Constant-memory Iterative Generation of Special Strings Representing Binary Trees

Sebastian Smyczyński

Faculty of Mathematics and Computer Science,
Nicolaus Copernicus University, Toruń, Poland
smyczek@mat.umk.pl

Abstract. The shapes of binary trees can be encoded as permutations having a very special property. These permutations are tree permutations, or equivalently they avoid subwords of the type 231. The generation of binary trees in natural order corresponds to the generation of these special permutations in the lexicographic order. In this paper we use a stringologic approach to the generation of these special permutations: decompositions of essential parts into the subwords having staircase shapes. A given permutation differs from the next one with respect to its tail called here the working suffix. Some new properties of such working suffixes are discovered in the paper and used to design effective algorithms transforming one tree permutation into its successor or predecessor in the lexicographic order. The algorithms use a constant amount of additional memory and they look only at those elements of the permutation which belong to the working suffix. The best-case, average-case and worst-case time complexities of the algorithms are $O(1)$, $O(1)$, and $O(n)$ respectively. The advantages of our stringologic approach are constant time and iterative generation, while other known algorithms are usually recursive or not constant-memory ones.

Keywords: tree permutations, stack-sortable permutations, 231-avoiding permutations, enumeration of binary trees

1 Introduction

The generation in natural order of the shapes of binary trees with n nodes corresponds to the lexicographic generation of all special permutations of elements $1, 2, \dots, n$. This is easy when done recursively and more technical when done iteratively. Our goal is to do it iteratively and at the same time with small time and small space (constant-memory). The natural order of trees as well as its corresponding tree permutations are defined in [3]. It's worth mentioning that the natural order of binary trees is also called an A-order of binary trees [6] and tree permutations are often referred to as stack-sortable permutations or 231-avoiding permutations [13].

In this paper we explore stringologic approach and consider carefully the structure of subwords of special permutations. We introduce the notion of the working suffix of the permutation and reveal its staircase structure. The working suffix is a concatenation of descending staircases, with bottom points of staircases strictly increasing.

We say that the permutation $p = (p_1, p_2, \dots, p_n)$ of the integer numbers $1, 2, \dots, n$ is 231-*avoiding* (or that p avoids the pattern 231) if there are no such indices $1 \leq i_2 < i_3 < i_1 \leq n$ such that $p_{i_1} < p_{i_2} < p_{i_3}$.

In other words the subsequence (a, b, c) matches the pattern 231 iff

$$a \ < \ b \ > \ c \ < \ a.$$

A permutation p avoids the pattern 231 if there is no subsequence of p which matches pattern 231 (see Figure 1).

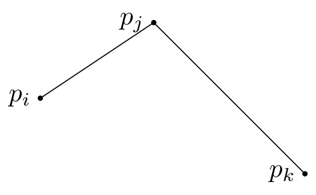


Figure 1. Subsequence (p_i, p_j, p_k) matching the pattern 231. Permutation p is 231-avoiding if there is no subsequence of p matching the pattern 231.

Example 1. The permutation

$$(4, 1, 2, 5, 3, 6, 7)$$

is not a 231-avoiding since the subsequence $4, 5, 3$ matches (in the sense of order) pattern 231, while the permutation

$$(4, 1, 2, 3, 5, 6, 7)$$

avoids the pattern 231.

A *binary tree* T is either a *null tree* or it consists of a node called the *root* and two binary trees denoted $left(T)$ and $right(T)$. Let $|T|$ denote the *size* of T. In the former case, the *size* of T is zero; in the latter case, $|T| = 1 + |left(T)| + |right(T)|$. The *natural order* [3] of binary trees follows the recursive definition:
We say that $T_1 \prec T_2$ if

1. $|T_1| < |T_2|$, or
2. $|T_1| = |T_2|$ and $left(T_1) \prec left(T_2)$, or
3. $|T_1| = |T_2|$ and $left(T_1) = left(T_2)$ and $right(T_1) \prec right(T_2)$,

This order is related to the order relation given by D. E. Knuth in [4, Sec. 2.3.1, excercise 25] specialized to unlabeled binary trees, and is also known as A-order of binary trees [6].

Let T be a binary tree on n nodes. We can represent the tree T as a sequence of the integer numbers $1, 2, \ldots, n$ first labeling the nodes with their position's number as they appear in the inorder traversal of the tree and then listing those labels as they appear in the preorder traversal of the tree. We shall call such a representation *preorder-inorder representation* and the corresponding sequence *tree permutation.*

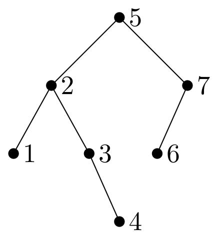


Figure 2. A binary tree T and the string $(5, 2, 1, 3, 4, 7, 6) = preorder(inorder(T))$ representing its shape. $(5, 2, 1, 3, 4, 7, 6)$ is the tree permutation of T.

Interestingly, the natural order of binary trees is preserved by the lexicographic order on their preorder-inorder representation [1] (see Figure 3).

Lemma 2. *For the binary trees $T_1 \prec T_2$ iff the tree permutation of T_1 is lexicographically smaller than tree permutation of T_2.*

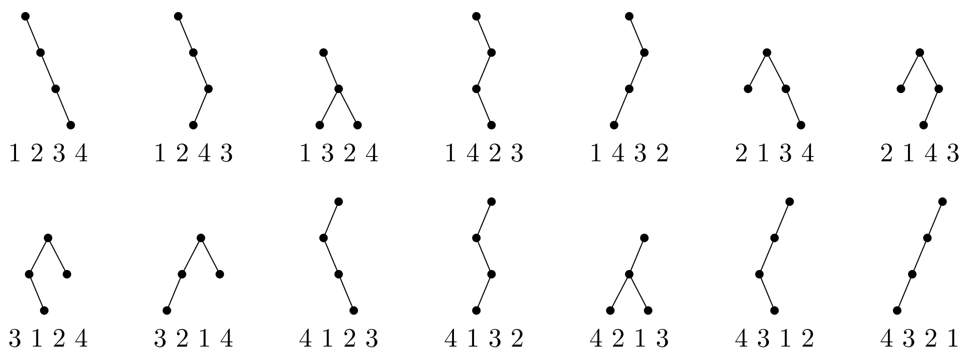


Figure 3. Binary trees for $n = 4$ listed in the lexicographic order on their preorder-inorder representation

The most important property of tree permutations which is employed in this paper is their equivalence with 231-avoiding permutations [1].

Lemma 3. *A permutation p is a tree permutation iff it is* 231*-avoiding.*

Further on we will refer to this property as *the basic property*.

Let $p = (p_1, p_2, \ldots, p_n)$ be a tree permutation. The suffix of p which makes p different from its successor in the lexicographic order is called *working suffix*.

Example 4. For tree permutations for $n = 4$ the working suffixes are underlined as follows:

$$\begin{array}{ccccccc}
1\,2\,\underline{3\,4} & 1\,\underline{2\,4\,3} & 1\,\underline{3\,2\,4} & 1\,4\,\underline{2\,3} & \underline{1\,4\,3\,2} & 2\,1\,\underline{3\,4} & \underline{2\,1\,4\,3} \\
3\,\underline{1\,2\,4} & \underline{3\,2\,1\,4} & 4\,1\,\underline{2\,3} & 4\,\underline{1\,3\,2} & 4\,\underline{2\,1\,3} & 4\,3\,\underline{1\,2} & 4\,3\,2\,1
\end{array}$$

The *working suffix* for permutation 1234 is 34 since the next permutation is 1243. The *working suffix* for 4321 is empty, since there is no successor for 4321.

We shall call a decreasing sequence of consecutive numbers a *descending stairs sequence* and refer to its first (largest) element as *the top* of the sequence and last (smallest) element as *the bottom* of the sequence. A single number always forms *trivial descending stairs sequence.*

2 Working Suffix Properties

In this section we elaborate on *the basic property* of tree permutations and present a few properties of the working suffix. Those properties in consequence let us construct effective algorithms transforming a given tree permutation into its successor or predecessor in the lexicographic order.

Lemma 5. *Let p be a tree permutation and let i be the index of the first position of its working suffix. If tree permutation q is the successor of p in the lexicographic order, then $q_i = p_i + 1$.*

Proof. Since the suffix $q_i, q_{i+1}, \dots, q_n$ is a permutation of the working suffix $p_i, p_{i+1}, \dots, p_n$ then there exists an index $k > i$ such that $q_k = p_i$.

Suppose for the sake of contradiction that $q_i > p_i + 1$. There exists an index j such that $p_j = p_i + 1$. Due to *the basic property* $j > i$ since otherwise we would have three indices $j < i < k$ for which $q_k = p_i < q_j = p_i + 1 < q_i$.

Having such an index $j > i$ for which $p_j = p_i + 1$ we can construct a permutation r by exchanging elements p_j and p_i and then sorting the suffix starting at index $i+1$ in ascending order. The permutation $r = (p_1, p_2, \dots, p_{i-1}, p_i + 1, r_{i+1}, \dots, r_n)$ is a valid tree permutation with $r_i = p_i + 1 > p_i$ and $r_{i+1} < r_{i+2} < \dots < r_n$. Hence $p \prec r$ and $r \prec q$ which contradicts with q being the successor of p. □

Lemma 6. *Let p be a tree permutation and let i be the index of the first position of its working suffix, then there exist no such indices j, k such that $i < j < k$ and $p_k = p_j + 1$.*

Proof. The proof is similar to that of Lemma 5.

Let $q = (q_1, q_2, \dots, q_{i-1}, q_i, q_{i+1}, \dots, q_n)$ be the successor of p. Since the working suffix of p starts at index i therefore $q = (p_1, p_2, \dots, p_{i-1}, q_i, q_{i+1}, \dots, q_n)$, with $q_i = p_i + 1$.

Suppose for the sake of contradiction that there exist indices $i < j < k$ such that $p_k = p_j + 1$. Then we can construct a new permutation r from p by exchanging elements p_j with p_k and sorting the suffix starting at index $j + 1$ in ascending order. The permutation $r = (p_1, p_2, \dots, p_{j-1}, p_k, r_{k+1}, r_{k+2}, \dots, r_n)$ with $r_j = p_k > p_j$ and $r_{k+1} < r_{k+2} < \dots < r_n$ is a valid tree permutation. Hence we have $p \prec r$ and also $r \prec q$ which contradicts with q being the successor of p. □

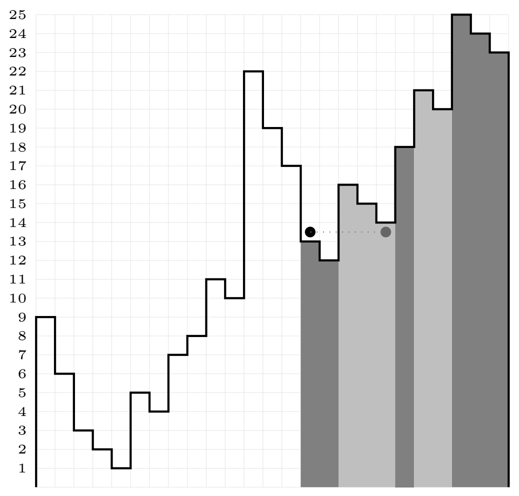
Lemma 7. *Let $p = (p_1, p_2, \dots, p_n)$ be a tree permutation and i be the starting index of its working suffix. For any index $i \le k < n$, $p_k > p_{k+1}$ implies that $p_k = p_{k+1} + 1$.*

Proof. Let $q = (q_1, q_2, \dots, q_{i-1}, q_i, q_{i+1}, \dots, q_n)$ be the successor of p. Since the working suffix of p starts at index i, therefore $q = (p_1, p_2, \dots, p_{i-1}, q_i, q_{i+1}, \dots, q_n)$, with $q_i = p_i + 1$.

Assume that there exists an index k such that $i \le k < n$ and $p_k > p_{k+1}$. Suppose for the sake of contradiction that $p_k > p_{k+1} + 1$. Let j be an index for which $p_j = p_{k+1} + 1$. Due to *the basic property* $j > k + 1$. We obtain a contradiction since we have indices $i < k + 1 < j$ for which $p_j = p_{k+1} + 1$ which is impossible with respect to Lemma 6. □

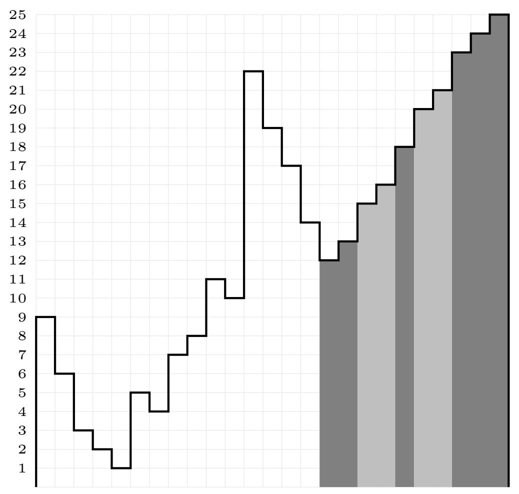
Theorem 8. *For any tree permutation $p = (p_1, p_2, \dots, p_n)$ its (not empty) working suffix starting at index i forms a staircase of descending stairs sequences (possibly of length 1) for which the top element p_i of the first descending stairs sequence is equal to $p_j - 1$, where j is the index of the bottom step of the following second descending stairs sequence. Furthermore for no other indices $i < k < l$, $p_k = p_l - 1$.*

This theorem is the direct consequence of the previously proven lemmas 5,6,7. See Figure 4 for graphic interpretation of this theorem.



$p = (9, 6, 3, 2, 1, 5, 4, 7, 8, 11, 10, 22, 19, 17, \mathbf{13}, 12, 16, 15, \mathbf{14}, 18, 21, 20, 25, 24, 23)$

Figure 4. The tree permutation p represented in a graphic form with shaded parts corresponding to the special staircase structure of the working suffix. The black dot appears above the top of the first sequence of descending stairs and the gray one at the bottom of the following sequence of descending stairs.



$q = (9, 6, 3, 2, 1, 5, 4, 7, 8, 11, 10, 22, 19, 17, \mathbf{14}, 12, \mathbf{13}, 15, 16, 18, 20, 21, 23, 24, 25)$

Figure 5. Graphic representation of the tree permutation q, which is the successor of the tree permutation p from Figure 4.

3 The Algorithm

The lemmas 5, 6 presented in the previous section provide us enough information about the structure of the working suffix to design the algorithm transforming given tree permutation p into its successor. The algorithm consists of two steps:

1. Finding the first pair of indices $i < j$ starting from the end of p such that $p_j = p_i + 1$. From lemmas 5 and 6 we know that index i must be then the starting position of the working suffix of p.

2. Transforming the found working suffix by exchanging the elements p_i and p_j and then sorting the suffix starting from position $i+1$ in ascending order.

The theorem 8 on the other hand gives us an exact way how to implement the above mentioned steps effectively by exploiting the staircase structure of the working suffix.

The algorithm $Next(p)$ presented in this section is in fact a direct implementation of the theorem 8.

Algorithm 1: Next(P)

```
Step 1. Find the working suffix.

Let lbs, cbs and cs denote respectively the index of the bottom of the last seen
descending stairs sequence, the index of the bottom of the current descending
stairs sequence, and the current step index.

$lbs := n$; $cbs := n$; $cs := n$;
repeat
    $cs := cs - 1$;
    If we processed the whole permutation then there is no next one
    if $cs < 1$ then
        return false;
    end
    Check if we reach the bottom of the previous descending stairs sequence
    if $P[cs] < P[cs+1]$ then
        $lbs := cbs$;
        $cbs := cs$;
    end
until $P[lbs] = P[cs] + 1$ ;

Step 2. Changing the working suffix to form the successor of $P$. The cs points
at the first element of the working suffix with $P[lbs] = P[cs] + 1$.

Exchange the elements $P[cs]$ and $P[lbs]$, and then sort the suffix starting at
position $cs + 1$ in ascending order (by reversing the stairs).

swap($P[cs]$, $P[lbs]$);
$cs := cs + 1$;
while $cs < n$ do
    Let es denote the end of the current descending stairs sequence
    $es = cs + 1$;
    while $es \leq n$ and $P[es] < P[cs]$ do
        $es := es + 1$;
    end
    Change the current descending stairs sequence into increasing sequence
    reverse($P$, $cs$, $es - 1$);
    $cs := es$;
end
return true; The permutation $P$ has been transformed to its successor.
```

The work done by the algorithm $Next(p)$ can simply be reverted. During the execution of the algorithm $Next(p)$ the staircase of descending stairs sequences is changed into a staircase of ascending stairs sequences. The element which was at the starting position of the working suffix is the top of the first of ascending stairs sequences, and the element which is placed at the starting position of the working suffix after the execution of the algorithm delimits the new staircase of ascending stairs sequences and simply allows us to find the starting index of the working suffix.

The algorithm $Prev(p)$ transforms the given tree permutation p into its predecessor in the lexicographic order.

Algorithm 2: Prev(P)

```
Step 1. Find the working suffix of P predecessor.

Let cts and cs denote respectively the index of the top of the current ascending
stairs sequence, and the current step index.

$cts := n$; $cs := n$;
repeat
    $cs := cs - 1$;
    If we processed whole permutation then there is no previous one
    if $cs < 1$ then
        return false;
    end
    Check if we reach the top of the preceding ascending stairs sequence
    if $P[cs] < P[cs + 1] - 1$ then
        $cts := cs$;
    end
until $P[cs] = P[cts] + 1$ ;

Step 2. Changing the working suffix to form the predecessor of P. The cs
points at the first element of the working suffix with $P[cs] = P[cts] + 1$.

Exchange the elements $P[cs]$ and $P[cts]$, and then reverse each ascending
stairs sequence in the rest of the working suffix starting at position $cs + 1$.

swap($P[cs]$, $P[cts]$);
$cs := cs + 1$;
while $cs < n$ do
    Let es denote the end of the current ascending stairs sequence
    $es = cs + 1$;
    while $es \leq n$ and $P[es] = P[es - 1] + 1$ do
        $es := es + 1$;
    end
    Change the current ascending stairs sequence into descending sequence
    reverse($P$, $cs$, $es - 1$);
    $cs := es$;
end
return true; The permutation P has been transformed to its predecessor.
```

4 Time Complexity

Let us recall that the number of binary trees with n nodes or equivalently the number of the tree permutations of length n is given by the Catalan number [4]

$$C_n = \binom{2n}{n}/(n+1).$$

Let W_n be the sum of lengths of the working suffixes for all tree permutations. Since the algorithm $Next(p)$ performs work proportional to the length of the working suffix of the given permutation p, therefore its average-case time-complexity in enumerating all C_n tree permutations is $O(\frac{W_n}{C_n})$.

Each tree permutation p can be represented as $p = p_1p'p''$ where p' is itself a tree permutation of length $p_1 - 1$ and p'' is a tree permutation of length $n - p_1$ translated by having p_1 added to each element [3] (see Figure 6).

$$p = 8\ \ \overbrace{4\ \ 1\ \ 3\ \ 2\ \ 6\ \ 5\ \ 7}^{p'}\ \ \overbrace{12\ \ 11\ \ 9\ \ 10}^{\overbrace{4\ \ 3\ \ 1\ \ 2}^{p''}\oplus 8}$$

Figure 6. Example of the decomposition of tree permutation $(8, 4, 1, 3, 2, 6, 5, 7, 12, 11, 9, 10)$.

Using this recursive property of the tree permutations we can formulate the recurrence formula for W_n. If we fix p_1 then p'' is a tree permutation of length $n - p_1$ and p' of length $p_1 - 1$. There are exactly C_{p_1-1} permutations of length $p_1 - 1$. So each working suffix of p'' appears in p exactly C_{p_1-1} times. Therefore the summarized length of all working suffixes of p starting in p'' is equal to $C_{p_1-1}W_{n-p_1}$.

When the working suffix of p starts in p' then its length is equal to the length of the working suffix of p' plus length of p'' which is equal to $n - p_1$. Since the working suffix of p starts in p' each time this permutation is going to be changed (except the last change which will be connected with changing the value of p_1) then the summarized length of working suffixes starting at p' is equal to $W_{p_1-1} + (n - p_1)(C_{i-1} - 1)$.

Now summarizing those values for each possible value of p_1 and adding $n(n - 1)$ for the summarized length of all working suffixes which changes the whole string, we obtain the following recurrence equation:

$$W_n = \sum_{i=1}^{n} \Big(C_{i-1}W_{n-i} + W_{i-1} + (n - i)(C_{i-1} - 1) \Big) + n(n - 1)$$

Solving this recurrence we obtain $W_n = C_{n+1} - n - 1$.

Since the Catalan numbers satisfy the recursive relation $C_1 = 1, C_{n+1} = \frac{2(2n+1)}{n+2}C_n$, we may conclude that the average-case time-complexity of the algorithm $Next(p)$ is constant since $O(\frac{W_n}{C_n}) = O(1)$.

This proves the following theorem:

Theorem 9. (Main Result) *For a tree permutation p we can compute the next tree permutation in the lexicographic order in constant amortized time using only a constant amount of additional memory.*

The algorithm $Prev(p)$ presented in the previous section was obtained by a simple modification of the $Next(p)$ algorithm and similarly performs work proportional to the length of the working suffix of the given permutation p, therefore we obtain a similar result:

Theorem 10. *For a tree permutation p we can compute the previous tree permutation in the lexicographic order in constant amortized time using only a constant amount of additional memory.*

Acknowledgements

The author thanks W. Rytter for the scientific pressure.

References

1. M. C. Er: *On enumerating tree permutations in natural order.* International Journal of Computer Mathematics, 22 1987, pp. 105–115.
2. T. Feil, K. Hutson, and R. M. Kretchmar: *Tree traversals and permutations.* Congressus Numerantium, 172 2005, pp. 201–211.
3. G. D. Knott: *A numbering system for binary trees.* Commun. ACM, 20(2) 1977, pp. 113–115.
4. D. E. Knuth: *The Art of Computer Programming, Volume I: Fundamental Algorithms, 3rd Edition*, Addison-Wesley, 1997.
5. D. E. Knuth: *The Art of Computer Programming, Volume 4: Generating All Trees*, Addison Wesley, 2006.
6. J. Pallo and R. Racca: *A note on generating binary trees in A-order and B-order.* International Journal of Computer Mathematics, 18 1985, pp. 27–39.
7. A. Proskurowski: *On the generation of binary trees.* J. ACM, 27(1) 1980, pp. 1–2.
8. D. Rotem: *Stack sortable permutations.* Discrete Mathematics, 33(2) 1981, pp. 185–196.
9. D. Rotem and Y. L. Varol: *Generation of binary trees from ballot sequences.* J. ACM, 25(3) 1978, pp. 396–404.
10. M. H. Solomon and R. A. Finkel: *A note on enumerating binary trees.* J. ACM, 27(1) 1980, pp. 3–5.
11. R. P. Stanley: *Enumerative Combinatorics, Vol. 2*, vol. 62 of Cambridge Studies in Advanced Mathematics, Cambridge University Press, 1999.
12. J. West: *Generating trees and forbidden subsequences*, in Proceedings of the 6th conference on Formal power series and algebraic combinatorics, Amsterdam, The Netherlands, The Netherlands, 1996, Elsevier Science Publishers B. V., pp. 363–374.
13. J. West: *Permutations with restricted subsequences and stack-sortable permutations*, PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1999.