Information Processing Letters 77 (2001) 231–238
Information Processing Letters
www.elsevier.com/locate/ipl

On generating k-ary trees in computer representation ☆

Limin Xiang [a,*], Kazuo Ushijima [a], Changjie Tang [b]

[a] *Department of Computer Science and Communication Engineering, Kyushu University, 6-10-1 Hakozaki, Higashi-ku, Fukuoka 812-8581, Japan*
[b] *Department of Computer Science, Sichuan University, 610064 Chengdu, Sichuan, China*

Received 16 May 2000; received in revised form 7 September 2000
Communicated by F.Y.L. Chin

Abstract

Many algorithms have been developed to generate sequences for trees, and a few are to generate trees themselves, i.e., in computer representation. Two new algorithms are presented in this paper to generate k-ary trees in computer representation. One is recursive with constant average time per tree and the other is loopless (non-recursive) with constant time per tree. Both of them are simple and able to be understood easily.

Keywords: k-ary trees; Computer representation; Loopless generation; Algorithms

1. Introduction

There have been many algorithms developed for generating (rooted and ordered binary or k-ary) trees [1–16,18,19,21–27]. Most of them list 1–1 correspondent sequences (codes) for trees, and a few generate trees in computer representation [10,11,18]. Solomon and Finkel [18] enumerate binary trees with n nodes in time $O(n)$ per tree, Lucas et al. [11] list binary trees in constant time per tree, and Korsh and Lipschutz [10] generate k-ary trees in constant time per tree.

In this paper, from Zaks' sequences [26] (a kind of well-formed integer sequences for k-ary trees) and Williamson's loopless algorithm [20], a recursive and a loopless algorithm are presented for generating k-ary trees in computer representation. They list k-ary trees in constant average time and constant time per tree, respectively. It is shown that Williamson's loopless algorithm for generating sequences in Gray code order can also be used to obtain a loopless algorithm for generating k-ary trees themselves well, not only the sequences. The two algorithms presented in this paper are simple and able to be understood easily.

2. The recursive algorithm

Zaks established a 1–1 correspondence between regular k-ary trees with n internal nodes and a set of integer sequences with length n, and gave a non-recursive algorithm to generate the sequences in constant average time per tree [26]. The set, denoted by $\mathcal{Z}_{n,k}$, is

☆ This research was supported by the JSPS-RFTF-96P00603.
* Corresponding author.
E-mail addresses: xiang@csce.kyushu-u.ac.jp (L. Xiang), ushijima@csce.kyushu-u.ac.jp (K. Ushijima), chjtang@scu.edu.cn (C. Tang).

0020-0190/01/$ – see front matter
PII: S0020-0190(00)00155-1

```
void gen_codeZ(int i)
{ z[i] = z[i − 1] + 1;
  if (i == n) show_codeZ(); else gen_codeZ(i + 1);
  do
  { z[i] = z[i] + 1;
    if (i == n) show_codeZ(); else gen_codeZ(i + 1);
  } while (z[i]! = (i − 1) ∗ k + 1);
}
```

Fig. 1. A recursive algorithm to generate $\mathcal{Z}_{n,k}$.

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

Fig. 2. $\mathcal{Z}_{3,3}$ generated by *gen_codeZ*().

$$\mathcal{Z}_{n,k} = \{(z_1, z_2, \ldots, z_n) \mid 1 = z_1 < z_2 < \cdots < z_n \text{ and } z_i \leqslant (i-1) * k + 1 \text{ for } 1 \leqslant i \leqslant n\}.$$

Since $\mathcal{Z}_{1,k} = \{(1)\}$ and $\mathcal{Z}_{n,1} = \{(1, 2, \ldots, n)\}$, i.e., $|\mathcal{Z}_{1,k}| = |\mathcal{Z}_{n,1}| = 1$, we hereafter assume that $n \geqslant 2$ and $k \geqslant 2$.

A k-ary tree T with n nodes represented by $(z_1, z_2, \ldots, z_n)$ means that the n nodes are numbered from 1 to n in preorder and z_i is the position of the ith node in the preorder traversal of T for all n nodes and $n * (k-1) + 1$ leaves. From the definition of $\mathcal{Z}_{n,k}$, a simple recursive algorithm can be obtained easily to generate the sequences of $\mathcal{Z}_{n,k}$ lexicographically. The recursive algorithm is given in Fig. 1 (the algorithm can be simplified as "for $(z[i] = z[i-1] + 1;\ z[i] <= (i-1) * k + 1;\ z[i]++)$ {if $(i == n)$ show_codeZ(); else *gen_codeZ*$(i+1)$;}"). Thus, all the sequences of $\mathcal{Z}_{n,k}$ will be listed lexicographically by doing $z[1] = 1$ and *gen_codeZ*(2). An example is shown in Fig. 2 for $\mathcal{Z}_{3,3}$ generated by *gen_codeZ*().

Let the n nodes of a k-ary tree T represented by $(z_1, z_2, \ldots, z_n)$ be $v_1, v_2, \ldots, v_n$ in preorder, and the subtree with nodes $v_1, v_2, \ldots, v_i$ be denoted by T_i for $1 \leqslant i \leqslant n$. For $1 < i \leqslant n$, given $(z_1, z_2, \ldots, z_{i-1})$, z_i will be a value in $\{z_{i-1} + 1, z_{i-1} + 2, \ldots, (i-1) * k + 1\}$. Since $z_i = z_{i-1} + 1$ is corresponding to v_i in T_i being the first son of v_{i-1}, $z_i = z_{i-1} + j$ to v_i in T_i being at the jth leaf of T_{i-1} on the right from the leaf being the first son of v_{i-1}, and $z_i = (i-1) * k + 1$ to v_i in T_i being the rightmost leaf of T_{i-1}, a recursive algorithm can be obtained from Algorithm *gen_codeZ*() above to generate trees themselves. This algorithm, denoted by *gen_tree*(), is shown in Fig. 3, which can be viewed as an application of the general backtracking algorithm in [17]. An example is given in Fig. 4 for 3-ary trees with 3 nodes generated by Algorithm *gen_tree*().

```
void gen_tree(int i)
{ let node v_i be the first son of node v_{i−1};
  if (i == n) tree_print(); else gen_tree(i + 1);
  do
  { move node v_i to the next leaf of tree T_{i−1} on its right;
    if (i == n) tree_print(); else gen_tree(i + 1);
  } while (node v_i is not the rightmost leaf of tree T_{i−1});
}
```

Fig. 3. A recursive algorithm to generate trees.

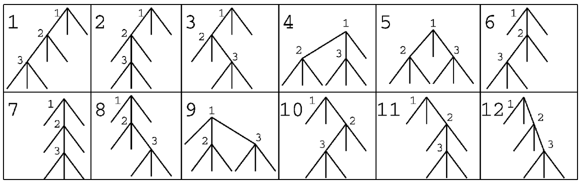

Fig. 4. 3-ary trees with 3 nodes generated by *gen_tree*().

Theorem 1. *Algorithm gen_tree() generates trees in constant average time per tree.*

Proof. Conventionally, the number of recursive calls is used as a measure of the time complexity for recursive algorithms [5,21,24,27]. Since the number of recursive calls is the same for algorithms *gen_tree*() and *gen_codeZ*(), the number of recursive calls of Algorithm *gen_codeZ*() is considered here for simplicity.

By using the inductive method on n, it is easily proved that the number of recursive calls is $\sum_{i=1}^{n-1} |\mathcal{Z}_{i,k}|$ for Algorithm *gen_codeZ*() to generate sequences of $\mathcal{Z}_{i,k}$. Therefore, the average number of recursive calls for each sequence is

$$\frac{\sum_{i=1}^{n-1} |\mathcal{Z}_{i,k}|}{|\mathcal{Z}_{n,k}|}.$$

Since

$$\frac{\sum_{i=1}^{n} |\mathcal{Z}_{i,k}|}{|\mathcal{Z}_{n,k}|} = \mathrm{O}(1) \quad [26],$$

$$\frac{\sum_{i=1}^{n-1} |\mathcal{Z}_{i,k}|}{|\mathcal{Z}_{n,k}|} = \frac{\sum_{i=1}^{n} |\mathcal{Z}_{i,k}|}{|\mathcal{Z}_{n,k}|} - 1 = \mathrm{O}(1).$$

This completes the proof. □

3. The non-recursive algorithm

In [22], the sequences of $\mathcal{Z}_{n,k}$ are generated in Gray code order by a loopless algorithm obtained from Williamson's loopless algorithm [20] (another independently discovered loopless algorithm can be found in [4] for generating $\mathcal{Z}_{n,k}$ in Gray code order, and it was not from Williamson's loopless algorithm). In this section, it will be shown that a loopless algorithm can also be obtained from Williamson's for generating k-ary trees themselves (i.e., computer representation by pointers), not only the sequences for k-ary trees.

Williamson's loopless algorithm is used to generate variations in Gray code order, i.e., elements of the product space

$$S = S_1 \times S_2 \times \cdots \times S_n, \quad \text{with}$$

$$S_i = \{0, 1, \ldots, r[i]-1\} \ (r[i] > 1 \text{ is a constant}) \quad \text{for } 1 \leqslant i \leqslant n,$$

such that two consecutive generated sequences differ in a single position [19,20]. Williamson's loopless algorithm which computes the successor of a sequence $v = (v[1], v[2], \ldots, v[n])$ in Gray code order can be described as in Fig. 5, where

```
void succ()
{ e[n+1] = n;
  if (d[j] == 1) v[j]++; else v[j]--;
  if ((v[j] == 0)||(v[j] == r[j]-1))
  { d[j] = 1 - d[j]; e[j+1] = e[j]; e[j] = j-1; }
  j = e[n+1];
}
```

Fig. 5. Williamson's loopless algorithm.

j:	indicates the position at which the value of v (i.e., $v[j]$) is to be changed for the successor,
$e[1] \sim e[n+1]$:	keep track of positions for the change of $v[j]$, and
$d[1] \sim d[n]$:	indicate the directions for the change of each $v[i]$, for $i = 1, 2, \ldots, n$, i.e., $v[i]$ is to be added or subtracted by 1 according to $d[i] == 1$ or 0, respectively.

The initial values of the variables above are
- $v[i] = 0$ and $d[i] = 1$ for $1 \leqslant i \leqslant n$,
- $e[i] = i - 1$ for $1 \leqslant i \leqslant n + 1$, and
- $j = n$.

With the first sequence $v[i] = 0$ ($1 \leqslant i \leqslant n$), *succ*() generates all the successors in Gray code order step by step until $j == 0$. The explanation in more detail for Williamson's loopless algorithm can be found in [19, 20,22].

An important property of Algorithm *succ*() is given in Lemma 2, which will be a basis to obtain a loopless algorithm for generating k-ary trees in computer representation.

Lemma 2. *For Algorithm succ*(), *when $j < i \leqslant n$ the value of $v[i]$ is its minimal* (*i.e.*, 0) *or maximal* (*i.e.*, $r[i]-1$) *one.*

Proof. From Algorithm *succ*(), it is known that the value of j gets smaller from the initial value n, i.e., a carry arises, only when $((v[j] == 0)||(v[j] == r[j]-1))$, and the value of j will return to n at once after the carry. This completes the proof. □

With Lemma 2, we can prove the following result.

Theorem 3. *A loopless algorithm can be obtained by modifying Williamson's loopless algorithm to generate k-ary trees in computer representation.*

Proof. From Algorithm *gen_tree*() in Section 2, it is known that for $2 \leqslant i \leqslant n$, v_i in T can be at a position in the leaves of T_{i-1} from $Lmin_i$ to $Lmax_i$, where, $Lmin_i$ is the first son of node v_{i-1} and $Lmax_i$ is the rightmost leaf of T_{i-1}. Let $d[i] == 1$ correspond to the direction for the change of v_i from $Lmin_i$ to $Lmax_i$, called *up*, and $d[i] == 0$ to that from $Lmax_i$ to $Lmin_i$, called *down*, respectively. If Williamson's algorithm is to be applied for obtaining the successor T' of T, for v_j to be moved, the following four cases should be considered.

(1) *v_j is to be moved up, and its next position P_N is not $Lmax_j$.* By Lemma 2, each node v_i with $j < i \leqslant n$ in T is at its $Lmin_i$ or $Lmax_i$, i.e., position P_N is a leaf of T. Therefore, T' is obtained only by moving v_j to P_N, and each node v_i with $j < i \leqslant n$ in T' is still at its $Lmin_i$ or $Lmax_i$,

(2) *v_j is to be moved up, and its next position P_N is $Lmax_j$.* By Lemma 2, each node v_i with $j < i \leqslant n$ in T is at its $Lmin_i$ or $Lmax_i$, i.e., the kth son of v_j is a leaf of T and there may be a node v_a $(j < a \leqslant n)$ at P_N (i.e., $Lmax_a == Lmax_j$). Therefore, T' is obtained by moving v_a, if any, to the kth son of v_j and v_j to P_N, and each node v_i with $j < i \leqslant n$ in T' is still at its $Lmin_i$ or $Lmax_i$,

(3) *v_j is to be moved down, and its current position P_C is not $Lmax_j$.* By Lemma 2, each node v_i with $j < i \leqslant n$ in T is at its $Lmin_i$ or $Lmax_i$, i.e., the position P_D to which v_j is to be moved *down* is a leaf of T. Therefore, T' is obtained only by moving v_j to P_D, and each node v_i with $j < i \leqslant n$ in T' is still at its $Lmin_i$ or $Lmax_i$, and

(4) *v_j is to be moved down, and its current position P_C is $Lmax_j$.* By Lemma 2, each node v_i with $j < i \leqslant n$ in T is at its $Lmin_i$ or $Lmax_i$, i.e., the position P_D to which v_j is to be moved *down* is a leaf of T, and there may be a node v_a $(j < a \leqslant n)$ at the kth son of v_j. Therefore, T' is obtained by moving v_j to P_D and v_a, if any, to P_C, and each node v_i with $j < i \leqslant n$ in T' is still at its $Lmin_i$ or $Lmax_i$.

From the discussion above, it is known that the successor T' of T can be obtained from T by moving one or two nodes. Thus, a loopless algorithm can be obtained by modifying Williamson's loopless algorithm to generate k-ary trees in computer representation. □

By Theorem 3, the loopless algorithm is given in Fig. 6 for generating k-ary trees in computer representation, where the functions, as well as the initial values, of $d[i]$ $(1 \leqslant i \leqslant n)$, e_i $(1 \leqslant i \leqslant n+1)$ and j are the same as those for Algorithm *succ*(). Obviously, the first tree is that for $2 \leqslant i \leqslant n$, node v_i is at $Lmin_i$, i.e., the first son of node v_{i-1}. Since there is

```
void tree_succ()
{ e[n+1] = n;
  if (d[j] == 1) move v_j up; else move v_j down;
  if (v_j is at Lmin_j or Lmax_j)
  { d[j] = 1 - d[j]; e[j+1] = e[j]; e[j] = j - 1; }
  j = e[n+1];
}
```

Fig. 6. Loopless algorithm to generate trees themselves.

only one position (the root) for node v_1, *tree_succ*() generates k-ary trees until $j = 1$, instead of $j = 0$. A complete implementation of the loopless algorithm is given in Appendix A to generate k-ary trees themselves.

4. Conclusion

Two new algorithms *gen_tree* and *tree_succ* have been presented in this paper for generating k-ary trees in computer representation. The recursive algorithm *gen_tree* generates k-ary trees just in the reversed order of Definition 2 in [26], which is also called *B-order* [13] or *local order* [5]. A recursive algorithm can be obtained easily from Algorithm *gen_tree* to generate k-ary trees in B-order, which is left to the reader. The loopless algorithm *tree_succ* lists k-ary trees in the same order as that of Algorithm *next_tree* in [10], which is the first loopless algorithm (and only one found in the literature) for generating k-ary trees in computer representation. Based on the *shift graph* $\mathrm{SG}_{n,k}$, Algorithm *next_tree* threads all nodes with a *chain* and uses *finished* and *unfinished* lists for *shifting* each node *up* or *down*. Using Williamson's method, Algorithm *tree_succ* is conceptually simple, and it keeps all nodes in an array and uses another array e to keep track of indices for *moving* each node *up* or *down*. To obtain the next tree from the current tree,

- in the best/worst case, *next_tree* needs $4/11$ assignment or (composite) comparison operations on pointers and a call of *shift_up*() or *shift_down*(), and *tree_succ* needs $4/7$ assignment or simple comparison operations on integers in arrays and a call of *move_up*() or *move_down*();

Table 1
Some actual running times

(n,k)	(7, 7)	(7, 8)	(7, 9)	(7, 10)	(8, 7)	(8, 8)	(8, 9)	(8, 10)
T_{XUT}	$1.75''$	$4.06''$	$8.34''$	$16.03''$	$25.65''$	$1'07.83''$	$2'38.89''$	$5'39.76''$
T_{KL}	$2.36''$	$5.49''$	$11.36''$	$21.69''$	$34.54''$	$1'31.45''$	$3'34.15''$	$7'41.04''$
$T_{\mathrm{XUT}}/T_{\mathrm{KL}}$	74.15%	73.95%	73.42%	73.91%	74.26%	74.17%	74.19%	73.69%

- in the best/worst case, a call of *shift_up*() executes $11/16$ assignment or comparison operations, and a call of *move_up*() executes $10/16$ assignment or comparison operations;
- in the best/worst case, a call of *shift_down*() executes $13/17$ assignment or comparison operations, and a call of *move_down*() executes $9/15$ assignment or comparison operations.

Therefore, our loopless algorithm is more efficient than the loopless one in [10], which is also confirmed by the *actual running times* in Table 1, where the same computer and the same compiler were used for the two algorithms, and in each case *printing a tree* was replaced with *adding* 1 *to a counter*.

In addition, as mentioned in [22], Williamson' loopless algorithm has a characteristic that variations can be generated in different orders by changing some initial values only. With the similar method in [22], for $2 \leqslant i \leqslant n$, initially, let v_i be at $Lmin_i$ or $Lmax_i$ (i.e., there are 2^{n-1} choices) and $d[i]$ be 1 (for v_i at $Lmin_i$) or 0 (for v_i at $Lmax_i$), k-ary trees with n nodes can be generated by *tree_succ* in 2^{n-1} different orders.

Acknowledgements

The authors would like to thank the referees for helpful comments.

Appendix A. A complete implementation of the loopless algorithm

```
#include <stdio.h>
#define Null 0
#define Len sizeof(struct node_type)

typedef struct node_type
{ struct node_type *parent,*succ,*pred,*subtree[22];
  int pos,s_pos;
} *tree_type;

int e[22],d[22],n,k,j,z,bound;
long tree_num;
tree_type tree, node[22];

void get_n_k()
{ printf("N= (2-20):\n"); do scanf("%d",&n); while ((n<2)||(n>20));
  printf("K= (2-20):\n"); do scanf("%d",&k); while ((k<2)||(k>20));
  printf("\n");
}
```

```
void print_tree(tree_type Atree)
{ int s;
  z++;
  if (Atree)
  { printf("%3d",z);
    for (s=1;s<=k;s++) print_tree(Atree->subtree[s]);
  };
}

void print_tree_ln()
{ z=0; print_tree(tree); printf("  No.%d\n",++tree_num); }

void init()
{ int i,s;

  for (i=1; i<=n; i++) node[i]=(tree_type)malloc(Len);
  node[0]=Null; node[n+1]=Null;
  for (i=1; i<=n; i++)
  { node[i]->parent=node[i-1];
    node[i]->succ=node[i-1];
    node[i]->pred=node[i];
    node[i]->pos=1;
    node[i]->s_pos=2;
    node[i]->subtree[1]=node[i+1];
    for (s=2; s<=k; s++) node[i]->subtree[s]=Null;
    e[i]=i-1; d[i]=1;
  }
  tree=node[1]; tree->succ=Null; j=n; tree_num=0;
}
void move_up()
{ tree_type vj;

  vj=node[j];

  vj->parent->subtree[vj->pos]=Null;

  if (vj->succ->subtree[vj->s_pos])
  { vj->subtree[k]=vj->succ->subtree[k];
    vj->subtree[k]->parent=vj;
  }

  if (vj->parent!=vj->succ)
    vj->succ->subtree[vj->s_pos-1]->pred=vj->parent;

  vj->parent=vj->succ;
  vj->pos=vj->s_pos;
  vj->parent->subtree[vj->pos]=vj;

  bound=0;
  if (vj->pos<k) vj->s_pos=vj->pos+1; else
  { vj->succ=vj->parent->succ;
    vj->s_pos=vj->parent->s_pos;
```

```
    if (vj->succ)
      vj->succ->subtree[vj->s_pos-1]->pred=vj; else
      bound=1;
  }
}

void move_down()
{ tree_type vj, v;

  vj=node[j];

  vj->parent->subtree[vj->pos]=Null;

  if ((vj->succ==Null)&&(vj->subtree[k]))
  { vj->subtree[k]->parent=vj->parent;
    vj->parent->subtree[k]=vj->subtree[k];
    vj->subtree[k]=Null;
  }
  else if ((vj->succ)&&(vj->pos==k))
    vj->succ->subtree[vj->s_pos-1]->pred=vj->parent;

  if (vj->parent->subtree[vj->pos-1]==Null)
  { vj->s_pos=vj->pos--;
    vj->parent->subtree[vj->pos]=vj;
    vj->succ=vj->parent;
    bound=(vj->pos==1);
  } else
  { bound=0;
    v=vj->parent->subtree[vj->pos-1]->pred;
    vj->parent->subtree[vj->pos-1]->pred=vj;
    v->subtree[k]=vj;
    vj->succ=vj->parent;
    vj->parent=v;
    vj->s_pos=vj->pos;
    vj->pos=k;
  }
}

void tree_succ()
{ e[n+1]=n;
  if (d[j]) move_up(); else move_down();
  if (bound) { d[j]=1-d[j]; e[j+1]=e[j]; e[j]=j-1; }
  j=e[n+1];
}

int main()
{ get_n_k(); init(); print_tree_ln();
  do { tree_succ(); print_tree_ln(); } while (j>1);
  printf("Number of Trees=%d\n",tree_num); return 0;
}
```

References

[1] S.G. Akl, I. Stojmenovic, Generating t-ary trees in parallel, Nordic J. Comput. 3 (1996) 63–71.

[2] V. Bapiraju, V.V. Bapeswara Rao, Enumeration of binary trees, Inform. Process. Lett. 51 (1994) 125–127.

[3] D. Roelants van Baronaigien, A loopless algorithm for generating binary tree sequences, Inform. Process. Lett. 39 (1991) 189–194.

[4] D. Roelants van Baronaigien, A loopless Gray-code algorithm for listing k-ary trees, J. Algorithms 35 (2000) 100–107.

[5] M.C. Er, Enumerating ordered trees lexicographically, Comput. J. 28 (1985) 538–542.

[6] M.C. Er, Efficient generation of k-ary trees in natural order, Comput. J. 35 (1992) 306–308.

[7] G.D. Knott, A numbering system for binary trees, Comm. ACM 20 (1977) 113–115.

[8] J.F. Korsh, Loopless generation of k-ary tree sequences, Inform. Process. Lett. 52 (1994) 243–247.

[9] J.F. Korsh, P. LaFollette, Loopless generation of Gray codes for k-ary trees, Inform. Process. Lett. 70 (1999) 7–11.

[10] J.F. Korsh, S. Lipschutz, Shifts and loopless generation of k-ary trees, Inform. Process. Lett. 65 (1998) 235–240.

[11] J.M. Lucas, D. Roelants van Baronaigien, F. Ruskey, On rotations and the generation of binary trees, J. Algorithms 15 (1993) 343–366.

[12] J. Pallo, Enumerating, ranking and unranking binary trees, Comput. J. 29 (1986) 171–175.

[13] J. Pallo, R. Racca, A note on generating binary trees in A-order and B-order, Intern. J. Comput. Math. 18 (1985) 27–39.

[14] A. Proskurowski, On the generation for binary trees, J. ACM 27 (1980) 1–2.

[15] D. Rotem, Y.L. Varol, Generation of binary trees from Ballot sequences, J. ACM 25 (1978) 396–404.

[16] F. Ruskey, Generating t-ary trees lexicographically, SIAM J. Comput. 7 (1978) 424–439.

[17] F. Ruskey, D. Roelants van Baronaigien, Fast recursive algorithms for generating combinatorial objects, Congr. Numer. 41 (1984) 53–62.

[18] M. Solomon, R.A. Finkel, A note on enumerating binary trees, J. ACM 27 (1980) 3–5.

[19] V. Vajnovszki, On the loopless generation of binary tree sequences, Inform. Process. Lett. 68 (1998) 113–117.

[20] S.G. Williamson, Combinatorics for Computer Science, Computer Science Press, Rockville, MD, 1985.

[21] L. Xiang, C. Tang, K. Ushijima, Grammar-oriented enumeration of binary trees, Comput. J. 40 (1997) 278–291.

[22] L. Xiang, K. Ushijima, C. Tang, Efficient loopless generation of Gray codes for k-ary trees, Inform. Process. Lett. 76 (2000) 169–174.

[23] L. Xiang, K. Ushijima, Iterative formulas for enumerating binary trees, Research Report on Inform. Sci. and Elect. Engin. of Kyushu Univ. 2 (1997) 179–183.

[24] L. Xiang, K. Ushijima, Grammar-oriented enumeration of arbitrary trees and arbitrary k-ary trees, IEICE Trans. Inf. Syst. E82-D (1999) 1245–1253.

[25] L. Xiang, K. Ushijima, S.G. Akl, Generating regular k-ary trees efficiently, Comput. J. (2000), to appear.

[26] S. Zaks, Lexicographic generation of ordered trees, Theoret. Comput. Sci. 10 (1980) 63–82.

[27] D. Zerling, Generating binary trees using rotations, J. ACM 32 (1985) 694–701.