Pop & push: Ordered tree iteration in $O(1)$-time

Harry Treeman
Arboretum College

Abstract

The number of ordered trees (also known as plane trees) with n nodes is the $(n-1)$st Catalan number C_{n-1}. An ordered tree can be stored directly using nodes and pointers, or represented indirectly by a Dyck word. This paper presents a loopless algorithm for generating ordered trees with n nodes using pointer-based representations. In other words, we spend $O(C_{n-1})$-time to generate all of the trees, and moreover, the delay between consecutive trees is worst-case $O(1)$-time.

To achieve this run-time, each tree must differ from the previous by a constant amount. In other words, the algorithm must create a type of Gray code order. Our algorithm operates on the children of a node like a stack, by popping the first child off of one node's stack and pushing the result onto another node's stack. We refer to this pop-push operation as a pull, and consecutive trees in our order differ by one or two pulls. There is a simple two-case *successor rule* that determines the pulls to apply directly from the current tree. Moreover, the first case relocates a path, while the second case relocates a path and a leaf. Interestingly, our rule applies a left-shift in the corresponding Dyck word, and these shifts generate a cool-lex variant of lexicographic order.

Our results represent the first pull Gray code for ordered trees, and the first fully published loopless algorithm for ordered trees using pointer representations. More importantly, our algorithm is incredibly simple: A full implementation in C, including initialization and output, uses only three loops and three if-else blocks.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Mathematics of computing → Combinatorics; Mathematics of computing → Combinatorial algorithms

Keywords and phrases combinatorial generation, ordered trees, Gray code, loopless, cool-lex

Digital Object Identifier 10.4230/LIPIcs.ESA.2022.

Funding *Harry Treeman*: Funding

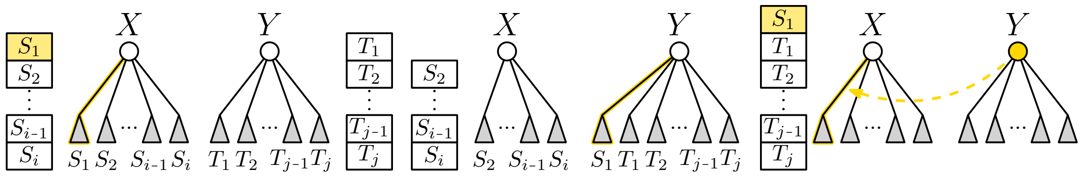
1 Introduction

This article is focused on iterating through every ordered tree with n nodes as quickly and simply as possible. By quickly we mean *loopless*, which is worst-case $O(1)$-time per tree. This means that we store a single ordered tree in memory, and in constant time we change it into the next tree. In other words, there is always a *constant delay* between successive trees.

To achieve such an algorithm we need to first develop a suitable order. That is, we need to create an ordering in which each tree differs from the previous tree by a constant amount. When measuring the difference between two trees, it is necessary to know how they are represented, or stored in memory. We focus on link-based representations, in which the parent-child relationships are specified using pointers or references.

The way in which we operate on the ordered trees is novel. We treat the children of each node like a stack, and we modify a tree by popping and pushing these stacks. In particular, a *pull* involves popping one stack, and pushing the result onto another stack. In other words, we remove the first subtree of one node, and insert it as the new first subtree of another node. The pull operation is illustrated in Figure 1a.

Remarkably, we are able to generate all ordered trees by only pulling subtrees that are paths. More specifically, we create each successive tree using one or two of these more restrictive *path-pull* operations. Moreover, the location of the pulled paths is very predictable, which allows us to locate them in constant time. This point is illustrated in Figure 1b. A sample listing of all ordered trees with $n = 6$ nodes appears in Figure 4.



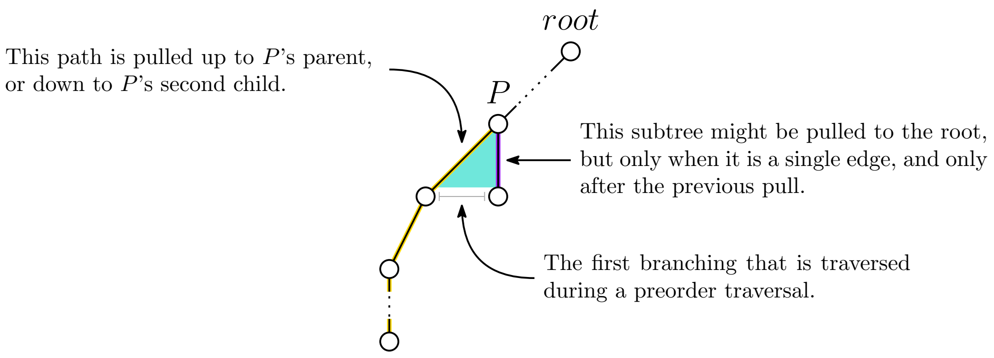
S_1
S_2
$S_{i\text{-}1}$
S_i
X
Y
$S_1\ S_2\ S_{i\text{-}1}\ S_i$
$T_1\ T_2\ T_{j\text{-}1}\ T_j$
T_1
T_2
$T_{j\text{-}1}$
T_j
$S_2\ S_{i\text{-}1}\ S_i$
$S_1\ T_1\ T_2\ T_{j\text{-}1}\ T_j$

Ordered tree $\mathcal{T}$ with nodes X and Y, and their respective stacks of children.

The result of $\text{pull}_{\mathcal{T}}(Y, X)$, and the updated stacks.

Denoting $\text{pull}_{\mathcal{T}}(Y, X)$ with an arrow.

(a) The pull operation relocates first subtrees. In (a) the node X has children rooted at $S_1, S_2, \ldots, S_i$ with $i \geq 1$, and the node Y has children $T_1, T_2, \ldots, T_j$ with $j \geq 0$. The children are stored in a stack as shown on the right. The result of Y pulling X is shown in (b). During this operation, the node Y pulls the first child off of X, and adds the result as its new first child. This operation can be understood as popping X's stack, and pushing the result onto Y's stack, as shown on the right. We use the style in (c) when illustrating pulls in subsequent figures n



root
P
This path is pulled up to P's parent, or down to P's second child.
This subtree might be pulled to the root, but only when it is a single edge, and only after the previous pull.
The first branching that is traversed during a preorder traversal.

(b) The next ordered tree is always created by one or two pulls. Furthermore, the pulls always involve subtrees that are paths near a specific location of the tree.

Figure 1 Our Gray code for ordered trees is based on relocating subtrees. More specifically, a subtree is always removed as a first child ('pop') and added as a first child ('push') and we refer to this stack-based change as a 'pull' as shown in (a). In addition, the pulled subtrees are always paths, and they are always located on either side of the tree's first branching in a preorder traversal (in turquoise) as shown in (b). A successor rule with two cases provides the details in (3) and Figure 3. Figure 5 has loopless (i.e., worst-case $O(1)$-time per tree) implementations in pseudocode and C.

One of the most notable aspects of our algorithm is its simplicity. Indeed, our C implementation uses only three loops and three if-else blocks, some of which are used for initialization and output.

Our main result is summarized by the following theorem. Figure 5 provides the main loop of Algorithm O in both pseudocode and C.

▶ **Theorem 1.** *Algorithm O iterates through the ordered trees with n nodes looplessly using only one or two path-pulls between successive trees.*

We refer to the order in which the trees are generated as a *Gray code*, in reference to the well-known binary reflected Gray code. More descriptively, it is a *pull Gray code*, or a *path-pull Gray code*, and Theorem 1 represents the first such order in either case. Algorithm O is also the first fully published loopless algorithm for generating ordered trees with a linked-based representation. The secret to our algorithm is *cool-lex order*, which is a well-known variation of co-lexicographic order, and which has previously been used to generate binary and k-ary trees. These details are further expanded upon in the next subsection.

1.1 Relationship to previous results

Many previous papers have focused on efficiently generating ordered trees, and other closely related objects, using various representations. For broad overviews of these results, we recommend Knuth's coverage of generating combinatorial objects in Volume 4A of *The Art of Computer Programming* [2], and Mütze's recent update [5] of Savage's well-known survey [11].

Skarbek provided the first algorithm for generating link-based representations of ordered trees in *constant amortized time (CAT)*, meaning that the delay is $O(1)$-time in an amortized sense, rather than worst-case [14]. Korsh and Lafolette [3] crafted a loopless algorithm for generating ordered trees with a fixed branching sequence. In other words, they generated subsets of ordered trees in which the number of children at each node form the same multiset. Their algorithm used a string-based representation, rather than a link-based representation, and thus answered a challenging open problem posed by van Baronaigien [17]. However, they also stated, in one sentence, that their results could be adapted to link-based representations. By 'layering' the output of that proposed algorithm, it may be possible to create a loopless link-based algorithm for generating all ordered trees with n nodes. It is also important to note that the results in [3] are not simple: All told, the provided C code includes over one hundred instances of the `if` and `else` keywords. Additional related results include [1, 6, 18].

Cool-lex order is a variation of co-lexicographic order that was first introduced for (s, t)-combinations by Ruskey and Williams [8, 9]. Since then it has been used to create Gray codes and efficient algorithms for a variety of combinatorial objects, with recent examples including multiset necklaces [13] and fixed-content Łukasiewicz words [4]. In particular, the application of cool-lex order to Dyck words [9] is relevant later in the article.

1.2 Outline

In Section 2 we discuss ordered trees in more detail. Section 3 then provides our Gray code, and Section 4 shows how to generate it efficiently. Our proof of correctness is found in Section 5, and we conclude with final remarks in Section 6.

2 Ordered Trees

In this section, we provide background information on ordered trees, starting with terminology in Section 2.1 and link-based representations in Section 2.2. Then in Section 2.3 we introduce the stack-based pull operation, and a further specialization in Section 2.4.

2.1 Terminology and Conventions

Throughout the article, we typeset ordered trees as $\mathcal{T}$, and we use n to denote the number of nodes in a tree. We typeset nodes of a tree as X, and we frequently refer to a node and the subtree rooted at that node interchangeably. Code is written with a `monospaced` font.

When discussing ordered tree $\mathcal{T}$, we use the term *first branching*, and this term could be misinterpreted. We intend it to mean the following: The first branching occurs when the preorder traversal goes up an edge, and then down an edge, for the first time. The node that is incident to both of these edges is the parent of the first branching. For example, these first branches are highlighted in turquoise in Figure 4. In particular, note that the subtree to the left of the first branch must be a path (see Lemma 2).

Another way of describing the first branching is as follows. Let O be the first second-child node that is visited during a preorder traversal. If P is the parent of O, then P is the parent of the first branching.

The number of ordered trees with n nodes is the $(n-1)$st Catalan number, C_{n-1}.

2.2 Link-Based Representations

We focus on *link-based* representations of ordered trees, meaning that children are linked by pointers or references. There are at least two natural variations, as discussed below.

- Each node has a first child reference and a right sibling reference. The children of a given node `o` form a linked list `o.first.right.right...`. In particular, a leaf `o` has a `o.first == null`, while a rightmost child of a node has `o.right == null`.
- Each node has a child array and a number of children. In particular, a leaf has `num == 0` and `child[0] == null` (using 0-based indexing).

We refer to the first as a *linked list representation* and the second as a *link array* representation.

Throughout this paper, we treat the children of a node as a stack. Since stacks can be implemented with constant-time operations using a linked list or an array, this allows us to ignore the specific type of link-based representation. However, our focus is on the linked list representation.

2.3 Stack Operations: Pop, Push, and Pull

Given an ordered tree $\mathcal{T}$ and a non-leaf node A, let $\text{pop}_{\mathcal{T}}(A)$ remove the first subtree of A and return it. In other words, if the child-list of A is viewed as a stack, then this operation pops the stack. In particular, if A has only one child in $\mathcal{T}$, then A will be a leaf in $\text{pop}_{\mathcal{T}}(A)$.

If $\mathcal{S}$ is a non-empty ordered tree, then let $\text{push}_{\mathcal{T}}(A, \mathcal{S})$ be the result of adding $\mathcal{S}$ as a new first subtree to node A. In other words, if the child-list of A is viewed as a stack, then this operation pushes a new subtree $\mathcal{S}$ onto the stack. In particular, if A is a leaf in $\mathcal{T}$, then A will have one child in $\text{push}_{\mathcal{T}}(A, \mathcal{S})$.

We combine these two operations by popping a node, and then pushing the removed subtree into the tree that was created by the pop. Note that the tree that results from this combined operation will have the same number of nodes as the original tree. Furthermore, the resulting ordered tree differs from the original ordered tree, if and only if, the nodes that are popped and pushed are not the same.

We refer to the combined pop-push operation as a *pull*, and we define it as follows

$$\text{pull}_{\mathcal{T}}(A, B) \text{ is } \text{push}_{\mathcal{T}'}(A, \text{pop}_{\mathcal{T}}(B)) \text{ where } \mathcal{T}' \text{ is the modification of } \mathcal{T} \text{ after popping.} \quad (1)$$

We think of $\text{pull}_{\mathcal{T}}(A, B)$ as node A pulling the first subtree off of B and adding it as its own first child. For this reason, we'll illustrate the operation with an arrow from A to B that shows how the subtree is pulled toward B.

Finally, we also use a *double pull*, which pulls from a given node twice in a row. In other words, the first subtree is pulled from the node, and then the new first subtree is pulled from the node. We define the operation and its notation as follows.

$$\text{pull}_{\mathcal{T}}(A; C, B) \text{ is } \text{pull}_{\mathcal{T}}(A, B) \text{ then } \text{pull}_{\mathcal{T}'}(C, B) \text{ where } \mathcal{T}' \text{ is created by the first pull.} \quad (2)$$

When illustrating (double) pulls in Figure 3, we use gold for the first pull, and purple for the second pull.

2.4 Path-Pulls

We refer to a pull operation as a *path-pull* if the pulled subtree is a path. All of the pull operations used in our Gray code are path-pulls. This distinction does not change the

```
▷ $A$ pulls $B$'s first child
function PULL($A$, $B$)
    $temp \leftarrow B.first$
    $B.first \leftarrow temp.right$
    $temp.right \leftarrow A.first$
    $A.first \leftarrow temp$
    $temp.parent \leftarrow A$
```

```
// A pulls B's first child.
node *pull(node *A, node *B){
  node *temp = A.first;
  B.first = temp.right;
  temp.right = A.first;
  A.first = temp;
  temp.parent = A;
}
```

Figure 2 The `pull` operation can be implemented as a `pop` followed by a `push`, namely, `pull(A,B)` is `push(A, pop(B))`. It can also be implemented directly, as shown above in pseudocode and C. The pulled node must not be a leaf.

efficiency of the operation, but it will become relevant in Section 5. The following lemma states that pulling on the left side of the first branching is always a path-pull.

▶ **Lemma 2.** *If $\mathcal{T}$ is an ordered tree, and P be the parent of its first branching, then the operation* $\text{pull}_{\mathcal{T}}(X, P)$ *is a path-pull for any node $X \neq P$.*

Proof. Since P is the parent of the first branching, all of the nodes in the first subtree of P must have only one child. Hence, the first subtree of P is a path. ◀

3 Gray Code order using Pulls

In this section, we define the order in which our algorithm will generate ordered trees with n nodes. In Section 5, we'll prove that the definition is correct, in the sense that every ordered tree will be created.

3.1 First and Last Trees

Our order starts and ends with two path-like trees.

- The first tree $\mathcal{T}_1$ has a root with two children, and the subtree rooted at the second-child is a path of length $n-2$.
- The last tree $\mathcal{T}_0$ is the unique path on n nodes.

For examples of these trees, refer to the beginning and end of Figure 4.

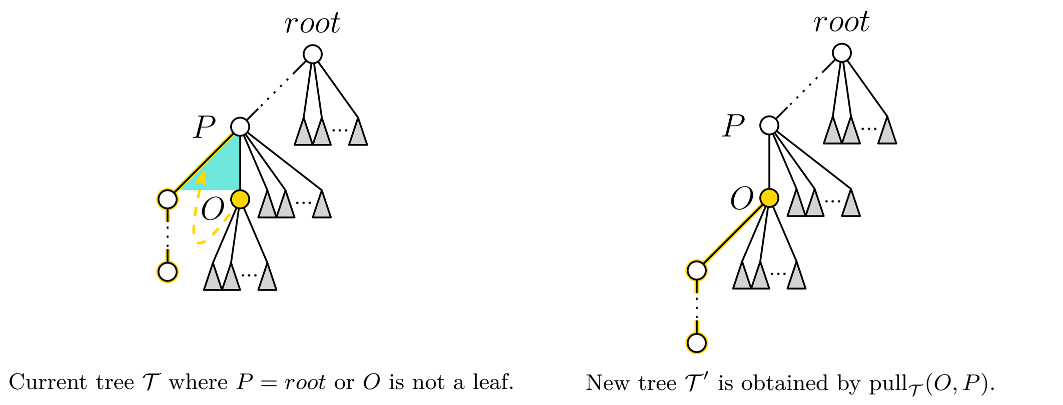
The rest of the order is generated by a successor rule, which transform any ordered tree — other than $\mathcal{T}_0$ — into the next ordered tree.

3.2 Successor Rule

Given an ordered tree $\mathcal{T}$ that is not a path (i.e., $\mathcal{T} \neq \mathcal{T}_0$, we distinguish several nodes that will be referenced by the successor rule. Let O be the first second-child that is visited during a preorder traversal of $\mathcal{T}$, and P be its parent, and G be its grandparent. In other words, P is the parent of the first branching that is traversed during a preorder traversal of $\mathcal{T}$, meaning that the traversal goes up into P and back down into its second-child O. Note that P and O are well-defined so long as $\mathcal{T} \neq \mathcal{T}_0$, while the grandparent G is only well-defined when the parent P is not equal to the *root* of the tree $\mathcal{T}$. Given these nodes, the successor rule can now be stated.

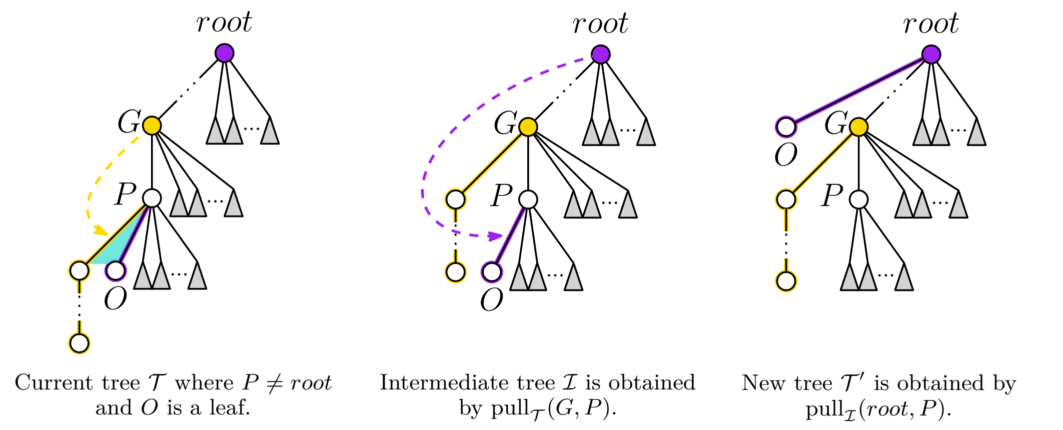
$$\text{next}(\mathcal{T}) = \begin{cases} \text{pull}_{\mathcal{T}}(O, P) & \text{if } P = root \text{ or } O \text{ has a child} \quad (3a) \\ \text{pull}_{\mathcal{T}}(G; root, P) & \text{otherwise (i.e. } P \neq root \text{ and } O \text{ is a leaf)} \quad (3b) \end{cases}$$

Figure 3 illustrates the pulls used in 3, while Figure 4 illustrates the full order for $n = 6$.



Current tree $\mathcal{T}$ where $P = root$ or O is not a leaf. New tree $\mathcal{T}'$ is obtained by $\text{pull}_{\mathcal{T}}(O, P)$.

(a) Case 3b. After the pull, the node O is updated to be the new second-child of O (if non-null) or the new second-child of P (if non-null), or *null*. The pull is equivalent to $\text{push}(O, \text{pop}(P))$.



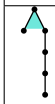
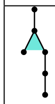
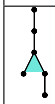
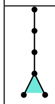
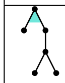
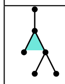
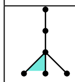
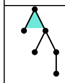
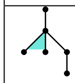
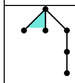
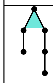
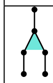
Current tree $\mathcal{T}$ where $P \neq root$ and O is a leaf. Intermediate tree $\mathcal{I}$ is obtained by $\text{pull}_{\mathcal{T}}(G, P)$. New tree $\mathcal{T}'$ is obtained by $\text{pull}_{\mathcal{I}}(root, P)$.

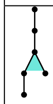
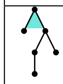
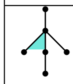
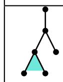
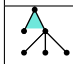
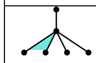
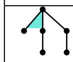
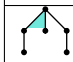
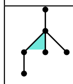
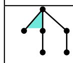
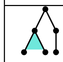
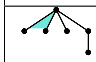
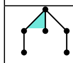
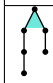
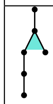
(b) Case 3b. After the pull, the node O is updated to be the new second-child of *root*. The pull operations are equivalent to $\text{push}(G, \text{pop}(P))$ followed by $\text{push}(root, \text{pop}(P))$

Figure 3 Our pull Gray code is generated by the two case successor rule in (3). This rule describes how to transform the current ordered tree $\mathcal{T}$ into the new next ordered tree $\mathcal{T}'$ using either one or two pulls. In these figures, O is the first node in a preorder traversal that is not on the path from the root to the leftmost descendent, and P is its parent, and G is its grandparent (if applicable). White circles denote non-null nodes, and grey triangles denote an unspecified number of children and subtrees. The pull operations, which are always path-pulls, are highlighted. The first pulling node and pulled path are in gold, while the second are in purple. The captions also explain how to update the value of O.

4 Loopless implementation

In this section, we show how to generate the order proposed in Section 3.2 by a loopless algorithm that we refer to as *Algorithm O*.

Tree	Dyck word	Next
	1011110000	(3a)
	1101110000	(3a)
	1110110000	(3a)
	1111010000	(3b)
	1011101000	(3a)
	1101101000	(3a)
	1110101000	(3b)
	1011011000	(3a)
	1101011000	(3b)
	1010111000	(3a)
	1100111000	(3a)
	1110011000	(3a)

Tree	Dyck word	Next
	1111001000	(3b)
	1011100100	(3a)
	1101100100	(3a)
	1110100100	(3b)
	1011010100	(3a)
	1101010100	(3b)
	1010110100	(3a)
	1100110100	(3a)
	1110010100	(3b)
	1011001100	(3a)
	1101001100	(3b)
	1010101100	(3a)
	1100101100	(3a)
	1110010100	(3a)
	1111000100	(3b)

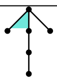
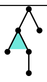
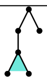
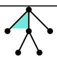
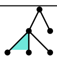
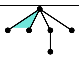
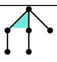
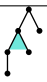
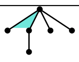
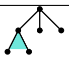
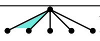
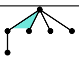
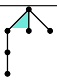
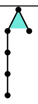
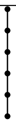
Tree	Dyck word	Next
	1011100010	(3a)
	1101100010	(3a)
	1110100010	(3b)
	1011010010	(3a)
	1101010010	(3b)
	1010110010	(3a)
	1100110010	(3a)
	1110010010	(3b)
	1011001010	(3a)
	1101001010	(3b)
	1010101010	(3a)
	1100101010	(3a)
	1110001010	(3a)
	1111000010	(3a)
	1111100000	

Figure 4 The ordered trees with $n = 6$ nodes as generated by our algorithm. The next tree is obtained by pull(O, P) when (3a) is specified, or by pull$(G; root, P)$ when (3b) is specified. These nodes are situated around the first branching that is traversed during a preorder traversal, which is highlighted in turquoise. More specifically, O is the second-child of this branching, P is its parent, and G is its grandparent. The last tree is the unique tree which has no such branching.

Algorithm O begins by creating the initial tree $\mathcal{T}_1$. Then it repeatedly applies the successor rule in (3). To apply the successor rule, the algorithm keeps track of node O, with Figure 3 illustrating how its location changes after each application of (3). Given the location of O, the individual pointer updates can be easily inferred from Figure 3. The successor rule is applied, and the next tree is visited, until the final tree $\mathcal{T}_0$ is created. This termination condition occurs exactly when O is set to null.

Figure 5 states the algorithm in pseudocode, and provides the main loop in C. A complete C implementation, including initialization and visiting routines, can be found in the appendix.

```
function COOL-ORDERED-TREES($n$)
    ▷ Generate initial tree
    $O \leftarrow root.first$
    visit(root)
    while $O \neq NULL$ do
        $P \leftarrow O.parent$
        if $O.first \neq NULL$ then
            pull($O, P$)
            $O \leftarrow O.first.right$
        else
            if $O.parent == root$ then
                pull($O, P$)
            else
                pull($O, P$)
                pull($root, P$)
            $O \leftarrow O.right$
        visit(root)
```

(a) Generate all ordered trees with $t+1$ nodes

```
void coolOtree(int n){
  node* root = get_initial_tree(n);
  node* o=root->first->right;
  visit(root);
  while(o){
    p=o->parent;
    if (o->first) { // if o has a child, shift 1
      pull(o,p);
      o = o->first->right;
    } else {
      if (p == root) { // if the string is tight, shift a 1
        pull(o,p);
      } else { // if the string isn't tight, shift a 0
        pull(p->parent,p);
        pull(root,p);
      }
      o = o->right;
    }
    visit(root);
  }
}
```

(b) C implementation of coolOtree

Figure 5 Algorithm O is presented in pseudocode in (a), and its main function is written in C in (b).

5 Proof of Correctness

We now prove that the order from Section 3 is correct, in the sense that it generates every ordered tree with n nodes, starting from $\mathcal{T}_1$ and ending at the path $\mathcal{T}_0$. Our approach is to understand the successor rule from Section 3.2 in terms of Dyck words. We start by recounting the standard bijection between ordered trees and Dyck words in Section 5.1, then we show how certain pull operations in ordered trees translate to changes in the associated Dyck words in Section 5.2. In Section 5.3, we recall the successor rule for generating Dyck words in cool-lex order from [9]. Finally, in Section 5.4, we prove that the successor rule for ordered trees proceeds in the same way as the cool-lex rule for Dyck words. In other words, our successor rule for ordered trees is guaranteed to generate all C_{n-1} ordered trees because the corresponding Dyck words are generated in cool-lex order.

5.1 Bijection with Dyck words

A *Dyck word of order n* is a binary string with n copies of 1 and n copies of 0, where every prefix has at least as many 1s as 0s. The number of Dyck words of order n is C_n.

The standard mapping from an ordered tree $\mathcal{T}$ with n nodes to a Dyck word of order $n-1$ involves running a preorder traversal of $\mathcal{T}$ and writing down 1 when going down an edge, and 0 when going up an edge [15]. Given an ordered tree $\mathcal{T}$, we let Dyck($\mathcal{T}$) be the corresponding Dyck word. Examples of this mapping can be found in Figure 4.

5.2 Pulls in Ordered Trees and Shifts in Dyck Words

Now we attempt to understand how the path-pulls in (3) translate to changes in the associated Dyck words. We'll find that the changes are *shifts*, which moves one bit elsewhere in the Dyck word. In particular, a *left-shift* moves a bit to the left, and a *right-shift* moves a bit to the right. We present two lemmas that mirror the cases in (3) and their depictions in Figure 6. Our first lemma is associated with the single pull used in (3a).

▶ **Lemma 3.** *Let $\mathcal{T}$ be an ordered tree that is not a path and let O be the first node in a preorder traversal that is not a first-child, and P be its parent. Let $\mathcal{T}'$ be the result of* $\text{pull}_{\mathcal{T}}(O, P)$*. Then there exists $p \geq q \geq 1$ and an $\alpha \in \{0,1\}^*$ in which*

$$\text{Dyck}(\mathcal{T}) = 1^p 0^q 1\alpha \text{ and } \text{Dyck}(\mathcal{T}') = 11^p 0^q \alpha. \tag{4}$$

In other words, the 1 at index $p+q+1$ is shifted to index 1.

Proof. From Figure 6a, we can see that $\text{pull}_{\mathcal{T}}(O, P)$ transforms $\mathcal{T}$ into $\mathcal{T}'$ with

$$\text{Dyck}(\mathcal{T}) = 1^a 11^b 0^b 01\alpha \text{ and } \text{Dyck}(\mathcal{T}') = 1^a 111^b 0^b 0\alpha.$$

Therefore, the result follows by setting $p = a + b + 1$ and $q = b + 1$. This is because $\text{Dyck}(\mathcal{T}) = 1^a 11^b 0^b 01\alpha = 1^p 0^q 1\alpha$ and $\text{Dyck}(\mathcal{T}') = 1^a 111^b 0^b 0\alpha = 11^p 0^q \alpha$. ◀

The second lemma considers the double pull used in (3b).

▶ **Lemma 4.** *Suppose that* (3b) *transforms $\mathcal{T}$ into $\mathcal{T}'$. That is, $\mathcal{T}$ is an ordered tree that is not a path, where O is a non-leaf node that is the first node in a preorder traversal that is not a first-child, $P \neq root$ is its parent, G is its grandparent, $\mathcal{I}$ is the result of* $\text{pull}_{\mathcal{T}}(G, P)$*, and $\mathcal{T}'$ is the result of* $\text{pull}_{\mathcal{I}}(root, P)$*. Then there exists $p < q$ and an $\alpha \in \{0,1\}^*$ in which*

$$\text{Dyck}(\mathcal{T}) = 1^p 0^q 10\alpha \text{ and } \text{Dyck}(\mathcal{T}') = 101^{p-1} 0^q 1\alpha. \tag{5}$$

In other words, the 0 at index $p+q+2$ is shifted to index 2.

Proof. From Figure 6b, we can see that $\text{pull}_{\mathcal{T}}(G, P)$ transforms $\mathcal{T}$ into $\mathcal{I}$ with

$$\text{Dyck}(\mathcal{T}) = 1^a 111^b 0^b 0\alpha \text{ and } \text{Dyck}(\mathcal{I}) = 1^a 11^b 0^b 01\alpha.$$

Then Figure 6c, shows that $\text{pull}_{\mathcal{I}}(root, P)$ transforms $\mathcal{I}$ into $\mathcal{T}'$ with

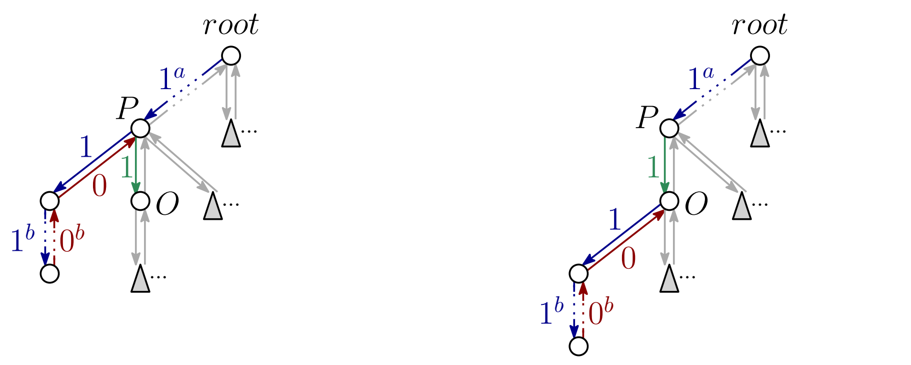
$$\text{Dyck}(\mathcal{I}) = 1^a 11^b 0^b 0110\alpha \text{ and } \text{Dyck}(\mathcal{T}') = 101^a 11^b 0^b 01\alpha.$$

Therefore, the result follows by setting $p = a + b + 1$ and $q = b + 1$. This is because $\text{Dyck}(\mathcal{T}) = 1^a 111^b 0^b 0\alpha = 1^p 0^q 10\alpha$ and $\text{Dyck}(\mathcal{T}') = 101^a 11^b 0^b 01\alpha = 101^{p-1} 0^q 1\alpha$. ◀

5.3 Cool-lex Order for Dyck Words

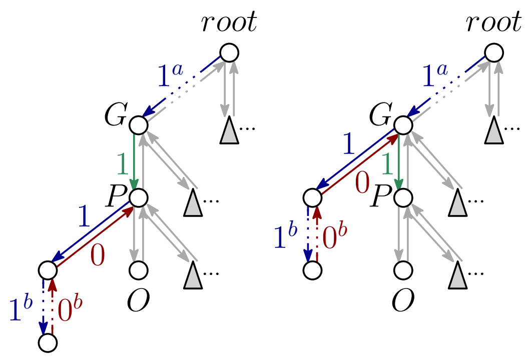
In [9] it was proven that the Dyck words of order n (also known as balanced parentheses) can be generated by a simple successor rule. The resulting order can be understood as a sublist of the cool-lex order of (n, n)-combinations [10] or binary strings of length $2n$ [16], and as a special case of the order for bubble languages [7] or fixed-content Łukasiewicz words [4][1], both of which can be generated efficiently [12, 4]. However, none of these results on strings have immediate implications for generating ordered trees.

[1] The recent paper of Lapey and Williams [4] uses cool-lex order to generate strings that correspond to ordered trees with a fixed branching sequence, however, it also does not provide a Gray code for the associated ordered trees.



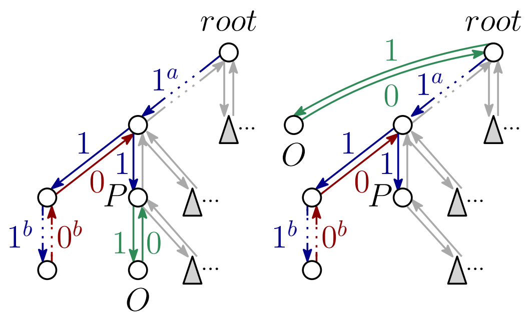
Ordered tree $\mathcal{T}$ with $\text{Dyck}(\mathcal{T}) = 1^a 1 1^b 0^b 0 1\alpha$. Ordered tree $\mathcal{T}'$ with $\text{Dyck}(\mathcal{T}') = 1^a 1 1 1^b 0^b 0\alpha$.

(a) The Dyck word is modified by shifting a single 1 to the left during the operation $\text{pull}_{\mathcal{T}}(O, P)$.



Ordered tree $\mathcal{T}$ with $\text{Dyck}(\mathcal{T}) = 1^a 1 1 1^b 0^b 0\alpha$. Ordered tree $\mathcal{I}$ with $\text{Dyck}(\mathcal{I}) = 1^a 1 1^b 0^b 0 1\alpha$.

(b) The Dyck word is modified by shifting a single 1 to the right during the operation $\text{pull}_{\mathcal{T}}(G, P)$.



Ordered tree $\mathcal{I}$ with $\text{Dyck}(\mathcal{I}) = 1^a 1 1^b 0^b 0 1 1 0\alpha$. Ordered tree $\mathcal{T}'$ with $\text{Dyck}(\mathcal{T}') = 1 0 1^a 1 1^b 0^b 0 1\alpha$.

(c) The Dyck word is modified by shifting 10 to the left during the operation $\text{pull}_{\mathcal{I}}(root, P)$.

Figure 6 Illustrating how several path-pull operations change the corresponding Dyck words. Lemmas 3–4 are illustrated in (a)–(c), with the first mirroring the pull in (3a) and Figure 3a, and the latter two mirroring the double pulls in (3b) and Figure 3b, respectively. Blue and red arrows indicate 1 and 0 bits during a preorder traversal, respectively, with green reserved for changes, and gray for suffixes that do not change.

▶ **Theorem 5** ([9]). *The cool-lex order of Dyck words of order n are generated starting from $101^{n-1}0^{n-1}$ by the following successor rule. In each case, $\alpha \in \{0,1\}*$ denotes a suffix that does not change.*

(a) *If the current string equals $1^p0^q11\alpha$ or $1^p0^p10\alpha$, then the next string is $11^p0^q1\alpha$ or $11^p0^p0\alpha$, respectively. In other words, the 1 at index $p+q+1$ is shifted to index 1.*

(b) *If the current string equals $1^p0^q10\alpha$ with $p > q$, then the next string is $101^{p-1}0^q1\alpha$. In other words, the 0 at index $p+q+2$ is shifted to index 2.*

The only string that is not covered by one of these cases is the last Dyck word, 1^n0^n.

Notice that in each case of Theorem 5, the next string is obtained by a left-shift. In other words, a single symbol is deleted, and then reinserted somewhere to the left

5.4 Proof of Correctness

Now we prove our main result, which was stated as Theorem 1 in Section 1.

Proof of Theorem 1. It is clear that Algorithm O in Figure 5 is loopless and correctly implements the order defined in Section 3. What remains to be proven is that the algorithm actually generates all of the ordered trees with n nodes. The proof equates the successor (3) for ordered trees into the cool-lex rule for Dyck words of order $n-1$.

First observe that the first ordered tree $\mathcal{T}_1$ has $\text{Dyck}(\mathcal{T}_1) = 101^{n-1}0^{n-1}$, which matches the first string in the cool-lex order of Dyck words. Similarly, the last ordered tree $\mathcal{T}_0$ has $\text{Dyck}(\mathcal{T}_0) = 1^n0^n$, which matches the last string in the cool-lex order of Dyck words. It remains to be proven that the successor rule for ordered trees in (3) matches the cool-lex successor rule for Dyck words. Fortunately, this follows immediately from Lemmas 3–4 and Theorem 5. ◀

6 Final Remarks

We conclude this article with additional results and open problems.

6.1 Additional Results

For simplicity and space constraints, we did not mention that our Gray code order is cyclic. More specifically, a single path-pull transforms the last ordered tree $\mathcal{T}_0$ into the first ordered tree $\mathcal{T}_1$.

Our presentation of pseudocode and C code is focused on the linked list implementation of the link-based representation of an ordered tree, however, the same results hold when using link arrays. This is because the children of a node can be stored in an array of sufficient size, and each pop and push can be performed at end of array.

The cool-lex order of Dyck words was also used to create a loopless algorithm for binary trees in [9]. By storing a an additional pointer to a node's left sibling, it is possible to augment our algorithms so that they simultaneously output a Gray code of ordered trees and binary trees. We are looking forward to publishing this in an extended version of the paper.

6.2 Open Problems

Research in combinatorial generation is often focused on finding the sharpest operation that is sufficient for ordering a given set of combinatorial objects. In this paper, we showed that two pulls are sufficient for listing ordered trees, even when the pulled subtrees are always paths, and when they are located near the first branching. This raises a number of follow-up questions.

- Is there a 1-pull Gray code for ordered trees? In other words, successive trees are obtained by one pull.
- Is there a 1-path-pull Gray code for ordered trees? This restricts the previous question by only allowing a single path to be pulled.

One could also consider these questions together with the aforementioned restriction to ordered trees with a fixed branching sequence. Ideally, a positive answer to any of these questions would also be accompanied by an efficient algorithm.

References

1 MC Er. Enumerating ordered trees lexicographically. *The Computer Journal*, 28(5):538–542, 1985.

2 Donald E Knuth. *Art of Computer Programming, Volume 4, Fascicle 4, The: Generating All Trees–History of Combinatorial Generation.* Addison-Wesley, 2013.

3 James F Korsh and Paul LaFollette. Multiset permutations and loopless generation of ordered trees with specified degree sequence. *Journal of Algorithms*, 34(2):309–336, 2000.

4 Paul Lapey and Aaron Williams. A shift Gray code for Lukasiewicz words. In *International Workshop on Combinatorial Algorithms*, page in press. Springer, 2022.

5 Torsten Mütze. Combinatorial Gray codes-an updated survey. *arXiv preprint arXiv:2202.01280*, 2022.

6 Victor Parque and Tomoyuki Miyashita. An efficient scheme for the generation of ordered trees in constant amortized time. In *2021 15th International Conference on Ubiquitous Information Management and Communication (IMCOM)*, pages 1–8. IEEE, 2021.

7 Frank Ruskey, Joe Sawada, and Aaron Williams. Binary bubble languages and cool-lex order. *Journal of Combinatorial Theory, Series A*, 119(1):155–169, 2012.

8 Frank Ruskey and Aaron Williams. Generating combinations by prefix shifts. In *International Computing and Combinatorics Conference*, pages 570–576. Springer, 2005.

9 Frank Ruskey and Aaron Williams. Generating balanced parentheses and binary trees by prefix shifts. In *Proceedings of the fourteenth symposium on Computing: the Australasian theory-Volume 77*, pages 107–115, 2008.

10 Frank Ruskey and Aaron Williams. The coolest way to generate combinations. *Discrete Mathematics*, 309(17):5305–5320, 2009.

11 Carla Savage. A survey of combinatorial Gray codes. *SIAM review*, 39(4):605–629, 1997.

12 Joe Sawada and Aaron Williams. Efficient oracles for generating binary bubble languages. *the electronic journal of combinatorics*, pages P42–P42, 2012.

13 Joe Sawada and Aaron Williams. A universal cycle for strings with fixed-content (which are also known as multiset permutations). In *Workshop on Algorithms and Data Structures*, pages 599–612. Springer, 2021.

14 Wladyslaw Skarbek. Generating ordered trees. *Theoretical Computer Science*, 57(1):153–159, 1988.

15 Richard P Stanley. *Catalan numbers*. Cambridge University Press, 2015.

16 Brett Stevens and Aaron Williams. The coolest way to generate binary strings. *Theory of Computing Systems*, 54(4):551–577, 2014.

17 D Roelants Van Baronaigien. A loopless algorithm for generating binary tree sequences. *Information Processing Letters*, 39(4):189–194, 1991.

18 Shmuel Zaks. Lexicographic generation of ordered trees. *Theoretical Computer Science*, 10(1):63–82, 1980.

A C Implementation

```
#include <stdio.h>
#include <stdlib.h>
typedef struct node { struct node *parent, *first, *right; } node;

void pull(node *A, node *B){
  node *child = B->first;
  B->first = child->right;
  child->right = A->first;
  A->first = child;
  child->parent = A;
}

void visit(node* root){ //prints an ordered tree as a dyck word
  node* child = root->first;
  while(child){
    putchar('1');
    visit(child);
    child = child->right;
    putchar('0');
  }
  if (!root->parent) putchar('\n');
}

void cool0tree(int n) {
  node* root = (node*)calloc(sizeof(node),1);
  node* curr = root;
  for(int i = 0; i < n; i++){
    curr->first = (node*)calloc(sizeof(node),1);
    curr->first->parent=curr;
    curr=curr->first;
  }
  pull(root,curr->parent);
  node *p, *o = root->first->right;
  visit(root);
  while (o) {
    p=o->parent;
    if (o->first) { // if o has a child, shift 1
      pull(o,p);
      o = o->first->right;
    } else {
      if (p == root) { // if the string is tight, shift a 1
        pull(o,p);
      } else { // if the string isn't tight, shift a 0
        pull(p->parent,p);
        pull(root,p);
      }
      o = o->right;
    }
    visit(root);
  }
}

int main(int argc, char **argv) { cool0tree(atoi(argv[1])); }
```