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. © 2001 Elsevier Science B.V. All rights reserved.

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 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 $Z_{n,k}$, is...
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 \( Z_{n,k} \).

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. \( Z_{3,3} \) generated by \( \text{gen}_\_\text{code}Z() \).

Since \( Z_{1,k} = \{1\} \) and \( Z_{n,1} = \{(1, 2, \ldots, n)\} \), i.e., \(|Z_{1,k}| = |Z_{n,1}| = 1\), we hereafter assume that \( n \geq 2 \) and \( k \geq 2 \).

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 \( i \)th node in the preorder traversal of \( T \) for all \( n \) nodes and \( n \cdot (k - 1) + 1 \) leaves. From the definition of \( Z_{n,k} \), a simple recursive algorithm can be obtained easily to generate the sequences of \( Z_{n,k} \) lexicographically. The recursive algorithm is given in Fig. 1 (the algorithm can be simplified as “for \((z[i] = z[i-1] + 1; z[i] <= (i - 1) * k + 1; \) if (i == n) show_codeZ(); else \( \text{gen}_\_\text{code}Z(i + 1); \)”). Thus, all the sequences of \( Z_{n,k} \) will be listed lexicographically by doing \( z[1] = 1 \) and \( \text{gen}_\_\text{code}Z(2) \). An example is shown in Fig. 2 for \( Z_{3,3} \) generated by \( \text{gen}_\_\text{code}Z() \).

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 \( 1 \leq i \leq n \). For \( 1 < i \leq n \), 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 \( v_j \) in \( T_i \) being the \( j \)th 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 \( \text{gen}_\_\text{code}Z() \) above to generate trees themselves. This algorithm, denoted by \( \text{gen}_\_\text{tree}() \), 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 \( \text{gen}_\_\text{tree}() \).

**Theorem 1.** Algorithm \( \text{gen}_\_\text{tree}() \) 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 \( \text{gen}_\_\text{tree}() \) and \( \text{gen}_\_\text{code}Z() \), the number of recursive calls of Algorithm \( \text{gen}_\_\text{code}Z() \) is considered here for simplicity.

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

\[
\frac{\sum_{i=1}^{n-1}|Z_{i,k}|}{|Z_{n,k}|}.
\]
Since
\[
\sum_{i=1}^{n} |Z_{i,k}| \leq |Z_{n,k}| = O(1) \quad [26],
\]
\[
\sum_{i=1}^{n} \frac{|Z_{i,k}|}{|Z_{n,k}|} = \frac{\sum_{i=1}^{n} |Z_{i,k}|}{|Z_{n,k}|} - 1 = O(1).
\]
This completes the proof. □

3. The non-recursive algorithm

In [22], the sequences of \(Z_{n,k}\) 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 \(Z_{n,k}\) in Gray code order, and it was not from Williamson’s loopless algorithm). In this section, it will be shown that a loopless algorithm for generating \(Z_{n,k}\) in Gray code order can be independently discovered loopless algorithm can be found in [19, 20, 22].

Williamson’s loopless algorithm is used to generate variations in Gray code order, i.e., elements of the product space \(S = S_1 \times S_2 \times \cdots \times S_n\), with

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

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

\[
j:
\]
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 \(1 \leq i \leq n\),
- \(e[i] = i - 1\) for \(1 \leq i \leq n + 1\), and
- \(j = n\).

With the first sequence \(v[i] = 0\) \((1 \leq i \leq n)\), \(\text{succ()}\) 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 \(\text{succ()}\) 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 \(\text{succ()}\), when \(j < i \leq n\) the value of \(v[i]\) is its minimal (i.e., 0) or maximal (i.e., \(r[i] - 1\)) one.

**Proof.** From Algorithm \(\text{succ()}\), 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 \(\text{gen_tree()}\) in Section 2, it is known that for \(2 \leq i \leq n\), \(v_i\) in \(T\) can be at a position in the leaves of \(T_{i-1}\) from \(L_{min_i}\) to \(L_{max_i}\), where, \(L_{min_i}\) is the first son of node \(v_{i-1}\) and \(L_{max_i}\) is the rightmost leaf of \(T_{i-1}\). Let \(d[i] == 1\) correspond to the direction for the change of \(v_i\) from \(L_{min_i}\) to \(L_{max_i}\), called \(up\), and \(d[i] == 0\) to that from \(L_{max_i}\) to \(L_{min_i}\).
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 $j < i < n$ 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 $j < i < n$ 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 $j < i < n$ in $T$ is at its $Lmin_i$ or $Lmax_i$, i.e., the $k$th son of $v_j$ is a leaf of $T$ and there may be a node $v_a$ ($j < a < n$) at $P_N$ (i.e., $Lmax_a == Lmax_j$). Therefore, $T'$ is obtained by moving $v_a$, if any, to the $k$th son of $v_j$ and $v_j$ to $P_N$, and each node $v_i$ with $j < i < n$ 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 $j < i < n$ 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 $j < i < n$ in $T'$ is still at its $Lmin_i$ or $Lmax_i$.

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 $j < i < n$ 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 $v_a$ ($j < a < n$) at the $k$th 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 $j < i < n$ 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 $d[i]$ ($1 \leq i \leq n$), $e[i]$ ($1 \leq i \leq n + 1$) and $j$ are the same as those for Algorithm succ(). Obviously, the first tree is that for $2 \leq i \leq n$, node $v_1$ is at $Lmin_1$, i.e., the first son of node $v_{i-1}$. Since there is 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 $SG_{n,k}$, 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 $4n/11$ assignment or (composite) comparison operations on pointers and a call of shift_up() or shift_down(), and tree_succ needs $4n/7$ assignment or simple comparison operations on integers in arrays and a call of move_up() or move_down();

```c
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.
Table 1
Some actual running times

<table>
<thead>
<tr>
<th>(n, k)</th>
<th>(7, 7)</th>
<th>(7, 8)</th>
<th>(7, 9)</th>
<th>(7, 10)</th>
<th>(8, 7)</th>
<th>(8, 8)</th>
<th>(8, 9)</th>
<th>(8, 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{XUT}$</td>
<td>1.75&quot;</td>
<td>4.06&quot;</td>
<td>8.34&quot;</td>
<td>16.03&quot;</td>
<td>25.65&quot;</td>
<td>1'07.83&quot;</td>
<td>2'38.89&quot;</td>
<td>5'39.76&quot;</td>
</tr>
<tr>
<td>$T_{KL}$</td>
<td>2.36&quot;</td>
<td>5.49&quot;</td>
<td>11.36&quot;</td>
<td>21.69&quot;</td>
<td>34.54&quot;</td>
<td>1'31.45&quot;</td>
<td>3'34.15&quot;</td>
<td>7'41.04&quot;</td>
</tr>
<tr>
<td>$T_{XUT}/T_{KL}$</td>
<td>74.15%</td>
<td>73.95%</td>
<td>73.42%</td>
<td>73.91%</td>
<td>74.26%</td>
<td>74.17%</td>
<td>74.19%</td>
<td>73.69%</td>
</tr>
</tbody>
</table>

- 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’s 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 $2 \leq i \leq n$, initially, let $v_i$ be at $L_{min}$ or $L_{max}$ (i.e., there are $2^{n-1}$ choices) and $d[i]$ be 1 (for $v_i$ at $L_{min}$) or 0 (for $v_i$ at $L_{max}$), $k$-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

```c
#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->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[vj->pos]=vj->subtree[k];
        vj->subtree[k]=Null;
    }
    else if ((vj->succ)&&(vj->pos==k))
    {
        vj->succ->subtree[vj->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