A Constant Amortized Time Algorithm for Generating Left-Child Sequences in Lexicographic Order

Kung-Jui Pai[1], Jou-Ming Chang[2(✉)], and Ro-Yu Wu[3]

[1] Department of Industrial Engineering and Management, Ming Chi University of Technology, New Taipei City, Taiwan
poter@mail.mcut.edu.tw
[2] Institute of Information and Decision Sciences, National Taipei University of Business, Taipei, Taiwan
spade@ntub.edu.tw
[3] Department of Industrial Management, Lunghwa University of Science and Technology, Taoyuan, Taiwan
eric@mail.lhu.edu.tw

Abstract. Wu et al. (Theoret. Comput. Sci. 556:25–33, 2014) recently introduced a new type of sequences, called left-child sequences (LC-sequences for short), for representing binary trees. They pointed out that such sequences have a natural interpretation from the view point of data structure and gave a characterization of them. Based on this characterization, Pai et al. (International conference on combinatorial optimization and applications. Springer, Cham, pp. 505–518, 2016) showed that there is an easily implementing algorithm that uses generate-and-test approach to filter all LC-sequences of binary trees with n internal nodes in lexicographic order, while in general this algorithm is not efficient at all. In this paper, we design two novel rotations that allow us to drastically alter the shape of binary trees (and thus their corresponding LC-sequences). As an application, these operations can be employed to generate all LC-sequences in lexicographic order. Accordingly, we present a more efficient algorithm associated with the new types of rotations for generating all LC-sequences and show that it takes only constant amortized running cost.

Keywords: Constant amortized time algorithm · Binary trees · Left-child sequences · Lexicographic order · Generation algorithms · Amortized cost

1 Introduction

Binary trees are one of the most fundamental data structures in computer science and have been widely studied over half a century. Usually, binary trees are encoded by using integer sequences and many types of integer sequences have been introduced (e.g., see [9,11] for surveys). For convenience, hereafter the terms

M. Xiao and F. Rosamond (Eds.): FAW 2017, LNCS 10336, pp. 221–232, 2017.
DOI: 10.1007/978-3-319-59605-1_20

of binary trees and their corresponding sequences are often used interchangeably. Due to many practical applications in computer science, such as combinatorial object search or algorithm performance analysis, exhaustively generating all binary tree sequences is an important issue in research topic. Generation algorithms customarily produce sequences in a specific ordering, such as lexicographic order [16,22,23] or Gray-code order [14,17]. For algorithmic efficiency, in general, sequences generated in lexicographic order is demanded to run in constant amortized time [2,3,18]. By contrast, sequences generated in Gray-code order is in need of taking a constant time for each generation (e.g., the so-called loopless algorithms proposed by Ehrlich [4]). For more references of binary tree sequences generation, we refer to [10,13–16,19,20].

Recently, Wu et al. [20] proposed a loopless algorithm associated with the usual tree rotations (i.e., left rotation and right rotation for AVL-trees) to generate four types of binary tree sequences simultaneously. In particular, the generation includes two new types of sequences called *left-child sequences* (LC-sequences for short) and their mirror images called *right-child sequences* (RC-sequences for short), as defined later in Sect. 2. It is well-known that the practice in implementing binary trees usually adopted the so-called structure-pointer representation so that the spaces of nodes in a tree are dynamically allocated by structured memory and children of nodes are accessed via pointers. Wu et al. [20] thereby claimed that LC- and RC-sequences are inspired by such a natural structure representation. Moreover, they gave characterizations of the two types of sequences (see Theorem 1). However, both LC- and RC-sequences generated in [20] are not in lexicographic order or Gray-code order. In fact, the difference between two successive LC-sequences (resp., RC-sequences) in the generated list is either one or two digits.

Later on, based on the characterization of LC-sequences provided in [20], Pai et al. [12] showed that there is an algorithm using generate-and-test approach that allows developers to easily implement for generating all LC-sequences of binary trees with n internal nodes in lexicographic order (see Procedure Lex-Gen-Tree in Sect. 3), while this algorithm is quite not efficient. Indeed, the purpose of [12] is to develop efficient ranking algorithm (i.e., a function that determines the rank of a given sequence in the generated list) and unranking algorithm (i.e., a function that produces the sequence corresponding to a given rank) of LC-sequences in lexicographic order. As expected, their ranking and unranking algorithms can be run in amortized cost of $\mathcal{O}(n)$ time and space. Since the difference between two consecutive LC-sequences in the lexicographic order may vary widely, it means that the shapes of corresponding binary trees are possibly changed drastically. To adapt to this unavoidable situation, in this paper we design two massive rotations which can deal with a great variety of changes to assist our generation. As a result, we develop an algorithm called Refined-Lex-Gen-Tree that associates with these tree rotations to generate all LC-sequences in lexicographic order. Moreover, we show that this algorithm is more efficient and has constant amortized running cost. By symmetry, generation of RC-sequences can be developed by a similar way.

The rest of this paper is organized as follows. In Sect. 2, we formally give the definitions of LC- and RC-sequences, and introduces a coding trees structure for representing all LC-sequences in lexicographic order. In Sect. 3, we define two new types of rotations for binary trees, and then propose a constant amortized-time algorithm associated with these rotations for generating all LC-sequences in lexicographic order. Finally, concluding remarks are given in the last section.

2 Preliminaries

An *extended binary tree* is a rooted and ordered tree such that every internal node has exactly two children called the *left child* and the *right child* [9]. Let T be an extended binary tree with n internal nodes numbered from 1 to n in inorder (i.e., visit recursively the left subtree, the root and then the right subtree of T). Henceforth, we shall not distinguish the terms between a node and its inorder number. For a node $i \in T$, the subtree rooted at i is denoted by T_i. Also, the subtree rooted at the left child (resp., right child) of i is called the *left subtree* (resp., *right subtree*) of i and is denoted by L_i (resp., R_i). The *left arm* (resp., *right arm*) of T is the path from the root to its leftmost leaf (resp., rightmost leaf).

2.1 Left-Child Sequences

Recently, Wu et al. [20] introduced new types of sequences called *left-child sequence* (LC-sequence for short) and *right-child sequence* (RC-sequence for short) to represent binary trees. Given a binary T with n internal nodes labeled by $1, 2, \ldots, n$ in inorder, the LC-sequence of T, denoted by $\ell(T) = (\ell[1], \ell[2], \ldots, \ell[n])$, is an integer sequence so that the term $\ell[i]$, $1 \leqslant i \leqslant n$, is defined as follows:

$$\ell[i] = \begin{cases} 0 & \text{if the left child of } i \text{ is a leaf;} \\ j & \text{if } j \text{ is the left child of } i \text{ in } T. \end{cases} \tag{1}$$

Similarly, $r(T) = (r[1], r[2], \ldots, r[n])$ denotes the RC-sequence of T, where we use the right child instead of the left child in Eq. (1) to define the term $r[i]$. For instance, the LC-sequence and RC-sequence of the binary tree T shown in Fig. 1 are $\ell(T) = (0, 1, 0, 3, 0, 2, 0, 0, 7)$ and $r(T) = (0, 4, 0, 5, 0, 9, 8, 0, 0)$, respectively.

Wu et al. [20] showed that the two types of binary tree sequences can transform to each other in linear time. Moreover, they characterized the two types of sequences as follows.

Theorem 1 (Wu et al. [20]). *Let $c = (c_1, c_2, \ldots, c_n)$ be an integer sequence. Then,*

(a) *c is the LC-sequence of a binary tree T with n internal nodes if and only if the following conditions are fulfilled for all $i \in \{1, 2, \ldots, n\}$: (1) $0 \leqslant c_i < i$ and (2) $c_j = 0$ or $c_j > c_i$ for all $c_i < j < i$.*

(b) *c is the RC-sequence of a binary tree T with n internal nodes if and only if the following conditions are fulfilled for all $i \in \{1, 2, \ldots, n\}$: (1) $c_i > i$ or $c_i = 0$ and (2) $c_j = 0$ or $c_j < c_i$ for all $i < j < c_i$.*

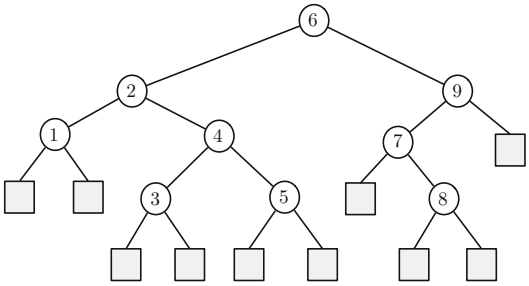

Fig. 1. A binary tree T with LC-sequence $\ell(T) = (0, 1, 0, 3, 0, 2, 0, 0, 7)$ and RC-sequence $r(T) = (0, 4, 0, 5, 0, 9, 8, 0, 0)$.

2.2 Coding Tree Structure

Let $\mathscr{T}_n$ be the set of binary trees with n internal nodes. It is well-known that $|\mathscr{T}_n| = \frac{1}{n+1}\binom{2n}{n}$ (i.e., the Catalan number). To depict all binary tree sequences, a systematic way by using coding trees was suggested in [10]. For a rooted tree, a path from the root to a leaf is called a *full path*. A *coding tree* $\mathbb{T}_n$ is a rooted tree consisting of n levels of nodes such that every node is associated with a label and the labels along a full path in $\mathbb{T}_n$ represent the sequence of a binary tree with n internal nodes. Figure 2 demonstrates the coding tree $\mathbb{T}_5$ for representing LC-sequences, where each node x_i in a full path $(x_1, x_2, \ldots, x_n)$ is labeled by $\ell[i]$. For notational convenience, we also write $\ell(x_i) = \ell[i]$ when we provide the full path $(x_1, x_2, \ldots, x_n)$ corresponding to a binary tree T.

For instance, in $\mathbb{T}_5$, labels in the left arm represent the right-skewed tree with LC-sequence $(0, 0, 0, 0, 0)$, and labels in the right arm represent the left-skewed tree with LC-sequence $(0, 1, 2, 3, 4)$. Hereafter, we consider a specific coding tree $\mathbb{T}_n$ in which all LC-sequences of binary trees are emerged from left to right in lexicographic order.

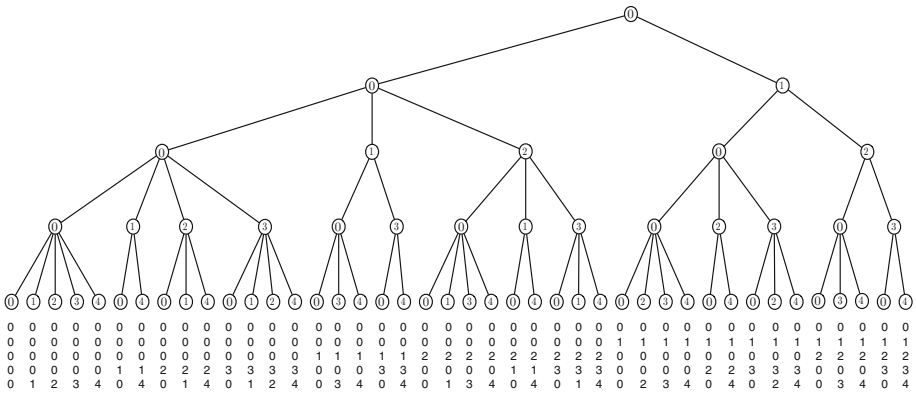

Fig. 2. A coding tree $\mathbb{T}_4$ for representing LC-sequences and LD-sequences.

3 Generating LC-sequences in Lexicographic Order

Based on the characterization of LC-sequences described in Theorem 1, Pai et al. [12] showed that there is an easy implementing algorithm (see Fig. 3) to generate all LC-sequences of binary trees with n internal nodes in lexicographic order. In this algorithm, the outer loop specifies the range of condition (a.1) in Theorem 1, and the **if** $\cdots$ **then** statement in the inner loop is the testing of condition (a.2) in Theorem 1. Initially, we set $\ell[1] = 0$, and then perform a procedure call Lex-Gen-Tree(2) to start the generation.

```
Procedure Lex-Gen-Tree(i)
begin
    if i = n + 1 then Print(ℓ[1], ℓ[2], ..., ℓ[n]);
    else
        for ℓ[i] ← 0 to i − 1 do      // The range of condition (a.1) in Theorem 1.
            flag ← 1; j ← ℓ[i] + 1;
            while flag = 1 and j < i do
                // The testing of the condition (a.2) in Theorem 1.
                if ℓ[j] ≠ 0 and ℓ[j] ⩽ ℓ[i] then
                    flag ← 0;
                j ← j + 1;
            if flag = 1 then call Lex-Gen-Tree(i + 1);
```

Fig. 3. An procedure for generating LC-sequences in lexicographic order.

The above algorithm uses generate-and-test approach for filtering out all non-valid LC-sequences. Although the algorithm is correct, in general it is not efficient at all. For instance, we reveal some non-efficient evidences as follows. Suppose that $c = (c_1, c_2, \ldots, c_n)$ is a non-valid LC-sequence satisfying $c_j \neq 0$ and $c_j \leqslant c_i$ for some integers $i \in \{3, 4, \ldots, n\}$ and $c_i < j < i$. By Theorem 1, all subsequent sequences $(c_1, \ldots, c_{i-1}, c'_i, c_{i+1}, \ldots, c_n)$ for $c'_i \in \{c_i + 1, \ldots, i - 2\}$ are also non-valid. However, Lex-Gen-Tree does not detect this aspect and it performs the sequence generation and testing continuously.

A *rotation* is a simple operation that reconstructs a binary tree into another tree with the same number of nodes and preserves its inorder to be unchanged. In what follows, we design two new types of rotations for binary trees, where one is adjustable and the other is non-adjustable. Then, we present a more efficient algorithm, called Refined-Lex-Gen-Tree, that associates with these rotations for generating all LC-sequences in lexicographic order. Particularly, both rotations in a binary tree T are performed at node n (i.e., the parent of the rightmost leaf in T). Also, we imagine that T is the right subtree of a dummy node numbered by 0. The first one is called the *flip-on-site rotation*, denoted by FOS(k), which is an operation that flips the node n and its k immediate descendants in the left arm of the subtree T_n. Note that this operation is adjustable because the

number of all flipped nodes is dependent on k. In fact, the degenerate case of this operation when $k = 1$ is the usual right rotation (for AVL trees) at node n in T. See Fig. 4(a) for an illustration.

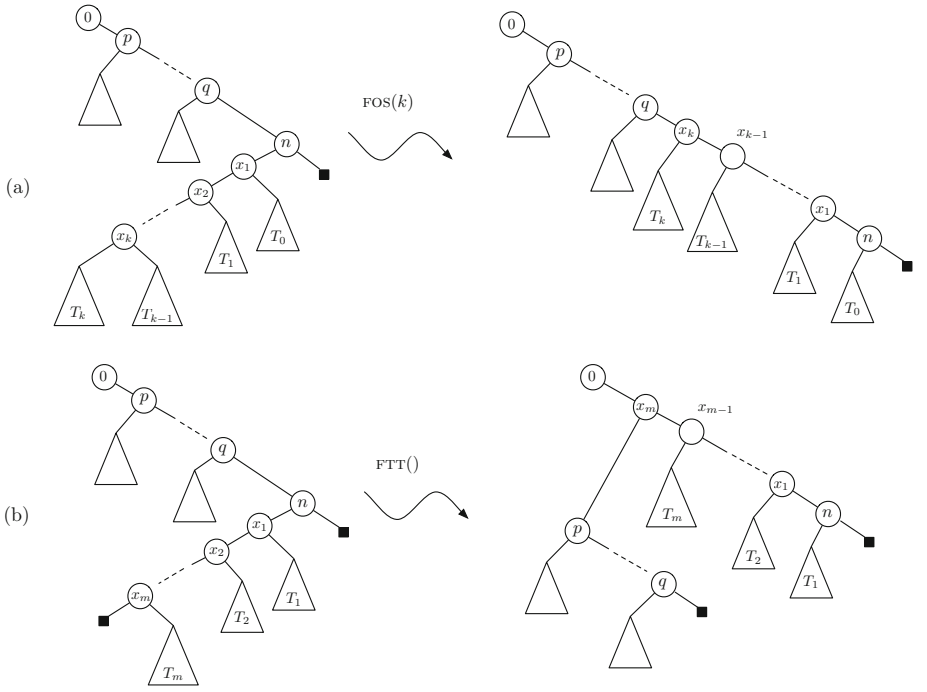

Fig. 4. (a) A flip-on-site rotation FOS(k); (b) A flip-to-top rotation FTT().

The second operation is called the *flip-to-top rotation*, denoted by FTT(), which is an operation that flips the node n and its all descendants in the left arm of T_n, and then moves this flipped list to the position between the root 0 and its right child. See Fig. 4(b) for an illustration. In general, the two types of rotations consist of a sequence of usual right rotations except for the last move of the flipped list in FTT().

Initially, the improved algorithm, called Refined-Lex-Gen-Tree, generates the first tree (i.e., the right-skewed tree with n internal nodes), and then repeatedly call a procedure Next-Tree() to generate all subsequent trees by using rotations defined above until a certain condition (i.e., CARRY $\leqslant n$, as explained later in the list of algorithm) is fulfilled. In each generation, a boolean variable FLIP_TO_TOP can determine which of FOS(k) and FTT() is the current operation. To preserve the inorder of binary trees to be unchanged after a rotation, we need the following three arrays $\ell[0..n]$, $r[0..n]$ and $p[0..n]$, where the first two are used for LC- and RC-sequences, and the last one is used for storing the parent information of

nodes. For each usual right rotation in the flipped list, the node where the rotation acts on is indicated by the variable "this", and its parent and left child are indicated by variables "prev" and "next", respectively. The detail of the refined algorithm is as shown in Fig. 5.

```
Algorithm  Refined-Lex-Gen-Tree
begin
    for i ← 0 to n do        // Generate the first tree.
        ℓ[i] ← 0;   r[i] ← i + 1;   p[i] ← i − 1;
    r[n] ← 0;
    Print(ℓ[1], ℓ[2], ..., ℓ[n]);      // Print the first tree sequence;
    CARRY ← 2;
    while CARRY ⩽ n do       // Generate the next tree sequence
        Next-Tree();
```

```
Procedure Next-Tree()
 1  begin
 2      this ← n;   prev ← p[this];   next ← ℓ[this];
 3      if ℓ[n] = 0 then FLIP_TO_TOP ← true;
 4      else if r[ℓ[n]] ≠ 0 then FLIP_TO_TOP ← false;
 5      else
 6          repeat                                   // Flip a sequence of nodes.
 7              r[next] ← this;   p[this] ← next;   ℓ[this] ← 0;
 8              p[next] ← 0;      this ← next;      next ← ℓ[this];
 9          until ℓ[this] ≠ this − 1;
10          if ℓ[this] = 0 then FLIP_TO_TOP ← true;
11          else FLIP_TO_TOP ← false;
12      if FLIP_TO_TOP then                          // Perform a flip-to-top rotation.
13          r[prev] ← 0;   ℓ[this] ← r[0];   p[r[0]] ← this;
14          r[0] ← this;   p[this] ← 0;
15      else                                         // Perform a usual right rotation.
16          ℓ[this] ← r[next];   p[r[next]] ← this;   r[next] ← this;
17          p[this] ← next;      r[prev] ← next;      p[next] ← prev;
18      Print(ℓ[1], ℓ[2], ..., ℓ[n]);      // Print the current tree sequence;
19      if ℓ[CARRY] = CARRY − 1 then CARRY ← CARRY + 1;
```

Fig. 5. A refined algorithm for generating LC-sequences in lexicographic order.

In the above algorithm, we use a global variable CARRY to control the generation in progress. Since the last tree in the generated list is the left-skewed tree satisfying $\ell[i] = i - 1$ for all $1 \leqslant i \leqslant n$, it is the errand of CARRY for completing this setting. Since $\ell[1] = 0$ is never changed, we set CARRY $= 2$ at the beginning. Once the setting in the current position is accomplished, CARRY is increased by one and goes ahead to the next position (see Line 19). The algorithm terminates when the condition CARRY $= n + 1$ is achieved.

In Next-Tree(), the variable FLIP_TO_TOP determines which type of rotations will be invoked. The decision is relied on the following rules:

(R1) if $\ell[n] = 0$, call FTT(); (Lines 13–14)
(R2) if $\ell[n] \neq 0$ and $r[\ell[n]] \neq 0$, call FOS(1); (Lines 16–17)
(R3) if $\ell[n] \neq 0$ and $r[\ell[n]] = 0$, flip a sequence of nodes by using usual right rotations until a rotation is performed at a node satisfying $\ell[\text{this}] \neq \text{this}-1$; (Lines 6–9)

Note that, for our Next-Tree() procedure, all flipped nodes triggered by the rule (R3) indeed have no right child because these nodes in the sequence meet with the condition $\ell[\text{this}] = \text{this} - 1$. After flipping the list of nodes in (R3), there are two statuses of the node where the last rotation acted on (i.e., the node indicated by the variable "this"). Accordingly, two kinds of subsequent processes are as follows:

(R4) if $\ell[\text{this}] = 0$, call FTT(); (Lines 13–14)
(R5) if $\ell[\text{this}] \neq 0$, call FOS(1); (Lines 16–17)

Actually, a flip-on-site rotation FOS(k) is an operation integrated with rules (R3) and (R5) for flipping the node n and partial descendants with $k-1$ nodes in the prefix segment of the left arm of T_n, as shown in Fig. 4(a). By contrast, a flip-to-top rotation FTT() is an operation integrated with rules (R3) and (R4) for flipping the node n and all descendants in the left arm of T_n, and then moving the flipped list to the top of the tree, as shown in Fig. 4(b). Obviously, FOS(k) operation requires $\mathcal{O}(k)$ time, and the complexity of FTT() operation is dependent on the length of the left arm of T_n. Since a usual right rotation requires only constant time, the complexity of each operation is indeed equal to the number of different digits between two consecutive LC-sequences, i.e., before operating and after operating.

We now at a position to show that the above rules can correctly generate the next sequence in lexicographic order by using the two types of rotations.

Lemma 1. *Let T be a binary tree and suppose that $\ell(T) = (\ell[1], \ell[2], \ldots, \ell[n])$ satisfies the condition of* (R1)*. If $\widetilde{T}$ is the binary tree obtained from T by taking a* FTT() *rotation at the node n, then $\ell(\widetilde{T})$ is the immediately succeeding sequence of $\ell(T)$ in lexicographic order.*

Proof. Let p be the right child of the root (i.e., the dummy node 0) in T. Since the node n has no left child, after performing FTT(), the whole sequence $\ell(T)$ keeps unchanged except the last position. In fact, the difference between $\ell(T)$ and $\ell(\widetilde{T})$ only occurs at $\ell[n]$, which is changed from 0 to p. We suppose to the contrary that the immediately succeeding sequence of $\ell(T)$ is $\ell(T') = (\ell[1], \ell[2], \ldots, \ell[n] = i)$ where $0 < i < p$ and T' is the corresponding binary tree. Since every number in an LC-sequence can appear at most once except the number 0, it implies that $\ell[j] \neq i$ for $1 \leqslant i \leqslant n-1$, and thus $\ell(T)$ does not contain i as an element. Moreover, since $i < p$, it follows that i must be the right child of some node in the subtree L_p of T. Let q be the least ancestor of i in T for which some node $k \in L_p$ takes q as its left child (i.e., $\ell[k] = q$). Clearly,

$q < i < k < p$. For T', since i is the left child of n, it implies $k \in R_i$ and $q \notin R_i$. However, this contradicts the fact that $\ell[k] = q$ remains unchanged in T' because $k \neq n$. □

Lemma 2. *Let T be a binary tree and suppose that $\ell(T) = (\ell[1], \ell[2], \ldots, \ell[n])$ satisfies the condition of* (R2)*. If $\widetilde{T}$ is the binary tree obtained from T by taking a usual right rotation at the node n, then $\ell(\widetilde{T})$ is the immediately succeeding sequence of $\ell(T)$ in lexicographic order.*

Proof. Suppose that x is the left child of n and let y be the right chile of x in T, i.e., $\ell[n] = x$ and $r[\ell[n]] = y$. Clearly, after performing FOS(1), the sequence $\ell(T)$ keeps unchanged except the last position $\ell[n]$, which is changed from x to y. Suppose to the contrary that the immediately succeeding sequence of $\ell(T)$ is $\ell(T') = (\ell[1], \ell[2], \ldots, \ell[n] = i)$ where $x < i < y$ and T' is the corresponding binary tree. Since $x < i < y$ and $\ell[j] \neq i$ for $1 \leqslant i \leqslant n-1$, i must be the right child of some node in the subtree L_y of T. Let q be the least ancestor of i in T for which some node $k \in L_y$ takes q as its left child (i.e., $\ell[k] = q$). Clearly, $q < i < k < y$. For T', since i is the left child of n, it implies $k \in R_i$ and $q \notin R_i$. However, this contradicts the fact that $\ell[k] = q$ remains unchanged in T' because $k \neq n$. □

Lemma 3. *Suppose that T and $\widetilde{T}$ are two binary trees generated by the procedure Next-Tree() such that $\widetilde{T}$ is obtained from T. Then, $\ell(\widetilde{T})$ is the immediately succeeding sequence of $\ell(T)$ in lexicographic order.*

Proof. By Lemmas 1 and 2, we have proved the correctness of generation produced by rotations without a flipped list of nodes, i.e., the status meets with the condition of (R1) or (R2). In general, to show the correctness when the current status meets with the condition of (R3), we may imagine that nodes in the flipped list are contracted to form a single node and it is indicated by the variable "this". This is due to the fact that every node in the flipped list contains no right child. As a result, the rule (R4) is in keeping with the rule (R1) if we treat the variable "this" as n. Also, we note that if the current status meets with the condition of (R5), it guarantees $r[\ell[\text{this}]] \neq 0$. Otherwise, we have $\ell[\text{this}] = \text{this} - 1$ and the loop (Lines 6–9) goes ahead to the next round. Similarly, if we treat the variable "this" as n, the rule (R5) is again in keeping with the rule (R2). Therefore, using arguments similar to Lemmas 1 and 2, we can prove the correctness if the status meets with the condition of (R4) or (R5), and thus the lemma follows. □

Theorem 2. *The algorithm* Refined-Lex-Gen-Tree *can correctly generates all LC-sequences of binary trees with n internal nodes in lexicographic order. In particular, each generation requires only constant amortized time with no more than* 2*.*

Proof. The correctness of the algorithm Refined-Lex-Gen-Tree directly follows from Lemma 3. We now give the complexity analysis as follows. Recall that we

use $\mathscr{T}_n$ to denote the set of binary trees with n internal nodes. Let EC_n be the expected cost of generating an LC-sequences of length n in Refined-Lex-Gen-Tree. In [10], Lucas et al. showed that several coding trees for representing binary tree sequences are isomorphic. Actually, a coding tree for representing a certain type of binary tree sequences may come from an old one by changing the sequence representation and rearranging sequence order. Thus, the number of nodes in each level of the coding tree does not be changed, i.e., a Catalan number $|\mathscr{T}_k|$ for $k \in \{1 \dots n\}$. Let N_k be the number of pairs of two consecutive LC-sequences of length n with k different digits in the lexicographic order. It is easy to observe that for each $k \in \{1 \dots n-1\}$,

$$N_k = |\mathscr{T}_{n-k+1}| - |\mathscr{T}_{n-k}|.$$

Since $|\mathscr{T}_1| = 1$, the total complexity of generating all LC-sequences of length n is

$$\begin{aligned}\sum_{k=1}^{n-1} kN_k &= \big(|\mathscr{T}_n| - |\mathscr{T}_{n-1}|\big) + 2\big(|\mathscr{T}_{n-1}| - |\mathscr{T}_{n-2}|\big) + 3\big(|\mathscr{T}_{n-2}| - |\mathscr{T}_{n-3}|\big) + \cdots \\ &\quad + (n-2)\big(|\mathscr{T}_3| - |\mathscr{T}_2|\big) + (n-1)\big(|\mathscr{T}_2| - |\mathscr{T}_1|\big) \\ &= S_n - n,\end{aligned}$$

where S_n denote the sum of the first n Catalan numbers [1]. In fact, it has been pointed out in [24] that $S_n < |\mathscr{T}_{n+1}|$. Thus, we have

$$EC_n = \frac{\sum_{k=1}^{n-1} kN_k}{|\mathscr{T}_n|} = \frac{|\mathscr{T}_n| + S_{n-1} - n}{|\mathscr{T}_n|} < \frac{2|\mathscr{T}_n| - n}{|\mathscr{T}_n|} < 2.$$

In particular, the expected cost $EC_n = \frac{4}{3}$ when n tends to infinite. This completes the proof. □

4 Concluding Remarks

In this paper, we propose a constant amortized-time algorithm for generating all LC-sequences of binary trees with n internal nodes in lexicographic order. It is especially interested that the proposed algorithm is associated with two new types of rotations called flip-on-site and flip-to-top. As we know that a rotation can be viewed as a transformation that changes the shape of a binary tree and usually preserves some desired property, such as keeping the inorder unchanged or adjusting to be a balanced tree, this leads to that tree transformation has many applications [5–8]. Thus, the design of efficient way for tree transformation is an important issue. However, up to now there are many discussions related to the usual rotations, and only a few attention has been focused on the design of massive rotations (e.g., see [21] as an instance). Since both flip-on-site and flip-to-top are massive rotations, we expect to find more applications that can be dealt with by these rotations in the near future.

Acknowledgments. This research was partially supported by MOST grants MOST 105-2221-E-131-027 (Kung-Jui Pai), 104-2221-E-141-002-MY3 (Jou-Ming Chang) and 104-2221-E-262-005 (Ro-Yu Wu) from the Ministry of Science and Technology, Taiwan.

References

1. Adamchuk, A.: A014138. The On-Line Encyclopedia of Integer Sequences (2006). http://oeis.org/A014138
2. Boyer, J.M.: Simple constant amortized time generation of fixed length numeric partitions. J. Algorithms **54**, 31–39 (2005)
3. Effler, S., Ruskey, F.: A CAT algorithm for generating permutations with a fixed number of inversions. Inform. Process. Lett. **86**, 107–112 (2003)
4. Ehrlich, G.: Loopless algorithms for generating permutations, combinations, and other combinatorial configurations. J. ACM **20**, 500–513 (1973)
5. Gibbons, A., Sant, P.: Rotation sequences and edge-colouring of binary tree pairs. Theor. Comput. Sci. **326**, 409–418 (2004)
6. Guibas, L., Hershberger, J., Suri, S.: Morphing simple polygons. Discret. Comput. Geometry **24**, 1–34 (2000)
7. Hershberger, J., Suri, S.: Morphing binary trees. In: Proceedings of the ACM-SIAM Sixth Annual Symposium on Discrete Algorithms (SODA), pp. 396–404 (1995)
8. Kensler, A.: Tree rotations for improving bounding volume hierarchies. In: IEEE Symposium on Interactive Ray Tracing, pp. 73–76. IEEE Computer Society, Washington (2008)
9. Knuth, D.E.: The Art of Computer Programming. Fascicle 4A - Generating All Trees, vol. 4. Addison-Wesley, Boston (2005)
10. Lucas, J.M., van Baronaigien, D.R., Ruskey, F.: On rotations and the generation of binary trees. J. Algorithms **15**, 343–366 (1993)
11. Mäkinen, E.: A survey on binary tree codings. Comput. J. **34**, 438–443 (1991)
12. Pai, K.-J., Wu, R.-Y., Chang, J.-M., Chang, S.-C.: Amortized efficiency of ranking and unranking left-child sequences in lexicographic order. In: Chan, T.H., Li, M., Wang, L. (eds.) COCOA 2016. LNCS, vol. 10043, pp. 505–518. Springer, Cham (2016). doi:10.1007/978-3-319-48749-6_37
13. Pallo, J.: Enumerating, ranking and unranking binary trees. Comput. J. **29**, 171–175 (1986)
14. Proskurowski, A., Ruskey, F.: Binary tree Gray codes. J. Algorithms **6**, 225–238 (1985)
15. van Baronaigien, D.R.: A loopless algorithm for generating binary tree sequences. Inform. Process. Lett. **39**, 189–194 (1991)
16. Ruskey, F., Hu, T.C.: Generating binary trees lexicographically. SIAM J. Comput. **6**, 745–758 (1977)
17. Savage, C.D.: A survey of combinatorial Gray codes. SIAM Rev. **39**, 605–629 (1997)
18. Sawada, J.: Generating bracelets in constant amortized time. SIAM Comput. **31**, 259–268 (2001)
19. Vajnovszki, V.: On the loopless generation of binary tree sequences. Inform. Process. Lett. **68**, 113–117 (1998)
20. Wu, R.-Y., Chang, J.-M., Chan, H.-C., Pai, K.-J.: A loopless algorithm for generating multiple binary tree sequences simultaneously. Theoret. Comput. Sci. **556**, 25–33 (2014)

21. Wu, R.-Y., Chang, J.-M., Wang, Y.-L.: A linear time algorithm for binary tree sequences transformation using left-arm and right-arm rotations. Theoret. Comput. Sci. **355**, 303–314 (2006)
22. Zaks, S.: Lexicographic generation of ordered trees. Theoret. Comput. Sci. **10**, 63–82 (1980)
23. Zaks, S., Richards, D.: Generating trees and other combinatorial objects lexicographically. SIAM J. Comput. **8**, 73–81 (1979)
24. Zumkeller, R.: A014138. The On-Line Encyclopedia of Integer Sequences (2010). http://oeis.org/A014138