A Linear Time Algorithm for Binary Tree Sequences Transformation Using Left-arm and Right-arm Rotations

Ro-Yu Wu[1,2], Jou–Ming Chang[3], Yue-Li Wang[1,4,*]

[1] Department of Information Management, National Taiwan University of Science and Technology, Taipei, Taiwan, ROC

[2] Department of Industrial Engineering Management, Lunghwa University of Science and Technology, Taoyuan, Taiwan, ROC

[3] Department of Information Management, National Taipei College of Business, Taipei, Taiwan, ROC

[4] Department of Computer Science and Information Engineering, National Chi Nan University, Nantou, Taiwan, ROC

Abstract

In this paper we consider a transformation on binary trees using new types of rotations. Each of the newly proposed rotations is permitted only at nodes on the left-arm or the right-arm of a tree. Consequently, we develop a linear time algorithm with at most $n-1$ rotations for converting weight sequences between any two binary trees. In particular, from an analysis of aggregate method for a sequence of rotations, each rotation of the proposed algorithm can be performed in a constant amortized time. Next, we show that a specific directed rooted tree called rotation tree can be constructed using one of the new type rotations. As a by-product, a naive algorithm for enumerating weight sequences of binary trees in lexicographic order can be implemented by traversing the rotation tree.

Keywords: Algorithms; Amortized analysis; Binary trees; Rotations; Tree transformation; Enumeration; Lexicographic order;

*All correspondence should be addressed to Professor Yue-Li Wang, Department of Information Management, National Taiwan University of Science and Technology, No. 43, Section 4, Kee-Lung Road, Taipei, Taiwan, Republic of China. (Phone: 886–2–27376768, Fax: 886–2–27376777, Email: ylwang@cs.ntust.edu.tw).

1. Introduction

Binary trees are a fundamental data structure in computer science and has been widely studied over the past 40 years [1, 2, 4, 36]. One of the most common operations for reconstructing binary trees is the use of rotations. The usual rotations in the past researches are the left/right rotations for balancing binary search trees [17]. Thus a sequence of rotations can be viewed as a transformation that changes a tree into another tree with the same number of nodes. Since tree transformation using rotations has application on edge-coloring of binary trees [12] and is closely related to the problem of morphing (i.e., continuously deforming) one simple polygon into another [11, 13, 14], many researches have focused on the design of efficient way for tree transformation, especially a transformation such that the number of utilized rotations coincides with a measure called *rotation distance* (i.e., the least number of rotations necessary to convert one tree into the other). Unfortunately, it remains an open question whether rotation distance can be computed in polynomial time if the usual rotations on binary trees are applied. Therefore, it seems natural to consider restricted rotations in order to obtain more simple transformation. On the other hand, one can also consider variant types of rotations (especially, massive rotations) in order to obtain more efficient transformation.

According to the fact that any restricted rotation distance is bounded below by the usual rota-

tion distance, many restricted rotations have been introduced to estimate the usual rotation distance efficiently. Bonnin and Pallo [6] restricted the usual rotations to a special case where rotations are allowed only at nodes with a leaf as its left subtree. They showed that the rotation distance between two binary trees in this case can be computed in quadratic time. Sundar [38] studied tree transformation when only a single direction of rotation, called right rotation, is permitted. More recently, Cleary [8] considered the case when the rotation is permitted only at two nodes, the root and the right child of the root. He proved that $12n$ rotations are sufficient to complete the transformation between any two binary trees when this restriction is adopted. This bound was improved to $4n-8$ by Cleary and Taback [9], and shown to be sharp. Pallo [28] generalized this case to the situation where rotations are restricted only at nodes on the right-arm of the tree. In addition, he established an efficient algorithm to compute this right-arm rotation distance. For more information about tree transformation, please refer to [10, 24, 37] for general concept, [20, 21, 24, 25, 26, 29] for restricted rotation operations, and [22, 23, 27, 33, 37] for the computation of usual rotation distance.

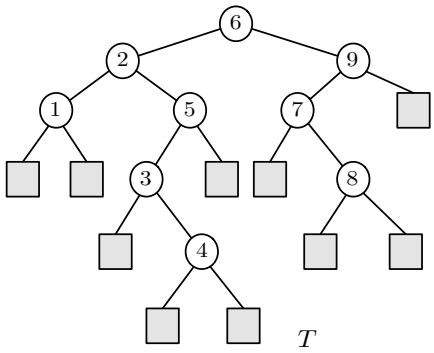
In contrast, if we provide a set of variant types of massive rotations that contains the usual rotations as a special case, then this type of rotation distance is bounded above by the usual rotation distance. However, up to now only a few attention has been given to the variant instances of massive rotations for tree transformation. In [7], Chen et al. generalized the usual rotations by considering the four types of rotations used in AVL trees. They proposed an efficient algorithm that constructs a sequence of rotations for estimating this type of rotation distance. In this paper, we introduce new type rotations which can rotate a massive part of a tree. All newly proposed rotations are unusual and differ from those of the past. We allow rotations to be performed at nodes on the right-arm (i.e., the path from the root to its rightmost leaf) or the left-arm (i.e., the path from the root to its leftmost leaf) of the tree. Consequently, we develop a simple linear time algorithm that uses no more than $n-1$ rotations to transform any two binary trees. The proposed algorithm takes the left weight sequence as input which will be defined in the next section. From an analysis of aggregate method for a sequence of weight renewals, we show that each rotation in the algorithm has a constant amortized time. Next, we show that the rotation graph with respect to one of the new type rotations is indeed a directed rooted tree, which will be called a rotation tree preferably. Then, we demonstrate that enumerating binary tree sequences in lexicographic order can be made by traversing the rotation tree.

2. Preliminaries

In this section we give some definitions and present preliminary results. An *extended binary tree* is a binary tree where each internal node (non-leaf) has exactly two children [17]. Further when we talk about binary trees, we mean extended binary trees. Let B_n denote the set of binary trees with n internal nodes (and thus with $n+1$ leaves) and $T \in B_n$. We assume that all internal nodes of T are numbered from 1 to n, according to the inorder traversal of T (i.e. visit recursively the left subtree, the root and then the right subtree of T). We shall not distinguish between a node and its number. For each internal node $i \in T$, the left subtree (respectively, right subtree) of i is the subtree of T rooted at the left child (respectively, right child) of i. Then the *left-weight* of i, denoted by $w_l(T,i)$, is the number of leaves in the left subtree of i. In [24], Pallo defined the integer sequence $w_l(T) = (w_l(T,1), w_l(T,2), \ldots, w_l(T,n))$ as the *left-weight sequence* (LW-sequence for short) of T and showed that every binary tree can be characterized as follows (see also reference [39]). An integer sequence $w = (w_1, w_2, \ldots, w_n)$ is the LW-sequence of a binary tree with n internal nodes if and only if for all $i \in [1,n]$ the following conditions are satisfied: (1) $1 \le w_i \le i$ and (2) $i - w_i \le i' - w_{i'}$ for all $i' \in [i-w_i+1, i]$. Obviously, the LW-sequence of T can be obtained from the inorder traversal of T in $\mathrm{O}(n)$ time.

In this paper the number of leaves in the right subtree of a node $i \in T$ is called the *right-weight* of i and is denoted by $w_r(T,i)$. Similarly, we refer the sequence $w_r(T) = (w_r(T,1), w_r(T,2), \ldots, w_r(T,n))$ as the *right-weight sequence* (RW-sequence for short) of T. It is easy to verify that every binary tree can also be characterized by its RW-sequence, i.e., an integer sequence $w = (w_1, w_2, \ldots, w_n)$ is the RW-sequence of a binary tree with n internal nodes if and only if for all $i \in [1,n]$ the following conditions are satisfied: (1) $1 \le w_i \le n-i+1$ and (2) $i + w_i \ge i' + w_{i'}$ for all $i' \in [i, i+w_i-1]$. Moreover, since the nodes of T are labeled by the inorder traversal, the number of leaves of node i in the left-arm (respectively, right-arm) is i (respectively, $n-i+1$). We write $P_L(T)$ as the left-arm consisting of nodes $\{i \in T : w_l(T,i) = i\}$, and $P_R(T)$ as the right-arm consisting of nodes $\{i \in T : w_r(T,i) = n-i+1\}$. For example, Figure 1 shows a tree $T \in B_9$ with $w_l(T) = (1,2,1,1,3,6,1,1,3)$ and $w_r(T) = (1,4,2,1,1,4,2,1,1)$.

Let T_L and T_R be the left subtree and right subtree of the root in a binary tree T, respectively. The *mirror image* $m(T)$ of T is recursively defined as follows: $m(T)_L = m(T_R)$, $m(T)_R = m(T_L)$, and $m(\square) = \square$, where $\square$ de-



$w_l(T) = (1, 2, 1, 1, 3, 6, 1, 1, 3)$

$w_r(T) = (1, 4, 2, 1, 1, 4, 2, 1, 1)$

Figure 1: The LW-sequence and the RW-sequence of a tree T.

notes a null tree. For a weight sequence w, let $m(w)$ be the reverse sequence of w. Then we have $w_r(T) = m(w_l(m(T)))$. For example in Figure 1, $w_l(m(T)) = (1, 1, 2, 4, 1, 1, 2, 4, 1)$ and $m(1, 1, 2, 4, 1, 1, 2, 4, 1) = (1, 4, 2, 1, 1, 4, 2, 1, 1) = w_r(T)$. Using the technique of mirror image, we can obtain a right weight sequence of a tree T in $O(n)$ time. In the rest of this section, we will provide a non-recursive procedure for obtaining the RW-sequence of T if the LW-sequence of T is given.

For a node $i \in T$, if $w_l(T, i) = 1$ (respectively, $w_r(T, i) = 1$), then we say that i has the *uni-left-weight* (respectively, *uni-right-weight*) in T. Also, we denote $L_1(T) = \{i \in T : w_l(T, i) = 1\}$ and $R_1(T) = \{i \in T : w_r(T, i) = 1\}$.

Lemma 1 *Let $T \in B_n$ be a binary tree. Then, $|L_1(T)| \geq \lfloor \frac{n}{2} \rfloor + 1$ or $|R_1(T)| \geq \lfloor \frac{n}{2} \rfloor + 1$.*

Proof. Since T contains $n+1$ leaves and each leaf is either the left child or the right child of a node, we have $|L_1(T)| + |R_1(T)| = n+1$. Thus, the result follows directly. □

A *rotation* is a simple operation for reconstructing a binary tree into another tree and preserving their inorder. Usually, we design such an operation to be performed in a constant time (e.g., four primitive types of rotations for balancing AVL-trees [1]). In this paper, we introduce new type rotations that each of them can be performed only at nodes on the left-arm or the right-arm of a tree. In order to detect whether a node is on the left-arm or the right-arm of some subtree, we need to provide the left-weight and the right-weight of nodes. In particular, r is the root of T if and only if $w_l(T, r) + w_r(T, r) = n + 1$. For convenience, we write $p_T(i)$ for the parent of a node $i \in T$. The following lemma shows that if both weight sequences of a tree are given, we can determine whether a specific node is contained in the left-arm or the right-arm of a subtree.

Lemma 2 *Let $T_L(i)$ and $T_R(i)$ be the left subtree and the right subtree of a node $i \in T$, respectively. If x is a descendant of i, then the following statements are true.*

(1) *x is contained in the left-arm of $T_R(i)$ if and only if $x - w_l(T, x) = i$;*

(2) *x is contained in the right-arm of $T_L(i)$ if and only if $x + w_r(T, x) = i$.*

Proof. (1) For the "if part", it is obviously that if $x \in T_L(i)$ then $x < i$. Moreover, if $x \in T_R(i)$ but it is not contained in the left-arm of $T_R(i)$ then $x - i > w_l(T, x)$. Conversely, if x is on the left-arm of $T_R(i)$, the number of internal nodes in the left subtree of x is $(x - 1) - i$. Thus, the number of leaves in the left subtree of x (i.e., $w_l(T, x)$) is $x - i$, which gives the desired result. (2) The proof is similar to case (1). □

Corollary 3 *For each node $x \in T$ which is not the root, $p_T(x)$ can be computed as follows:*

(1) *If x is a right child of a node, then $p_T(x) = x - w_l(T, x)$;*

(2) *If x is a left child of a node, then $p_T(x) = x + w_r(T, x)$.*

Proof. (1) It directly follows from Lemma 2 by considering that x is the root (and thus on the left-arm) of the right subtree of $p_T(x)$. (2) The proof is similar to case (1). □

Thus, we can compute $p_T(i)$ of a node $i \in T$ in a constant time if we already know that i is a right child or a left child. Also, an easy observation shows that if i is a right child, then $w_r(T, p_T(i)) = w_l(T, i) + w_r(T, i)$. Therefore, using the result of Corollary 3, we can compute the RW-sequence of a binary tree $T \in B_n$ easily if its LW-sequence is available.

Algorithm LW-SEQUENCE-TO-RW-SEQUENCE

```
for i = 1 to n do
    w_r(T, i) ← 0;
enddo
for i = n downto 1 do
    if (w_r(T, i) = 0) then
        w_r(T, i) ← 1;
    endif
    p ← i − w_l(T, i);
    if (p > 0 and w_r(T, p) = 0) then
```

$w_r(T,p) \leftarrow w_l(T,i) + w_r(T,i)$;

endif

enddo

Similarly, if the RW-sequence of a binary tree $T \in B_n$ is provided, we can design an analogous algorithm to compute the LW-sequence of T in linear time.

3. Left-arm and Right-arm Rotations

In what follows, we give the formal definition of rotations on the left-arm and/or the right-arm of a tree T (See Figure 2 for visual illustrations):

(a) The *left-arm left-rotation* (LL-rotation for short) at a node $i \in P_L(T)$ with $w_r(T,i) \neq 1$: At first, we assume that k is the right child of i and let j be the node with the smallest (inorder) number in the right subtree of i (i.e., $j = i+1$). It is possible that k and j are the same node. Also, if i is not the root of T, let $p = p_T(i)$. After applying this operation, p becomes the new parent of k if it exists, i becomes the new left child of j, the right child of i is replaced by a leaf, and all the rest in the tree remain unchanged. Note that all the right-weights of nodes are preserved under this rotation except the change $w_r(T,i) = 1$. Moreover, for each node x in the path from j to k, the left-weight $w_l(T,x)$ is augmented to $w_l(T,x) + w_l(T,i)$.

(b) The *left-arm right-rotation* (LR-rotation for short) at a node $i \in P_L(T)$ with $w_r(T,i) = 1$: This operation requires an extra parameter w for indicating which part of the tree T will be converted into the right subtree of i. We perform this rotation only in the case that the node $p = i + w$ is located in the left-arm (i.e., p is an ancestor of i). Let $j = p_T(i)$ and k be the left child of p. It is possible that k and j are the same node. If we complete this rotation, p becomes the new parent of i, k becomes the new right child of i, the left child of j is replaced by a leaf, and the remaining part of the tree is unchanged. Note that all the right-weights of nodes are unaltered except the change $w_r(T,i) = w$ after rotation. Moreover, for each node x in the path from j to k, the left-weight $w_l(T,x)$ is reduced to $w_l(T,x) - w_l(T,i)$.

(c) The *right-arm right-rotation* (RR-rotation for short) at a node $i \in P_R(T)$ with $w_l(T,i) \neq 1$: This operation is a symmetric case to the LL-rotation by exchanging "left" with "right" in all situations. We substitute the nodes g and h for the nodes k and j, respectively. So $h = i-1$ is the node with the largest number in the left subtree of i before rotation. When the rotation is terminated, all related positions of nodes are shown in Figure 2(b).

(d) The *right-arm left-rotation* (RL-rotation for short) at a node $i \in P_R(T)$ with $w_l(T,i) = 1$: This operation is the mirror image of the LR-rotation. Let w denote the weight that will be converted into the left subtree of i. We perform this rotation only in the case that the node $p = i - w$ is located in the right-arm, where w is an extra parameter as stated in the definition of LR-rotation. All related positions of nodes before operation and the adjustment after operation are shown in Figure 2(b).

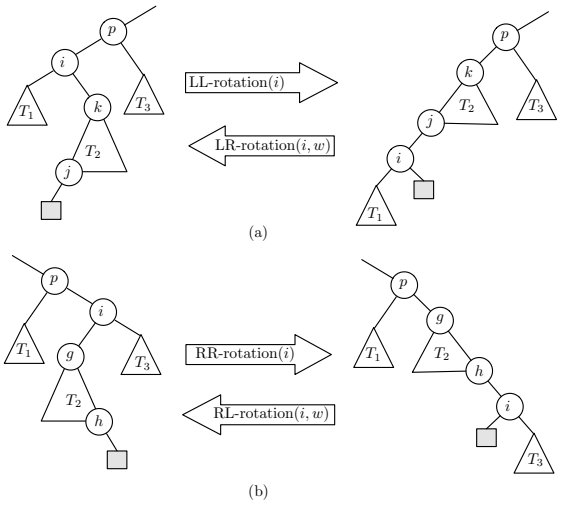


Figure 2: The left-arm and right-arm rotations.

According to the above definition, every single rotation requires updating three pointers in the tree. However, under the weight representation of a tree, it only needs to maintain the left-weights and the right-weights of nodes. We now implement the corresponding functions as follows (where T represents the current tree before rotation):

Function LL-rotation(i)

$w_r(T,i) \leftarrow 1$;

for each node x in the path from $j = i+1$ up to k **do**

$w_l(T,x) \leftarrow w_l(T,x) + w_l(T,i)$;

enddo

end LL-rotation

Function LR-rotation(i, w)
  $p \leftarrow i + w$;
  if $(p \in P_L(T))$ **then**
    $w_r(T, i) \leftarrow w$
    for each node x in the path from
    $j = p(i)$ up to k **do**
      $w_l(T, x) \leftarrow w_l(T, x) - w_l(T, i)$;
    enddo
  endif
end LR-rotation

Function RR-rotation(i)
  $w_l(T, i) \leftarrow 1$;
  for each node x in the path from
  $h = i - 1$ up to g **do**
    $w_r(T, x) \leftarrow w_r(T, x) + w_r(T, i)$;
  enddo
end RR-rotation

Function RL-rotation(i, w)
  $p \leftarrow i - w$;
  if $(p \in P_R(T))$ **then**
    $w_l(T, i) \leftarrow w$
    for each node x in the path from
    $h = p(i)$ up to g **do**
      $w_r(T, x) \leftarrow w_r(T, x) - w_r(T, i)$;
    enddo
  endif
end RL-rotation

For each function described above, by Corollary 3 we are easy to trace a path from a given node up to its ancestors for maintaining a sequence of weights. To determine which is the last node in this path, we can examine the condition of Lemma 2 for LL-rotations and RR-rotations (e.g., for an LL-rotation, every node contained in the path from j to k is on the left-arm of the right subtree of i). Oppositely, for LR-rotations and RL-rotations, the node p has been recognized before weight renewal. Indeed, we guarantee $p \in P_L(T)$ for LR-rotations and $p \in P_R(T)$ for RL-rotations when these functions are invoked by the main algorithm which will be introduced in the next section. Thus each of these two rotations can process the renewal along the path until p is arrived. Obviously, rotations of these types take $O(n)$ time. Interestingly, in the next section we will show that using the aggregate method of amortized analysis for a sequence of n rotations takes the worst case time $O(n)$ in total for maintaining the weight sequences of nodes. Thus, each rotation can be run in a constant amortized time.

4. An Algorithm of Tree Transformation

In this section, we describe our algorithm for converting T into T', where $T, T' \in B_n$. T is called the source tree and T' is the destination tree. Just the same as the input of algorithms in [24, 25, 28], we assume that both T and T' are given by their LW-sequences. Our algorithm has two phases, where the first phase converts the source tree into a skew tree, and the second phase converts the skew tree into the destination tree. Note that a *left-skew tree* (respectively, *right-skew tree*) is a skew tree in which every node in the tree has a uni-right-weight (respectively, uni-left-weight). Based on the scheme, we need to decide that which of the left-skew tree or the right-skew tree will be treated as an intermediate tree in the algorithm. Let $L_1(T)$ and $R_1(T)$ (respectively, $L_1(T')$ and $R_1(T')$) denote the set of nodes in T (respectively, T') with the uni-left-weight and the uni-right-weight, respectively. An essential resolution of the above criterion is that we choose a converting path passing through a skew tree such that it has a shorter length. The selection depends on the measure of the difference between $|L_1(T)| + |L_1(T')|$ and $|R_1(T)| + |R_1(T')|$. If $|L_1(T)| + |L_1(T')| \leq |R_1(T)| + |R_1(T')|$, then the left-skew tree is chosen as an intermediate tree. In this case, LL-rotations and LR-rotations will be used in the transformation. On the other hand (i.e., $|L_1(T)| + |L_1(T')| > |R_1(T)| + |R_1(T')|$), the right-skew tree is chosen as an intermediate tree and only RR-rotations and RL-rotations are used in the transformation. The following is our algorithm which takes $w_l(T)$ and $w_l(T')$ as the input.

Algorithm Tree-Conversion

---

1. Compute the RW-sequences $w_r(T)$ and $w_r(T')$;
2. **if** $(|L_1(T)| + |L_1(T')| \leq |R_1(T)| + |R_1(T')|)$ **then**
2.1. **for** $i = 1$ to n **do**
  if $(i \in P_L(T)$ and $w_r(T, i) \neq 1)$ **then**
    Perform LL-rotation(i);
  endif
enddo
2.2. **for** $i = n$ downto 1 **do**
  if $(w_r(T', i) \neq 1)$ **then**
    Perform LR-rotation$(i, w_r(T', i))$;
  endif
enddo
3. **else**
3.1. **for** $i = n$ downto 1 **do**
  if $(i \in P_R(T)$ and $w_l(T, i) \neq 1)$ **then**

```
            Perform RR-rotation(i);
        endif
    enddo
3.2. for i = 1 to n do
        if (w_l(T', i) ≠ 1) then
            Perform RL-rotation(i, w_l(T', i));
        endif
    enddo
endif
```

We now give an example to illustrate Algorithm Tree-Conversion. Figure 3(a) shows a source tree T with $|L_1(T)| = 3$ and $|R_1(T)| = 4$, and a destination tree T' with $|L_1(T')| = 3$ and $|R_1(T')| = 4$. According to the criterion in Step 2, the algorithm uses LL-rotations and LR-rotations for tree transformation. The detail of converting steps is shown in Figure 3(b). In Step 2.1, the right weights of nodes 2 and 4 are not 1. Therefore, an LL-rotation is performed at each of these two nodes in that order. After that, a left-skew tree is obtained. In Step 2.2, since only nodes 4 and 2 have $w_r(T, 4) \neq 1$ and $w_r(T, 2) \neq 1$, respectively, one LR-rotation is performed for each of these two nodes in that order. Moreover, since $w_r(T', 4) = 2$ (respectively, $w_r(T', 2) = 4$), the LR-rotation performed on node 4 (respectively, 2) is LR(4,2) (respectively, LR(2,4)). After stpes 2.1 and 2.2, a source tree T has been transformed to a destination tree T'.

Lemma 4 *The sequence of rotations performed in Algorithm* Tree-Conversion *has length no more than* $n - 1$.

Proof. By Lemma 1, we have $|L_1(T)| \geq \lfloor \frac{n}{2} \rfloor + 1$ or $|R_1(T)| \geq \lfloor \frac{n}{2} \rfloor + 1$ and $|L_1(T')| \geq \lfloor \frac{n}{2} \rfloor + 1$ or $|R_1(T')| \geq \lfloor \frac{n}{2} \rfloor + 1$. With the fact $|L_1(T)| + |L_1(T')| = |R_1(T)| + |R_1(T')| = n + 1$, this implies that $\max\{|L_1(T)| + |L_1(T')|, |R_1(T)| + |R_1(T')|\} \geq n + 1$. Recall that the tree transformation of Algorithm Tree-Conversion is designed to have two phases. We assume that the condition $|L_1(T)| + |L_1(T')| \leq |R_1(T)| + |R_1(T')|$ fulfills and hence we prove that Step 2.1 (the first phase) and Step 2.2 (the second phase) use at most $n-1$ rotations. For the other case (i.e., $|L_1(T)| + |L_1(T')| > |R_1(T)| + |R_1(T')|$), it can be proved by a similar way.

The first phase uses LL-rotations to transform T into a left-skew tree which has a uni-right-weight in every node. Since the number of nodes with uni-right-weight is adjusted by adding 1 for every LL-rotation, there are exactly $n-|R_1(T)|$ LL-rotations to be performed in this phase. Contrastively, since we need to perform LR-rotations only at nodes i with $w_r(T', i) \neq 1$, there are exactly $n - |R_1(T')|$ LR-rotations to be performed in the second phase. Thus, we totally use $2n - (|R_1(T)| + |R_1(T')|) \leq 2n-(n+1) = n-1$ rotations and the lemma follows. □

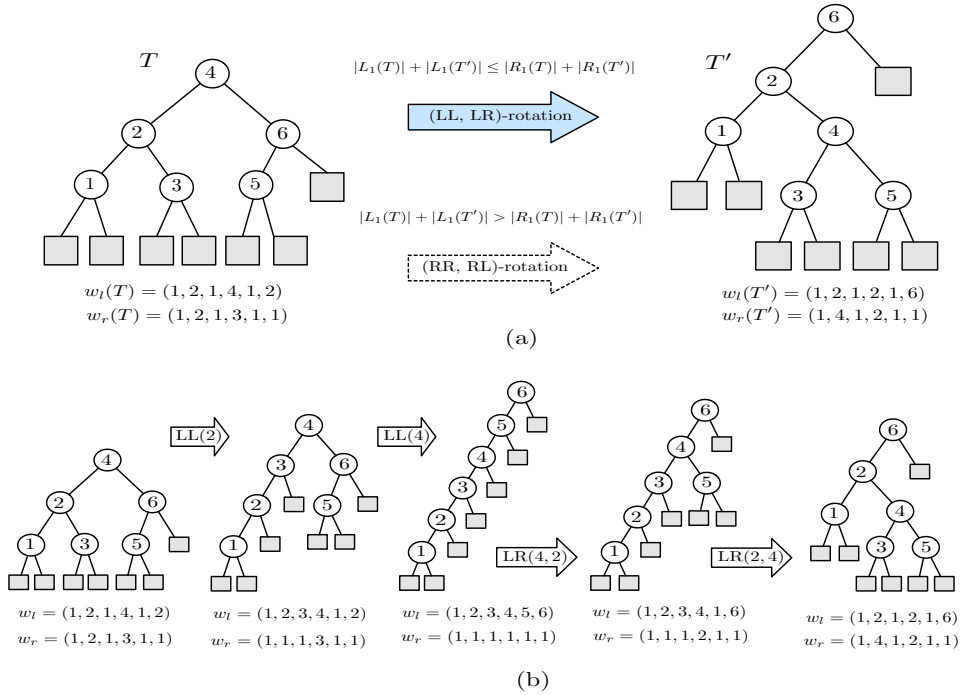


Figure 3: An example of tree transformation.

Theorem 5 *Given the left-weight sequences of two binary trees $T, T' \in B_n$, Algorithm* Tree-Conversion *correctly produces a sequence of rotations to convert T into T' in $O(n)$ time. In particular, every rotation in the algorithm takes a constant amortized time.*

Proof. First of all, both the RW-sequences of T and T' can be obtained by using Algorithm LW-sequence-to-RW-sequence in $O(n)$ time. From the symmetry, we only prove that Step 2 can correctly convert T into T'. Certainly, Step 2.1 can convert T into a left-skew tree using a sequence of LL-rotations. We consider Step 2.2 as follows. For each LR-rotation at node i, let T_c and T'_c be the current tree before rotation and after rotation, respectively. Since i is located on the left-arm of T_c and T'_c, we have $w_l(T_c, i) = w_l(T'_c, i) = i$. Moreover, the right-weight of i is changed from 1 to $w = w_r(T', i)$ and never updated again. Thus, $w_r(T'_c, i) = w_r(T', i)$. Let $p = i + w$ be a node in T_c and $p(i)$ be the parent of i in T'_c. Then, $p = w_l(T_c, i) + w_r(T', i) = w_l(T'_c, i) + w_r(T'_c, i) = w_l(T'_c, p(i)) = p(i)$. These equalities hold because i is the left child of $p(i)$ and $p(i)$ is located on the left-arm of T'_c. Therefore, p is also contained in the left-arm of T_c. This shows that the parameter $w = w_r(T', i)$ in the LR-rotation can correctly convert a part of the tree into the right subtree of i. As a result, the destination tree T' can be derived from a sequence of LR-rotations. Since every rotation accurately maintains the LW-sequence and the RW-sequence of the current tree, the correctness of the algorithm can be achieved.

To show that the time complexity of Algorithm Tree-Conversion is linear, by Lemma 4, we need to prove that the entire sequence of no more than $n - 1$ rotations in each of Step 2 or Step 3 takes at most $O(n)$ time even if a single rotation might be expensive. Indeed, we want to show that in the worst case the sequence of $n - 1$ rotations for each type requires updating weights at most $O(n)$ times. Again by the symmetry, we omit the case of Step 3. Since an LR-rotation can be viewed as a reverse function of an LL-rotation, let us merely analyze a sequence of LL-rotations in Step 2.1. According to the definition, we have known that every LL-rotation has exactly a single right-weight renewal, so the analysis is inclined to attain the aggregation of the number of left-weight renewals. Since the sequence of LL-rotations is performed at nodes in increasing order (from 1 to n) to reconstruct a left-skew tree, if an LL-rotation at node i is carried out then the left-weight of nodes j with $j \leq i$ has never been changed again. Furthermore, the change of the left-weight for a node x can occur only in the case that x is moved from the left-arm of the right subtree of a node i to the left-arm of the current tree (see Figure 2(a)). By the fact that every node can be moved to the left-arm of the left-skew tree at most once, the sequence of rotations takes a total of $O(n)$ time for weight renewals. Thus, the average cost of a rotation is $O(1)$. Usually, we assign the amortized cost of each operation to be the average cost in an aggregate analysis. Therefore, each type of rotations in the algorithm has a constant amortized time. □

5. Enumeration of Binary Tree Sequences

Many algorithms have been published for generating all binary trees with n nodes. In most of the algorithms, the trees are encoded as integer sequences and all such sequences are enumerated by lexicographic order. See, for example, the codeword representation [18, 41], the weight sequence [24, 39], the bitstring [3, 30, 35, 40], the distance representation [23], and the tree permutation [16, 34]. In [19], Lucas et al. showed that there exist strong relationships among various representations of binary trees. In this section, we will construct a directed rooted tree $\mathbb{T}_n$ using RL-rotation defined in the previous section such that every node of $\mathbb{T}_n$ corresponds to the LW-sequence of a binary tree with n nodes. Consequently, a naive algorithm for enumerating all binary tree sequences can be implemented by traversing $\mathbb{T}_n$.

The *rotation graph* $\mathbb{G}_n$ is a digraph with vertex set consisting of all binary trees of B_n, and two vertices are connected by an arc if there is a single rotation that converts one tree into the other. For convenience, we use LW-sequences instead of binary trees to represent the nodes of $\mathbb{G}_n$. Using the left-arm and right-arm rotations defined in Section 3, the rotation graph with respect to B_n is determined uniquely. In particular, if we restrictedly use only RL-rotations, the resulting digraph is definitely a directed rooted tree with $(1, 1, \ldots, 1)$ (i.e., the right-skew tree) as its root.

Lemma 6 *The rotation graph $\mathbb{G}_n$ with respect to RL-rotations is a directed rooted tree.*

Proof. For any RL-rotation(i, w) performed in a tree T with n nodes, node i is contained in the right-arm of T by definition. Since every profitable rotation requires $w \geq 2$ and all the left-weights of nodes in T are unaltered except that $w_l(T, i)$ is changed from 1 to w after rotation, the LW-sequence of T will be converted into a sequence with largely lexicographic order. This shows that the resulting digraph is acyclic. In particular, the node $(1, 1, \ldots, 1)$

(i.e., the sequence consisting of n 1's) has no ingoing arc. Since we have shown that the second phase in Algorithm TREE-CONVERSION can converts the right-skew tree into destination tree via RL-rotations, it guarantees that any other node of $\mathbb{G}_n$ is reachable from $(1, 1, \ldots, 1)$. Thus $\mathbb{G}_n$ is a directed rooted tree with node $(1, 1, \ldots, 1)$ as its root. □

From the above lemma, the rotation graph with respect to RL-rotations will be called a *rotation tree* and is denoted by $\mathbb{T}_n$ preferably. For example, Figure 4 shows a rotation tree $\mathbb{T}_4$, where the node $(1, 1, 1, 1)$ can be converted into $(1, 1, 2, 1)$ via RL(3,2) rotation. Again, the node $(1, 1, 2, 1)$ can be converted into $(1, 1, 2, 4)$ via RL(4,4) rotation. Obviously, the rotation tree is an unordered tree. Also, an easy observation shows that if the nodes of $\mathbb{T}_n$ are permuted such a way that nodes from left to right in each level are labeled in lexicographic order, then so is the printout in the preorder traversal of $\mathbb{T}_n$. Thus, it is easy to develop an algorithm for enumerating LW-sequences of binary trees in lexicographic order by constructing the rotation tree with a specific order and then traversing it. We now give a recursive algorithm to construct $\mathbb{T}_n$ with a specific order starting from the root $(1, 1, \ldots, 1)$ as its parameter as follows:

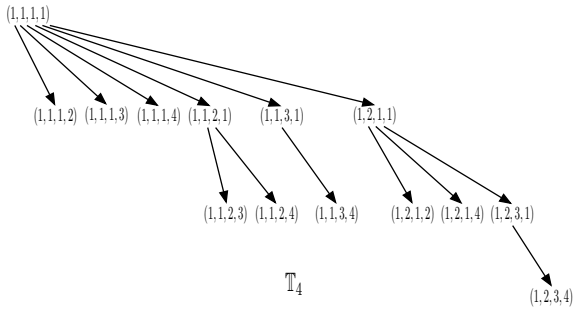


Figure 4: The rotation tree $\mathbb{T}_4$ whose nodes in each level are labeled in lexicographic order.

Procedure LexGenTree(T)
  for $i = n$ downto 2 **do**
    if ($w_l(T, i) = 1$ and $i \in P_R(T)$) **then**
      $T' \leftarrow T$;
      repeat
        $w \leftarrow w_l(T', i) + w_l(T', i - w_l(T', i))$;
        Create a new tree T' that is the same as T;
        Perform RL-rotation(i, w) in T';
        Insert the node T' as a child of T in the rotation tree $\mathbb{T}_n$;
        if ($w_l(T', n) = 1$) **then**
          LexGenTree(T');
        endif
      until ($w_l(T', i) = i$)
    endif
  enddo
end LexGenTree

For each recursive call LexGenTree(T), it generates all children of T in lexicographic order, where each child T' can be obtained from T by performing a single RL-rotation at a node i with $w_l(T, i) = 1$ on the right arm of T. In fact, to implement the above procedure, a standard representation of general trees is required. By applying the left-child-right-sibling structure (i.e., a structure dealt directly with the pointer representation of the binary tree) to $\mathbb{T}_n$, the variable T (respectively, T') in the above procedure signifies a pointer to a node consisting of three fields: a pointer to its first child, a pointer to its next sibling, and an integer sequence for representing LW-sequence of T. Thus, if a node which is not the root of $\mathbb{T}_n$ is the first created node when procedure LexGenTree is invoked, then this node will be the child of the previous created node; otherwise the node will be the sibling of the previous created node. Therefore, the resulting rotation tree is an ordered tree whose nodes in each level appear in lexicographic order. Also, notice that a preorder traversal on binary tree with the left-child-right-sibling structure can produce the same order information as that in the general tree representation. Although several interesting algorithms for generating binary trees sequence have been presented in the literature, however, the enumeration founded on tree traversal is conceptually simple. Thus, if the rotation tree $\mathbb{T}_n$ has been constructed, the running time of traversal algorithm for enumerating binary tree sequences is proportional to the number of binary trees in B_n.

6. Conclusion

In this paper, we define new types of rotations for tree transformation. These rotations can be performed only at nodes on the left-arm or the right-arm of a tree. Consequently, we develop a simple linear time algorithm to transform weight sequences between any two binary trees. The analysis of time complexity of rotations is especially interesting when both the left-weights and the right-weights of nodes are adopted. As it is, each rotation of the algorithm can be performed in a constant amortized time. As we have mentioned before, the rotation distance from a source tree T to a destina-

tion tree T', denoted by $dist(T, T')$, is the smallest number of rotations necessary to convert T into T'. From the proposed algorithm, we obtain an upper bound of $n-1$ on the new type rotation distance between any two binary trees $T, T' \in B_n$. An extreme instance which realizes this bound is shown in the shapes that T is a left-skew-tree and T' is a right-skew-tree, vice versa. Thus the bound $n-1$ is tight.

In succession, we show that a rotation tree $\mathbb{T}_n$ can be constructed by restricting the use of RL-rotations only. Then, we demonstrate that an algorithm for enumerating binary trees sequences in lexicographic order can be implemented by traversing the rotation tree. Recall that the number of binary trees in B_n is given by the Catalan number $\frac{1}{n+1}\binom{2n}{n}$. Thus $\mathbb{G}_n$ is exponentially large with respect to n. Many researches have studied the structure properties of rotation graphs. See, for example, Pallo [24, 25, 26], Lucas [18, 19], and Sleator, Tarjan, and Thurston [37]. Further results related to rotation graphs can also refer to [5, 15, 31, 32]. We close this paper with the following comparison about various types of rotation graphs, where $D(\mathbb{G}_n)$ denote the diameter of $\mathbb{G}_n$ (i.e., the maximum rotation distance over all pairs of vertices), and $\mu(\mathbb{G}_n)$ the average rotation distance of $\mathbb{G}_n$ which is defined as follows:

$$\begin{aligned}\mu(\mathbb{G}_n) &= \frac{\sum_{T,T' \in B_n, T \neq T'} dist(T, T')}{2 \cdot \binom{|B_n|}{2}} \\ &= \frac{\sum_{T,T' \in B_n, T \neq T'} dist(T, T')}{|B_n| \cdot (|B_n| - 1)}.\end{aligned}$$

$D(\mathbb{G}_n)$		$n=3$ $\lvert B_n\rvert = 5$	$n=4$ $\lvert B_n\rvert = 14$	$n=5$ $\lvert B_n\rvert = 42$
Restricted Rotation	[8]	4	8	12
	[28]	4	6	8
Usual Rotation		2	4	5
Variant Rotation	[7]	2	4	5
	*	2	3	4

$\mu(\mathbb{G}_n)$		$n=3$ $\lvert B_n\rvert = 5$	$n=4$ $\lvert B_n\rvert = 14$	$n=5$ $\lvert B_n\rvert = 42$
Restricted Rotation	[8]	2	3.65	5.77
	[28]	2	3.25	4.71
Usual Rotation		1.5	2.19	3.02
Variant Rotation	[7]	1.4	1.96	2.63
	*	1.4	2.03	2.79

Table 1: The diameter and the average rotation distance for various types of rotation graph $\mathbb{G}_n$ with $n = 3, 4$, and 5. (* indicates the result of this paper)

References

[1] G. M. Adel'son-Vel'skiĭ and E. M. Landis, An algorithm for organization of information, *Soviet Mathematics Doklady* 3 (1962) 1259–1263.

[2] A. Andersson, General balanced trees, *Journal of Algorithms* 30 (1999) 1–18.

[3] V. Bapiraju and V.V.B. Rao, Enumeration of binary trees *Information Processing Letters* 51 (1994) 125-127.

[4] R. Bayer, Symmetric binary B-trees: data structure and maintenance algorithms, *Acta Informatica* 1 (1972) 290–306.

[5] M. K. Bennet and G. Birkhoff, Two families of Newman lattices, *Algebra Universalis* 32 (1994) 115–144.

[6] A. Bonnin and J. Pallo, A shortest path metric on unlabeled binary Trees, *Pattern Recognition Letters* 13 (1992) 411–415.

[7] Yen-Ju Chen, Jou-Ming Chang, and Yue-Li Wang, An efficient algorithm for estimating rotation distance between two binary trees, to appear in *International Journal of Computer Mathematics*.

[8] S. Cleary, Restricted rotation distance between binary trees, *Information Processing Letters* 84 (2002) 333–338.

[9] S. Cleary and J. Taback, Bounding restricted rotation distance, *Information Processing Letters* 88 (2003) 251–256.

[10] K. Culik and D. Wood, A note on some tree similarity measures, *Information Processing Letters* 15 (1982) 39–42.

[11] B. Effantin, Generation of valid labelled binary trees, *Proc. International Conference on Computational Science and Its Applications* (ICCSA'2003), LNCS, vol. 2667 (2003) 245–253.

[12] A. Gibbons and P. Sant, Rotation sequences and edge-colouring of binary tree pairs, *Theoretical Computer Science* 326 (2004) 409–418.

[13] L. Guibas and J. Hershberger, Morphing simple polygons, *Proc. ACM 10th Annual Symposium of Computational Geometry* (SCG'94), 1994, 267-276.

[14] J. Hershberger and S. Suri, Morphing binary trees, *Proc. ACM-SIAM 6th Annual Symposium of Discrete Algorithms* (SODA'95), 1995, 396-404.

[15] F. Hurtado and M. Noy, Graph of triangulations of a convex polygon and tree of triangulations, *Computational Geometry* 13 (1999) 179–188.

[16] G. D. Knott, A numbering system for binary trees, *Communications of ACM* 20 (2), (1977), 113-115.

[17] D. E. Knuth, Sorting and Searching, in: *The Art of Computer Programming*, Vol. 3, Addison-Wesley, Reading, MA, 1973.

[18] J. M. Lucas, The rotation graph of binary trees is hamiltonian, *Journal of Algorithms* 8 (1987) 503–535.

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

[20] J. M. Lucas, A direct algorithm for restricted rotation distance, *Information Processing Letters* 90 (2004) 129–134.

[21] J. M. Lucas, Untangling binary trees via rotations, *The Computer Journal* 47 (2004) 259–269.

[22] F. Luccio and L. Pagli, On the upper bound on the rotation distance of binary trees, *Information Processing Letters* 31 (1989) 57–60.

[23] E. Mäkinen, On the rotation distance of binary trees, *Information Processing Letters* 26 (1987/88) 271–272.

[24] J. Pallo, Enumerating, ranking and unranking binary trees, *The Computer Journal* 29 (1986) 171–175.

[25] J. Pallo, On the rotation distance in the lattice of binary trees, *Information Processing Letters* 25 (1987) 369–373.

[26] J. Pallo, Some properties of the rotation lattice of binary trees, *The Computer Journal* 31 (1988) 564–565.

[27] J. Pallo, An efficient upper bound of the rotation distance of binary trees, *Information Processing Letters* 73 (2000) 87–92.

[28] J. Pallo, Right-arm rotation distance between binary trees, *Information Processing Letters* 87 (2003) 173–177.

[29] J. Pallo, Rotational tree structures on binary trees, *Proc. 11th International Conference on Automata and Formal Languages* (AFL'05), Dobogoko, Hungary, May 17-20, 263–274.

[30] A. Proskurowski, On the generating of binary trees, *Journal of the ACM* 27 (1980) 1–2.

[31] R. O. Rogers and R. D. Dutton, Properties of the rotation graph of binary trees, *Congressus Numerantium* 109 (1995) 51–63.

[32] R. O. Rogers and R. D. Dutton, On distance in the rotation graph of binary trees, *Congressus Numerantium* 120 (1996) 103–113.

[33] R. O. Rogers, On finding shortest paths in the rotation graph of binary trees, *Congressus Numerantium* 137 (1999) 77–95.

[34] D. Rotem, On a correspondence between binary trees and a certain type of permutation, *Information Processing Letters* 4 (1975) 58–61.

[35] F. Ruskey and A. Proskurowski, Generating binary trees by transpositions, *Journal of Algorithms* 11 (1990) 68–84.

[36] D. D. Sleator and R. E. Tarjan, Self-adjusting binary search trees, *Journal of the ACM* 32 (1985) 652–686.

[37] D. D. Sleator, R. E. Tarjan, and W. R. Thurston, Rotation distance, triangulations and hyperbolic geometry, *Journal of the American Mathematical Society* 1 (1988) 647–681.

[38] R. Sundar, On the deque conjecture for the splay algorithm, *Combinatorica* 12 (1992) 95–124.

[39] V. Vajnovszki, On the loopless generation of binary tree sequences, *Information Processing Letters* 68 (1998) 113–117.

[40] S. Zaks, Lexicographic generation of ordered trees, *Theoretical Computer Science* 10 (1980) 63–82.

[41] D. Zerling, Generating binary trees using rotations, *J. Assoc. Comput. Mach.* 32 (1985)