finding the parent tree as follows.

If $k = 1$, $S_{n,k}$ includes only one element which is the path with $n - 1$ edges, then generation is trivial. Also if $k = n - 1$, $S_{n,k}$ includes only the star of n vertices. Therefore, from now on we assume $1 < k < n - 1$.

Let $T \in S_{n,k}$. Let $LP(T) = (l_0(= r), l_1, \ldots, l_p)$ be the leftmost path of T, and l_p the leftmost leaf of T. Let $RP(T) = (r_0(= r), r_1, \ldots, r_q)$ be the rightmost path of T, and r_q the rightmost leaf of T. We denoted by $T[r_i]$, $0 \leq i \leq q$, the tree obtained from T by (1) removing the leftmost leaf and (2) attaching a new leaf to r_i as the rightmost child of r_i.

Now we explain an algorithm to generate all child trees of the given tree T in $S_{n,k}$. We have the following two cases.

Case 1: T is the root tree $R_{n,k}$.

Each $T[r_i]$, $0 \leq i \leq q - 2$, is a child of T, since $P(T[r_i]) = T$. Since $T[r_{q-1}]$ is isomorphic to the root tree $R_{n,k}$ in $S_{n,k}$, $T[r_{q-1}]$ is not a child tree of T. Since $P(T[r_q]) \neq T$, $T[r_q]$ is not a child of T.

Thus T has $q - 1$ of Type 1 children and no Type 2 child.

Case 2: T is not the root tree.

If l_{p-1} has two or more children, and the second child of l_{p-1} from left is not a leaf, then T has no child tree, since if T is the parent of some tree then the second child of l_{p-1} from left is a leaf (Case 1), or l_{p-1} has only one child and it is a leaf (Case 2). See Fig. 3. Now we assume otherwise. We have the following two subcases.

Case 2(a): l_{p-1} has two or more children.

Let T' be the tree obtained from T by removing l_p. Then T' has $k - 1$ leaves. Thus we should add a new vertex to T' so that in the resulting graph the number of leaves increases by one. The detail is as follows.

Each $T[r_i]$, $0 \leq i \leq q-1$, is a child tree of T, since $P(T[r_i]) = T$. On the other hand, $T[r_q]$ is not a child tree of T, since $P(T[r_q]) \neq T$.

Thus T has q of Type 1 children and no Type 2 child.

Case 2(b): l_{p-1} has only one child, which is l_p.

Any $T[r_i]$, $0 \leq i \leq q-1$, is not a child tree of T. $T[r_q]$ is the child tree of T. Thus T has exactly one Type 2 child.

The above case analysis gives the following algorithm.

```
    Procedure find-all-child-trees(T)
    begin
01    Output T
      {Output the difference from the previous tree.}
02    Let l_p and r_q be the leftmost leaf and the rightmost leaf of T.
03    Let RP(T) = (r_0(= r), r_1, r_2, ..., r_q) be the rightmost path of T.
04    if l_{p-1} has two or more children and the second child of l_{p-1} from left
      is not a leaf then
05      return
06    if l_{p-1} has two or more children then
07      for i = 0 to q - 1
08        find-all-child-trees(T[r_i])     {Case 2(a)}
09    else
10      find-all-child-trees(T[r_q])     {Case 2(b)}
    end

    Algorithm find-all-trees(n, k)
    begin
01    Output R_{n,k}
02    T = R_{n,k}
03    Let RP(T) = (r_0(= r), r_1, r_2, ..., r_q) be the rightmost path.
04    for i = 0 to q - 2
05      find-all-child-trees(T[r_i]) {Case 1}
    end
```

We have the following theorem.

Theorem 1. *The algorithm uses $O(n)$ space and runs in $O(|S_{n,k}|)$ time, where $|S_{n,k}|$ is the number of ordered trees with exactly n vertices including exactly k leaves.*

Proof. To construct $T[r_i]$ from T, our algorithm needs the references to the leftmost leaf l_p and the rightmost path of T. Each can be updated as follows. In Case 2(a) the second child of l_{p-1} from left becomes the leftmost leaf of $T[r_i]$. In Case 2(b) the parent l_{p-1} of l_p becomes the leftmost leaf of $T[r_i]$. In both cases the rightmost path is updated to the path from the newly added vertex to its root. Thus we can maintain in $O(1)$ time the leftmost leaf and the rightmost path. □

The algorithm generates all trees in $S_{n,k}$ in $O(|S_{n,k}|)$ time. Thus the algorithm generates each tree in $O(1)$ time "on average." However, after generating a tree corresponding to the last vertex in a large subtree of $T_{n,k}$, we have to merely return from the deep recursive call without outputting any tree. This may take much time. Therefore, the next tree cannot be generated in $O(1)$ time in worst case.

However, a simple modification [17] improves the algorithm to generate each tree in $O(1)$ time in worst case. The algorithm is as follows.

Procedure find-all-children2(T, *depth*)
{ T is the current tree, and *depth* is the depth of the recursive call.}
begin
01 **if** *depth* is even **then**
02 Output T {before outputting its child trees.}
03 Generate child trees by the method in the first algorithm, and recursively call **find-all-children2** for each child tree.
04 **if** *depth* is odd **then**
05 Output T {after outputting its child trees.}
end

One can observe that the algorithm generates all trees so that each tree can be obtained from the preceding one by tracing at most three edges of $T_{n,k}$. Note that if tree T corresponds to a vertex v in $T_{n,k}$ with odd depth, then we may need to trace three edges to generate the next tree. Otherwise we need to trace at most two edges to generate the next tree. Note that each tree is similar to the preceding one, since it can be obtained with at most three operations. Therefore we have the following theorem.

Theorem 2. *One can generate ordered trees with exactly n vertices including exactly k leaves in $O(1)$ time for each in worst case.*

5 Conclusion

In this paper, we have given an efficient algorithm to generate all ordered trees with exactly n vertices including exactly k leaves.

We defined a cleverer family tree than the one in [15]. By traversing the family tree, our algorithm generates all trees in $S_{n,k}$ in $O(1)$ time for each in worst case.

References

1. Avis, D., Fukuda, K.: Reverse search for enumeration. Discrete Appl. Math. 65(1-3), 21–46 (1996)
2. Beyer, T., Hedetniemi, S.M.: Constant time generation of rooted trees. SIAM J. Comput. 9(4), 706–712 (1980)
3. Fenner, T.I., Loizou, G.: A binary tree representation and related algorithms for generating integer partitions. The Computer J. 23(4), 332–337 (1980)

4. Goldberg, L.: Efficient algorithms for listing combinatorial structures. Cambridge University Press, New York (1993)
5. Kikuchi, Y., Tanaka, H., Nakano, S., Shibata, Y.: How to obtain the complete list of caterpillars. In: Warnow, T.J., Zhu, B. (eds.) COCOON 2003. LNCS, vol. 2697, pp. 329–338. Springer, Heidelberg (2003)
6. Knuth, D.: The art of computer programming. Generating all tuples and permutations, vol. 4, fascicle 2. Addison-Wesley, Reading (2005)
7. Knuth, D.E.: The art of computer programming. Generating all trees, history of combinatorial generation, vol. 4, fascicle 4. Addison-Wesley, Reading (2006)
8. Korsh, J.F., LaFollette, P.: Multiset permutations and loopless generation of ordered trees with specified degree sequence. Journal of Algorithms 34(2), 309–336 (2000)
9. Korsh, J.F., LaFollette, P.: Loopless generation of trees with specified degrees. The Computer Journal 45(3), 364–372 (2002)
10. Kreher, D.L., Stinson, D.R.: Combinatorial algorithms. CRC Press, Boca Raton (1998)
11. Li, G., Ruskey, F.: The advantages of forward thinking in generating rooted and free trees. In: Proc. 10th Annual ACM-SIAM Symp. on Discrete Algorithms (SODA 1999), pp. 939–940 (1999)
12. Li, Z., Nakano, S.: Efficient generation of plane triangulations without repetitions. In: Orejas, F., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 433–443. Springer, Heidelberg (2001)
13. McKay, B.D.: Isomorph-free exhaustive generation. J. Algorithms 26(2), 306–324 (1998)
14. Muramatsu, T., Nakano, S.: A random generation of plane trees with exactly k leaves. IEICE Transaction on Fundamentals J90-A(12), 940–947 (2007) (in Japanese)
15. Nakano, S.: Efficient generation of plane trees. Inf. Process. Lett. 84(3), 167–172 (2002)
16. Nakano, S.: Efficient generation of triconnected plane triangulations. Comput. Geom. Theory and Appl. 27(2), 109–122 (2004)
17. Nakano, S., Uno, T.: Constant time generation of trees with specified diameter. In: Hromkovič, J., Nagl, M., Westfechtel, B. (eds.) WG 2004. LNCS, vol. 3353, pp. 33–45. Springer, Heidelberg (2004)
18. Nakano, S., Uno, T.: Generating colored trees. In: Kratsch, D. (ed.) WG 2005. LNCS, vol. 3787, pp. 249–260. Springer, Heidelberg (2005)
19. Pallo, J.: Generating trees with n nodes and m leaves. International Journal of Computer Mathematics 21(2), 133–144 (1987)
20. Read, R.C.: Every one a winner or how to avoid isomorphism search. Annuals of Discrete Mathematics 2, 107–120 (1978)
21. Reingold, E.M., Nievergelt, J., Deo, N.: Combinatorial Algorithms. Prentice-Hall, Englewood Cliffs (1977)
22. Ruskey, F., van Baronaigien, D.R.: Fast recursive algorithms for generating combinatorial objects. Congressus Numerantium 41, 53–62 (1984)
23. Sawada, J.: Generating rooted and free plane trees. ACM Transactions on Algorithms 2(1), 1–13 (2006)
24. Stanley, R.P.: Enumerative combinatorics, vol. 2. Cambridge University Press, Cambridge (1999)
25. Wilf, H.S.: Combinatorial algorithms: An update. SIAM, Philadelphia (1989)
26. Wright, R.A., Richmond, B., Odlyzko, A., McKay, B.D.: Constant time generation of free trees. SIAM J. Comput. 15(2), 540–548 (1986)

27. Yamanaka, K., Kawano, S., Kikuchi, Y., Nakano, S.: Constant time generation of integer partitions. IEICE Trans. Fundamentals E90-A(5), 888–895 (2007)
28. Yamanaka, K., Nakano, S.: Listing all plane graphs. In: Nakano, S.-i., Rahman, M. S. (eds.) WALCOM 2008. LNCS, vol. 4921, pp. 210–221. Springer, Heidelberg (2008)
29. Zaks, S., Richards, D.: Generating trees and other combinatorial objects lexicographically. SIAM J. Comput. 8(1), 73–81 (1979)
30. Zoghbi, A., Stojmenović, I.: Fast algorithms for generating integer partitions. Int. J. Comput. Math. 70, 319–332 (1998)