Twist-Rotation Transformations of Binary Trees and Arithmetic Expressions

Ming Li[*], Louxin Zhang[†]

Abstract

The paper studies the computational complexity and efficient algorithms for the twist-rotation transformations of binary trees, which is equivalent to the transformation of arithmetic expressions over an associative and commutative binary operation. Main results are 1) a full binary tree with n labeled leaves can be transformed into any other in at most $3n \log n + 2n$ twist and rotation operations, 2) deciding the twist-rotation distance between two binary trees is NP-complete, and 3) the twist-rotation transformation can be approximated with ratio $6 \log n + 4$ in polynomial time.

1 Introduction

This paper studies two equivalent transformation problems. Consider the collection of full parenthesized expressions of n variables over an associative and commutative binary operation '+'. A *move* is an application of either the associative law (to regroup three consecutive subexpressions) or the commutative law (to exchange two subexpressions that are combined by the operation '+'). An interesting problem is how many moves are required to convert one expression into another. This problem and other expression transformations were discussed in studying the effect of algebraic laws on code generation [5, 11]. When certain algebraic laws such as the associative, commutative and distributive laws apply for an expression E, we are not required to generate codes for E, but we may generate codes for any equivalent expression E' obtained by applications of the laws.

A binary tree is *full* if every internal node has both left and right children [6]. Let T_n be the set of all full binary trees with n labeled leaves. We allow two operations on

[*]Supported in part by the NSERC Operating Grant OGP0046506, ITRC, a CGAT grant and DIMACS.
Address: Department of Computer Science, University of Waterloo, Waterloo, Ont. N2L 3G1, Canada. E-mail: mli@math.uwaterloo.ca

[†]Address: Kent Ridge Digital Labs, 21 Heng Mui Keng Terrace, Singapore 119613. BioInformatics Center, National University of Singapore; E-mail: lxzhang@krdl.org.sg. The work was done in University of Waterloo and supported by a CGAT grant.

a tree in T_n: a *twist*, in which the left and right subtrees of a specified internal node are exchanged, and a *rotation*, in which an internal node changes places with one of its children while the symmetric order in the tree is preserved. These two operations are illustrated in Figure 1. The problem is to transform a full binary tree with labeled leaves into another using the minimum number of twists and rotations. Several other tree transformations were also proposed and studied in molecular biology [9, 10, 14] and pattern matching [8].

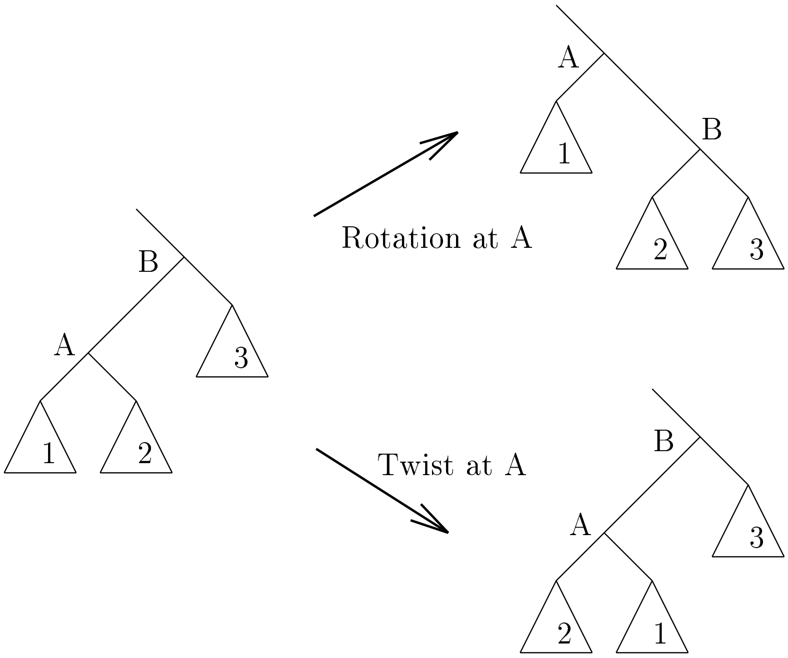

Figure 1: *A twist and a rotation*

These two transformation problems are obviously equivalent. The correspondence between full binary trees and arithmetic expressions over a binary operation is standard: a labeled leaf corresponds to a variable, and an internal node corresponds to an expression $(E_l + E_r)$, where E_l and E_r are the subexpressions corresponding to the left and right children of the node respectively. A rotation corresponds to the application of the associative law, and a twist to an application of the commutative law.

This problem was first studied by Sleator, Tarjan and Thurston [13]. In their paper, by studying short encodings of transformation sequences, they obtained a lower bound $\frac{n \log n}{4} - O(n)$ for the number of twist and rotation operations, O_n, required in worst case to convert a full binary tree with labeled leaves into another. On the other hand, Culik and Wood studied a slight different transformation system called *nearest neighbor interchange*[1], which was abstracted from studying phylogenetic (unordered) trees in biology [9, 10] and further studied in [2, 7] extensively. The Culik and Wood's result implies an upper bound $12n \log n + O(n)$ for O_n. If rotation is only allowed, the corresponding upper bound is exactly $2n - 6$ [12].

In this paper, we study the computational complexity and efficient algorithms for the twist-rotation transformation of full binary trees. We first show that a full binary tree with n labeled leaves can be converted into any other by at most $3n \log n + 2n$ twist-rotation operations in section 2. To this end, we use an analogue of merge sort, which is completely different from the approach used in [1]. In section 3, we show that deciding the twist-rotation distance is NP-complete by reduction from the Exact 3-Set Cover problem. Finally, we will present a polynomial time algorithm with approximation ratio $6 \log n + 4$ for transforming a full binary tree with n uniquely labeled leaves into another in section 4.

2 An Upper Bound

In this section, by using an analogue of merge-sort, we provide a better upper bound for the number of operations required in worst case for the twist-rotation transformation of full binary trees with same labeled leaves. In the rest of this paper, we assume that all leaves in a full n-node binary tree are labeled with integers in the range from 1 to n.

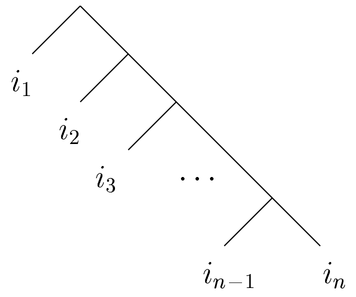

Figure 2: *Right line tree* $R = [i_1, i_2, \dots, i_n]$.

Theorem 2.1 *A full binary tree with* n *labeled leaves can be transformed into any other in at most* $3n \log n + 2n$ *twists and rotations.*

Proof. Given two full binary trees T and T' with n labeled leaves each, a transformation from T to T' using at most $3n \log n + 2n$ operations is a combination of the following three transformations.

1. Transform T into a right line tree $R = [i_1, i_2, \dots, i_n]$ as shown in Figure 2, in which all internal nodes appear on a path and have a leaf as the left subtree. This can be done by using at most one rotation at each internal node. Thus this transformation consists of at most $n - 1$ rotations.

2. Similarly, transform T' into a right line tree R'. This also consists of at most $n-1$ rotations.

3. Use an analogue of merge-sort to sort leaves in the right line tree R so that the resulting tree is the right line tree R'. For clarity, we assume that the number of leaves, n, is a power of 2 and labels in R' are arranged in the ascending order, i.e. $R' = [1, 2, \ldots, n]$.

The algorithm converts R to R' in $\log_2 n$ phases. At the start of i-th phase ($0 \leq i < \log_2 n$), the resulting tree is a right line tree $R^{(i)}$ (assume $R^{(0)} = R$) where all leaves are divided into $\frac{n}{2^i}$ sorted blocks of 2^i leaves each and the sorting order of blocks alternates between ascending and descending. Let $k = \frac{n}{2^{i+1}}$. By assumption, k is a power of 2. During the i-th phase, starting at the middle of $R^{(i)}$, we merge the corresponding blocks $B^{(i)}_{k-j}$ and $B^{(i)}_{k+j+1}$ ($0 \leq j < k$) into a block $B^{(i+1)}_{k-j}$ of sorted leaves and pull out. The merging process is illustrated in Figure 3 (a). The resulting tree is a right line tree $R^{(i+1)}$ consisting of k sorted blocks of 2^{i+1} leaves each, again alternating.

Now we describe the i-th phase in detail. By assumption, the blocks $B^{(i)}_{2m-1}$ consists of 2^i leaves sorted in ascending order and $B^{(i)}_{2m}$ consists of 2^i leaves sorted in descending order for $m = 1, 2, \cdots, k$. Let j be an integer in the range from 0 to $k-1$. Recalled that $B^{(i)}_{k-j}$ and $B^{(i)}_{k+j+1}$ will be merged to form the block $B^{(i+1)}_{k-j}$. Without loss of generality, we may assume that $B^{(i)}_{k-j}$ is in ascending order. Then $B^{(i)}_{k+j+1}$ must be in descending order. The resulting block $B^{(i+1)}_{k-j}$ will also be in ascending order. Set $p = 2^i$. Let $B^{(i)}_{k-j} = [x_1, x_2, \ldots, x_p]$ and let $B^{(i)}_{k+j+1} = [y_1, y_2, \ldots, y_p]$. Then, $x_l < x_{l+1}$ and $y_l > y_{l+1}$ for each $l < p$. At the start of merging $B^{(i+1)}_{k-j}$ and $B^{(i)}_{k+j+1}$, the resulting tree is illustrated in the middle of Figure 3 (a). We first compare x_p with y_1. If $x_p > y_1$, we move x_p to the branch where $B^{(i+1)}_{k-j+1}$ locates, which will be called the *sorted branch*; otherwise, we move y_1 to the sorted branch. In general, if we move x_t into the sorted branch by comparing x_t to y_s, where $t, s \leq p$, then we will compare x_{t-1} to y_s at the following step. If we move y_s into the sorted branch by comparing x_t to y_s, then we will compare x_t to y_{s+1} instead. Moving x_t requires 2 rotations and 1 twist as illustrated in Figure 3 (b). Moving y_s into the sorted branch requires 1 twist and 1 rotations as illustrated in in Figure 3 (c).

The correctness of the algorithm is obvious. Since moving one leaf into the sorted branch requires at most 3 operations, each phase can be executed in $3n$ operations. Overall, the whole transformation from R to R' uses at most $3n \log n$ operations. Thus the combination of above transformations takes at most $3n \log n + 2n$ operations to transform a full binary tree with labeled leaves to another. $\square$

3 NP-Completeness

Given two full binary trees T_1 and T_2, the *twist-rotation distance* between them is defined to be the minimum number of twists and rotations required to transform T_1 to T_2, denoted by $d(T_1, T_2)$. It is easily to see that such a distance satisfies the triangle inequality and the symmetry property. Here we prove that deciding the twist-rotation distance is NP-complete by a reduction from the Exact 3-Set Cover (X3C) problem [3]. Formally, deciding the twist-rotation distance is the decision version of the problem of computing the twist-rotation distance, i.e. given two full binary trees T_1 and T_2 and an integer d, decide if $d(T_1, T_2) \leq d$ or not.

Theorem 3.1 *Deciding the twist-rotation distance is NP-complete for full binary trees with labeled leaves.*

Proof. Obviously, it is in NP. We now prove its NP-hardness. Given an instance of the X3C problem: $S = \{s_1, s_2, \ldots, s_{3m}\}$ and n 3-subsets of S, $C_i = \{x_i, y_i, z_i\}$ $(1 \leq i \leq n)$. We construct two full binary trees T_1 and T_2 with labeled leaves as illustrated in Figure 4. T_1 consists of n long left line subtrees of length $3K + 2$ each, where K is a large integer to be specified later. At the bottom of the i-th long left line subtrees, there are three leaves labeled with the three elements in C_i, and all the other leaves are identically labeled with an extra element, say e, not in the set S. In T_2, there are $n - m$ long left line subtrees of length $3K - 1$ and $3m$ short left line subtrees of length K. The lowest leaf of the i-th short left line subtree is labeled with the i-th element $s_i \in S$. At the right bottom of T_2, there are $3(n - m)$ leaves labeled with elements in the multi-set $\cup_{i=1}^{n} C_i - S$. All the other leaves in T_2 are labeled with the element e. Our reduction is based on the following fact.

Lemma 3.1 $d(T_1, T_2) \leq 3nK + 2n + 7m + 3(4n - m)\log(4n - m)$ *if there is an X3C cover of S; otherwise, $d(T_1, T_2) \geq 3nK + K$.*

Proof. Suppose there is a cover of S, say $C_{i_i}, C_{i_2}, \ldots, C_{i_m}$. Then we transform T_1 into T_2 in the following way. For the long left line subtree with last three leaves labeled with elements in $C_{i_j} = \{x, y, z\}$ $(1 \leq j \leq m)$, we leave the leaf z at the bottom and move leaves x and y up using K rotations. The resulting tree is shown in Figure 5 (b). Then, we use 3 operations to split x and y so that we can leave y there (Figure 5 (c)). We then move x up continuously together with the subtree consisting of leaf z and other $K - 1$ leaves using K rotations (Figure 5 (d)). Then we use 4 operations to move x over the subtrees containing y and z (Figure 5 (e)). Again, we move the subtrees containing y and z up using K rotations (Figure 5 (f)). Finally, we use 2 rotations to splits three subtrees containing x and y and z respectively. Overally, the whole procedure takes $3K + 9$ operations. For a long left line subtree with last three leaves labeled with elements in a set C_i not in the cover, we simply move the last three leaves up. This takes $3K + 2$ operations.

After breaking all long left line subtrees into small ones, we convert it into T_2 by shuffling all small left line subtrees. Since there are $4n - m$ small subtrees, using the analogue of merger sort in the proof of Theorem 2.1, we can shuffle these subtrees using at most $3(4n - m)\log(4n - m)$ twist-rotation operations. Thus $d(T_1, T_2) \leq 3nK + 2n + 7m + 3(4n - m)\log(4n - m)$.

Assume that there is no cover. Then, to transform T_1 into T_2, $3nK+6$ are still necessary to move up all the leaves labeled with elements in S. However, since we do not have a cover, either one leaf labeled with an element in S must travel down a left line subtree of length K, or a left line subtree of length K must merge with another one. Both costs at least K operations. Thus, $d(T_1, T_2) \geq 3nK + K$. This finishes the proof of the lemma. □

Now we continue the proof of Theorem 3.1. Choose $K > 2n+7m+3(4n-m)\log(4n-m)$. Then, we have the fact that S has a cover if and only if $d(T_1, T_2) < 3nK + K$. Thus we have proved that deciding the twist-rotation distance is NP-complete. □.

Equivalently, we have

Theorem 3.2 *The following problem is NP-complete:*

> **Instance:** *Two full parenthesized expressions E_1 and E_2 of n variables over an associative, commutative binary operation, and an integer $k > 0$;*
>
> **Question:** *Can E_1 be transformed into E_2 using at most k applications of the associative and commutativity laws?*

Remarks. (1) If only the associative law is allowed, whether the corresponding problem is NP-complete or not is still unknown. The problem is equivalent to the rotation distance problem for binary trees and the diagonal-flip distance problem for triangulations of convex polygons [12].

(2) If only the commutativity law is allowed, the corresponding problem is polynomial time solvable. An algorithm for this problem can easily be designed using the dynamic programming technique and so is omitted.

(3) A subject in which the result is related is that of evaluation of arithmetic expressions. Assume that an expression over some variables is given and assume the some standard arithmetic laws hold. An interesting problem in code optimization is to find an equivalent expression with the fewest number of operations. In this aspect, the following problem is known to be NP-complete [4]: Given a bilinear form B over $\{0, 1\}$ and an integer k, the problem of determining whether B can be computed with k additions is NP-complete.

4 A Logarithmic Approximation

In this section, we present a polynomial time algorithm with approximation ratio $6 \log n + 4$ for twist-rotation transformation of full binary trees. Given a full binary tree T, its root is the unique internal node of degree 2. For simplicity, we add an extra leaf to a full binary tree T, called the *superroot*, and connect it to the root of T. After the modification, each internal node is adjacent to three nodes, its left child, right child and parent, and the twist-rotation distance is unchanged. Therefore, in the rest of this section, we will focus on such extended trees.

Given an extended, full binary tree T. An edge is called *internal* edge if it connects two internal nodes, and *external* if it is adjacent to a leaf. After the removal of an internal edge e, T is divided into two disjoint subtrees, each of which contains a non-empty subset of leaves. Therefore, each internal edge induces a partition of labeled leaves in T.

Given two extended, full binary trees T_1 and T_2 with n labeled leaves. Two edges e_1 in T_1 and e_2 in T_2 are a *good pair* if one of the following conditions is true:

1). They are both left (or right) internal edges and induce the same partition of leaves in T_1 and T_2;

2). They are both left (or right) external edges adjacent to the same labeled leave.

Definition 4.1 *Let T_1 and T_2 be two extended, full binary trees. A set $M \subseteq E(T_1) \times E(T_2)$ is a compatible edge-matching between T_1 and T_2 if the following two conditions hold:*

1). For any $(e, e') \in M$, e and e' form a good pair;

2). For any three elements $(e_1, e_1'), (e_2, e_2'), (e_3, e_3') \in M$, e_2 and e_3 belong to a subtree of T_1 formed after the removal of e_1 if and only if e_2' and e_3' belong to a subtree of T_2 formed after the removal of e_1'.

Recall that $d(T_1, T_2)$ denotes the twist-rotation distance between T_1 and T_2. Let $m(T_1, T_2)$ denote the cardinality of the maximum compatible edge-matchings between T_1 and T_2. Then we have

Lemma 4.1

$$d(T_1, T_2) \geq \frac{|E(T_1)| - m(T_1, T_2)}{2}.$$

Proof. For each edge $e \in E(T_1)$ not appearing in a maximum compatible edge-matching $M(T_1, T_2)$, either the partition of leaves induced by it is inconsistent with that induced by any edge in T_2 or it has a different direction against an edge of T_2 that induces the same partition. Therefore, in order to transform T_1 to T_2, an operation must be applied to this edge. Since each twist or rotation can affect at most two edges, we have $d(T_1, T_2) \geq \frac{|E(T_1)| - m(T_1, T_2)}{2}$. $\square$

It is unknown how to find a maximum compatible edge-matching in polynomial time for two extended, full binary trees with labeled leaves in general. However, if all leaves are uniquely labeled, each internal edge induces a unique leaf partition. Thus, we can easily compute a maximum compatible edge-matching for trees with uniquely labeled leaves. So we are able to establish the following result.

Theorem 4.1 *The twist-rotation distance can be approximated within a factor of $6 \log n + 4$ in polynomial time for any two full binary trees with n uniquely labeled leaves.*

Proof. Given two extended, full binary trees T_1 and T_2 with n uniquely labeled leaves, we first find a maximum compatible edge-matching $M(T_1, T_2)$ between T_1 and T_2. Let E_1 and E_2 denote the set of all the edges in T_1 and T_2 not appearing in $M(T_1, T_2)$ respectively. The edge set E_i induces a subforest F_i of T_i for $i = 1, 2$. Let F_1 have j connected components

$$T_{11}, T_{12}, \ldots, T_{1j},$$

and F_2 have k connected components

$$T_{21}, T_{22}, \ldots, T_{2k}.$$

Then, $j = k$ and there is one-to-one correspondence between these two component decompositions such that two corresponding components T_{1p} and T_{2q} are surrounded with the same set of good edges in a maximum compatible edge-matching $M(T_1, T_2)$.

We convert T_1 to T_2 by transforming each T_{1p} to its correspondence T_{2q} using at most $3|E(T_{1p})| \log |E(T_{1p})| + 2|E(T_{1p})|$ operations in top-down order. By Lemma 4.1, the approximation ratio of our algorithm is at most

$$\frac{\sum_{i=1}^{j} |E(T_{1p})|(3 \log |E(T_{1p})| + 2)}{\sum_{i=1}^{j} |E(T_{1p})|/2} \leq 6 \log n + 4.$$

This concludes the proof. $\square$

5 Concluding Remarks

We have proved that deciding twist-rotation distance is NP-complete and also presented an efficient approximation algorithm for the twist-rotation transformation of full binary trees in a special case when their leaves are uniquely labeled. Thus the following problems arise naturally:

1. Can we approximate the twist-rotation transformation of full binary trees within a good ratio (say logarithmic or constant) in general?

2. Can the upper bound in Theorem 2.1 be improved? What is the exact bound for the number, O_n, of twists and rotations required in worst case for transforming a full binary tree with labeled leaves into another?

These problems are much harder than they appear to be. Another problem is about compatible edge-matchings between two binary trees and is stated as follows:

3. Is the maximum compatible edge-matching polynomial time computable for any two full binary trees? If not, is it polynomial time approximated within constant factor?

Obviously, a positive answer to the first part of the last problem implies a positive solution to the first question.

References

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

[2] B. DasGupta, X. He, T. Jiang, M. Li, J. Tromp and L. Zhang. On distance between phylogenetic trees. In *Proc. of The Eighth Annual ACM-SIAM Symposium on Discrete Algorithms,* 427-436, 1997.

[3] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness,* W. H. Freeman, San Francisco, 1979.

[4] T. Gonzalez and J. Ja'Ja', On the complexity of computing bilinear forms with 0,1 constants, *Journal of Computer and System Science* **20** (1980), 85-95.

[5] T. Gonzalez and J. Ja'Ja', Evaluations of arithmetic expressions with algebraic identities. *SIAM Journal of Computing* **11** (1982), 633-662.

[6] D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Addison-Wesley, Reading, MA, 1968.

[7] M. Li, J. Tromp and L. Zhang, On the nearest neighbor interchange distance, *J. of Theoretical Biology* **182** (1996), 463-467.

[8] S. Y. Lu, A tree-to-tree distance and its application to cluster analysis. *IEEE Transactions on Pattern Analysis Machine Intelligence* **1** (1979), 323-332.

[9] G. W. Moore, M. Goodman and J. Barnabas, An iterative approach from the standpoint of the additive hypothesis to the dendrogram problem posed by molecular data sets, *Journal of Theoretical Biology* **38** (1973), 423-457.

[10] D. F. Robinson, Comparison of labeled trees with valency three, *Journal of Combinatorial Theory*, Series B, **11** (1971), 105-119.

[11] R. Sethi and J. D. Ullman, The generation of optimal code for arithmetic expressions, *Journal of the Association for Computing Machinery* **17** (1970), 715-728.

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

[13] D. Sleator, R. Tarjan, W. Thurston, Short encodings of evolving structures, *SIAM Journal on Discrete Mathematics* **5** (1992), 428-450.

[14] M. S. Waterman and T. F. Smith, On the similarity of dendrograms, *Journal of Theoretical Biology* **73** (1978), 789-800.

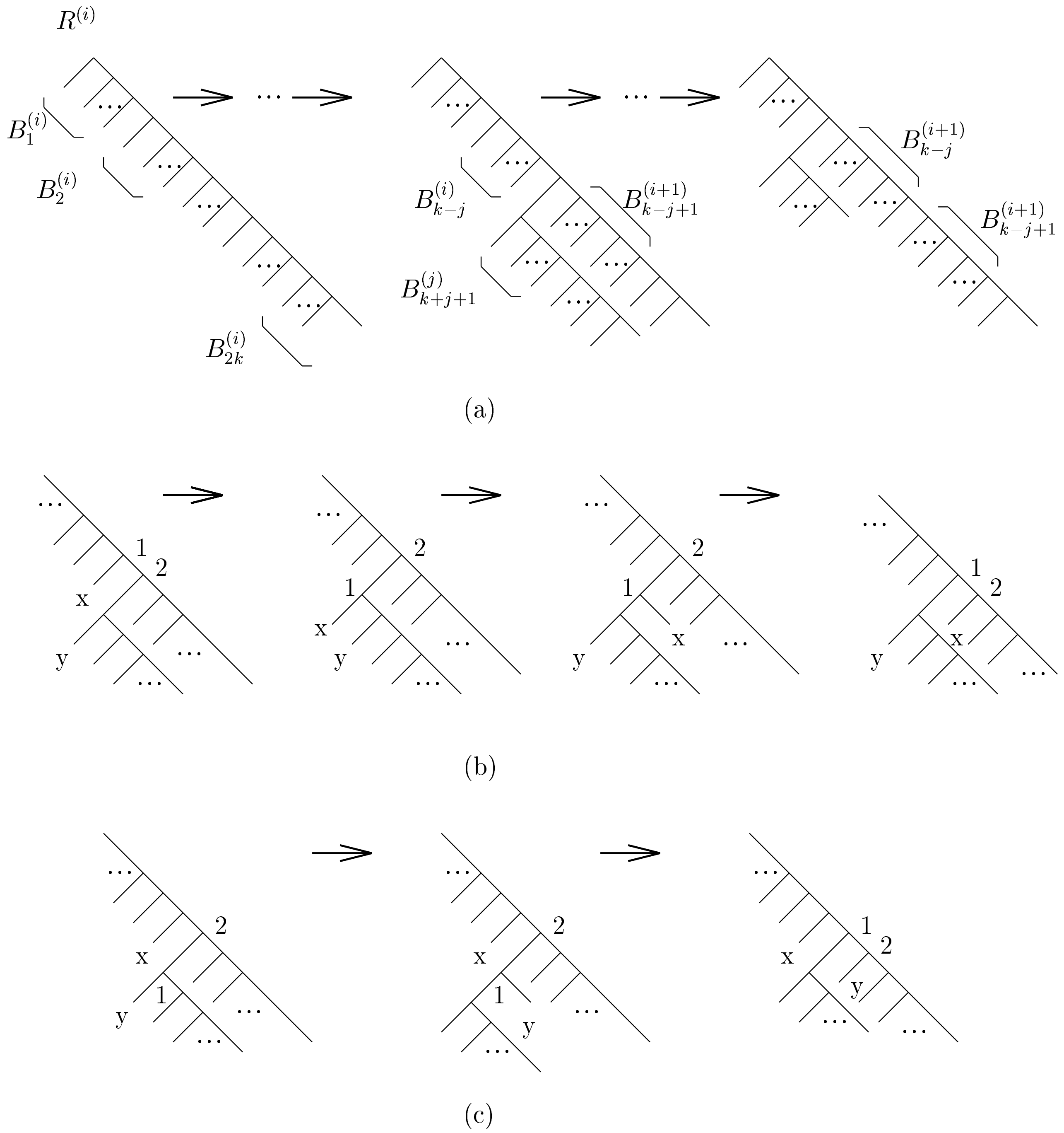

Figure 3: *Sorting leaves in a right line tree. (a) Merge blocks $B_{k-j}^{(i)}$ and $B_{k+j+1}^{(i)}$ into a new block $B_{k-j}^{(i+1)}$ of $B^{(i+1)}$; (b) Move $x = x_t$ into the sorted branch; (c) Move $y = y_s$ into the sorted branch.*

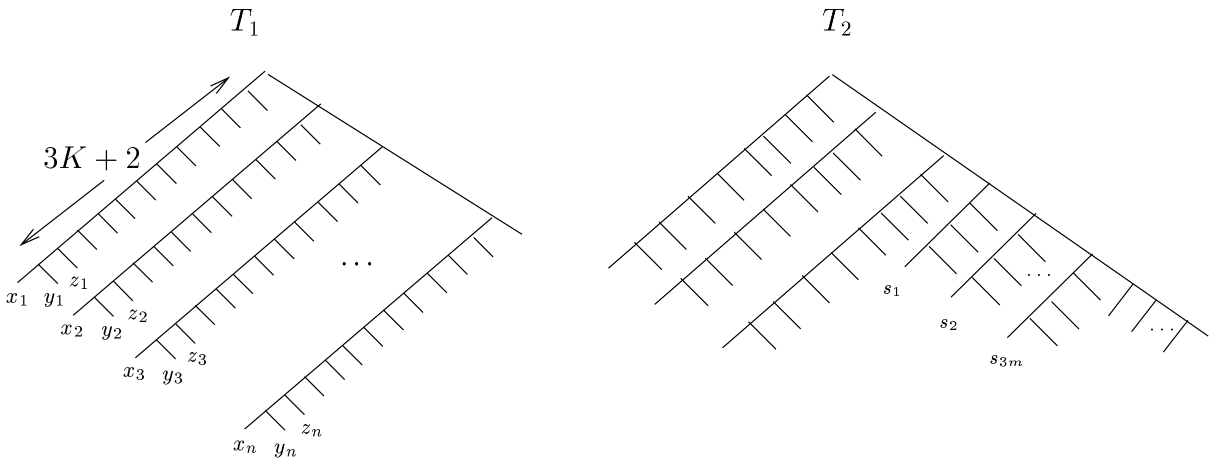

Figure 4: *Binary trees T_1 and T_2 in the proof of Theorem 3.1.*

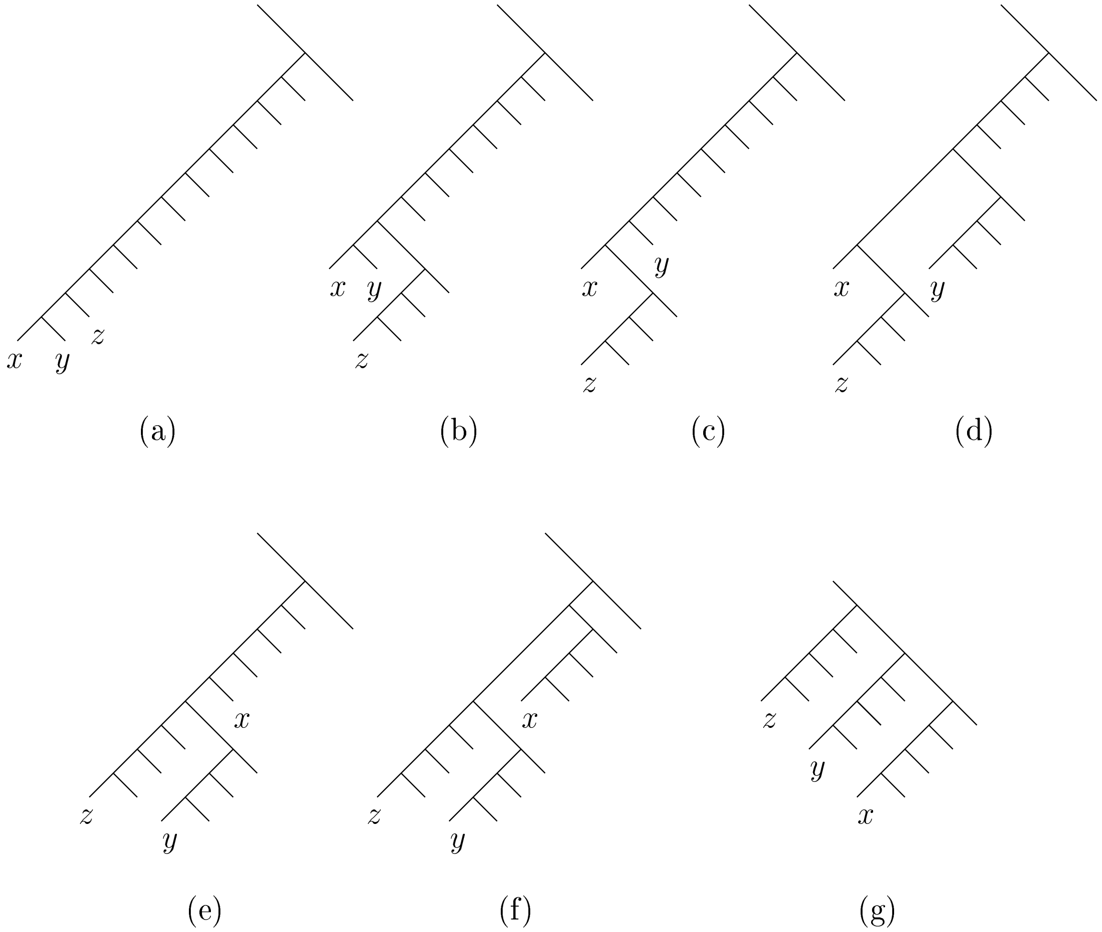

Figure 5: *Breaking a long left line subtree into three short ones where* $K = 3$.