Pop & Push: Ordered Tree Iteration in $\mathcal{O}(1)$-Time

Paul Lapey
Department of Computer Science, Williams College, Williamstown, MA, USA

Aaron Williams
Department of Computer Science, Williams College, Williamstown, MA, USA

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 $\mathcal{O}(C_{n-1})$-time to generate all of the trees, and moreover, the delay between consecutive trees is worst-case $\mathcal{O}(1)$-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. When converted to Dyck words, our rule corresponds to a left-shift, and these shift 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. Our work also establishes a simultaneous Gray code for Dyck words, ordered trees, and also binary trees, using cool-lex order.

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, Gray code, simultaneous Gray code, ordered trees, plane trees, Dyck words, binary trees, Catalan objects, loopless algorithm, cool-lex order

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2022.53

Supplementary Material *Software (Source Code)*: `https://gitlab.com/combinatronics/ordered-tree-iteration` [12]

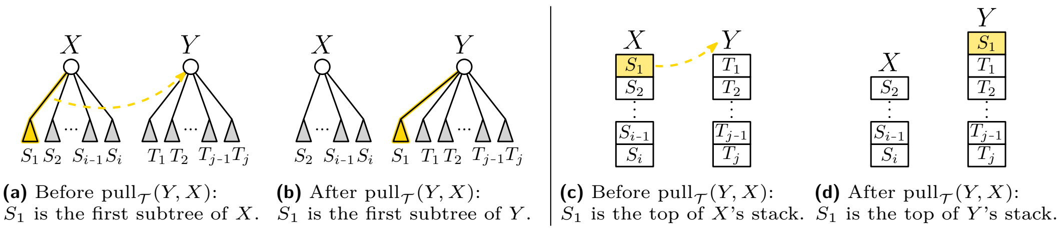
1 Introduction

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 $\mathcal{O}(1)$-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 1.

Remarkably, we are able to generate all ordered trees by only pulling subtrees that are paths. In fact, 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 2. A sample listing of all ordered trees with $n = 6$ nodes appears in Figure 6.



X
Y
S_1 S_2 S_{i-1} S_i
T_1 T_2 T_{j-1} T_j
(a) Before $\text{pull}_{\mathcal{T}}(Y, X)$: S_1 is the first subtree of X.
(b) After $\text{pull}_{\mathcal{T}}(Y, X)$: S_1 is the first subtree of Y.
(c) Before $\text{pull}_{\mathcal{T}}(Y, X)$: S_1 is the top of X's stack.
(d) After $\text{pull}_{\mathcal{T}}(Y, X)$: S_1 is the top of Y's stack.

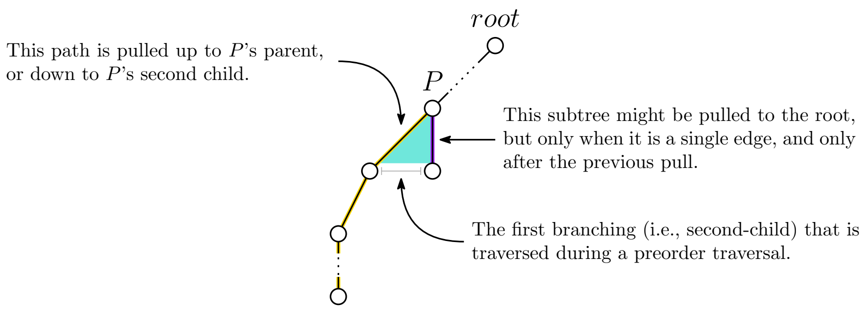
Figure 1 The pull operation relocates first subtrees in an ordered tree $\mathcal{T}$. More specifically, if X and Y are nodes of $\mathcal{T}$, then $\text{pull}_{\mathcal{T}}(Y, X)$ removes the first subtree of X and adds it as a new first subtree of Y; see (a)–(b). In other words, Y pulls X's first subtree so that it becomes its own first subtree. The operation is particularly natural when each node stores its subtrees using a stack, since the pull simply "pops" X's stack, and "pushes" the result onto Y's stack; see (c)–(d).

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, half of which are used for initialization and output. Our main result is summarized by the following theorem. Figure 7 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 [8] – meaning that the objects are ordered so that the next object always differs from the previous object in a constant amount according to some simple metric. More descriptively, our order is a *2-pull Gray code*, since two pulls are used in the worst-case [1] . Theorem 1 represents the first 2-pull Gray code for ordered trees.

[1] The pulled subtrees are always paths, so we could also describe our result as a *2-path-pull Gray code*.



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 (i.e., second-child) that is traversed during a preorder traversal.

Figure 2 Our Gray code for ordered trees is based on pulls. More specifically, the next ordered tree is always created by one or two pulls. Furthermore, 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 (6) and Figure 5. Figure 7 has loopless (i.e., worst-case $\mathcal{O}(1)$-time per tree) algorithm implementations in pseudocode and C, with full C implementations (including command-line argument processing) in the Appendices.

Algorithm O is also the first loopless algorithm for generating ordered trees with a linked-based representation in which all of the details are provided. The secret to our algorithm is ordering Dyck words using a variant of co-lexicographic order known as cool-lex order.

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 this research area, we recommend Knuth's coverage of generating combinatorial objects in Volume 4A of *The Art of Computer Programming* [10], and Mütze's recent update [16] of Savage's well-known survey [27]. Many practical programs for generating combinatorial objects can also be found in Arndt [2], and on the *Combinatorial Object Server* [17]. Readers are also directed to a new preprint by Nakano [18] that provides a nice alternative Gray code for ordered trees: successive trees are obtained by removing a leaf and appending it to another node.

1.1.1 Catalan Objects

Ordered trees are a *Catalan object* since they are enumerated by the Catalan numbers. More specifically, the number of ordered trees with n nodes is the $(n-1)$st Catalan number C_{n-1}. Other well-known Catalan objects include Dyck words and binary trees. A *Dyck word of order* n is a binary string with n copies of 1 and n copies of 0, where each prefix has at least as many 1s as 0s. These strings have a trivial correspondence with balanced parentheses of length $2n$: map 1s to (s and 0s to)s. These sets of strings are shown in (1) for $n=3$.

$$\{101010, 101100, 110010, 110100, 111000\} \qquad \{()()(), ()(()), (())(), (()()), ((()))\} \tag{1}$$

Dyck words of order n are counted by the nth Catalan number C_n, as are binary trees with n nodes. Many previous results have focused on ordering and generating these objects using different operations, with a very well-known example being binary trees by rotations [14].

1.1.2 Loopless vs Constant Amortized Time

Skarbek [30] provided the first algorithm for generating link-based representations of ordered trees in *constant amortized time (CAT)*, meaning that the delay is $\mathcal{O}(1)$-time in an amortized sense, rather than in the worst-case as in a loopless algorithm. Korsh and Lafolette [11] 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, and answered an open problem posed by Roelants van Baronaigien [34]. 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.

Theoretically speaking, loopless algorithms are more significant achievements than CAT algorithms. In particular, standard lexicographic orders can often be generated in CAT [21], and this has been accomplished for ordered trees on multiple occasions [7, 19]. However, loopless algorithms are often less practical than their CAT counterparts for several reasons.

- Efficiency. Loopless algorithms typically involve more instructions than CAT algorithms, and their overall run-times are often longer.
- Ease of implementation. Generation algorithms are often ported from one programming language (or pseudocode) to another, so longer implementations are again less desirable.

- Novelty. Most loopless algorithms involve the creation of a novel ordering of the underlying objects, which can hinder debugging and efficient ranking/unranking (see Section 1.1.3).

For example, [11] creates a novel order, and its provided C code includes over 100 instances of the `if` and `else` keywords. In contrast, we generate a familiar variant of lexicographic order and our C implementation requires only 4 instances of `if` and `else`; see [12] or the Appendix.

1.1.3 Simultaneous Gray Codes

Suppose that X and Y are sets of combinatorial objects that are in one-to-one correspondence according to bijection $f : X \to Y$. If $x_1, x_2, \ldots, x_m$ is a Gray code for X, then we may ask whether the corresponding order for the objects in Y, namely $f(x_1), f(x_2), \ldots, f(x_m)$, is also a Gray code. If this is true, then we refer to the order as a *simultaneous Gray code* for X and Y according to f. As a trivial example, let X_n be the set of n-bit binary strings, and Y_n be the set of subsets of $[n] = \{1, 2, \ldots, n\}$, with $f : X_n \to Y_n$ mapping each incidence vector $b_1 b_2 \cdots b_n$ to its subset. In this case, the binary reflected Gray code provides an ordering of X_n in which successive strings differ in a single bit, and the corresponding subsets in Y_n differ by adding or removing one element, as seen for $n = 3$ in (2) and (3), respectively.

$$000 \quad 001 \quad 011 \quad 010 \quad 110 \quad 111 \quad 101 \quad 100 \tag{2}$$

$$\emptyset \quad \{3\} \quad \{2,3\} \quad \{2\} \quad \{1,2\} \quad \{1,2,3\} \quad \{1,3\} \quad \{1\} \tag{3}$$

The literature features relatively few non-trivial simultaneous Gray codes[2]. This scarcity is in part due to the difficulty in finding such orders. To illustrate this important point, let us consider the goal of constructing a simultaneous Gray code for Dyck words and ordered trees. During this discussion, we'll use the standard mapping of an ordered tree $\mathcal{T}$ with $n-1$ nodes to a Dyck word of order n found in Stanley's *Catalan Addendum* [31]:

Conduct a preorder traversal of the ordered tree $\mathcal{T}$, and record 1 *when going down an edge, and* 0 *when going up an edge.*

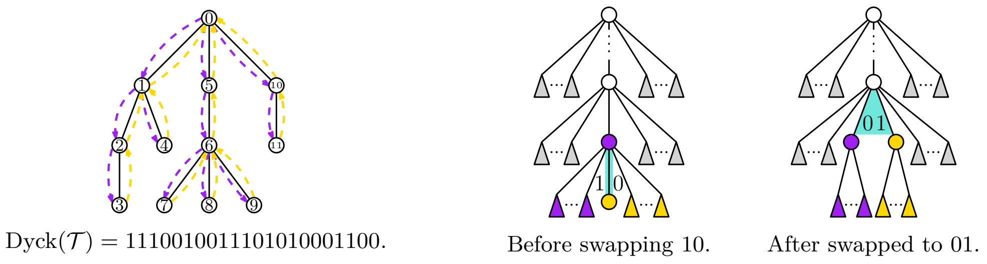
Figure 3a illustrates this mapping, where Dyck($\mathcal{T}$) is the Dyck word corresponding to $\mathcal{T}$.

One of the simplest operations on strings, including Dyck words, is an *adjacent-transposition*, which replaces a substring xy with yx. An adjacent-transposition is also known as a *swap*. Figure 3b illustrates how a swap in a Dyck word affects the corresponding ordered tree.

To understand Figure 3b, first note that a 10 substring in a Dyck word corresponds to a leaf in an ordered tree. Swapping 10 to 01 causes the leaf (shown in gold) to become the next sibling of its parent (purple) with its rightward siblings (gold) becoming its children. As a result, this small change in a Dyck word does not correspond to a constant amount of work in the ordered trees, so long as the nodes have `parent` pointers. Hence, a swap Gray code for Dyck words won't provide a simultaneous Gray codes for ordered trees, unless (a) a non-standard bijection is used, (b) `parent` pointers are omitted, or (c) the swaps used can be further constrained to change fewer `parent` pointers. There is another complication with this proposed approach: Swap Gray codes don't always exist for Dyck words[3] [20, 22].

[2] A recent exception involves Baxter permutations and mosaic floorplans that arise in VLSI design [38]. A Gray code for Baxter permutations was found by generalizing the greedy interpretation of plain change order [36], and it provides a simultaneous Gray code for mosaic floorplans according to the bijection in [1]. More broadly, Algorithm J [9] generates a jump Gray code for various types of pattern-avoiding permutations and simultaneously orders various types of rectangulations by flips and wall-slides [15].

[3] A 2-swap Gray code of Dyck words does exist by Vajnovszki and Walsh [33]. Their order uses swaps



(a) During a preorder traversal of $\mathcal{T}$, record 1 when going down an edge (purple) and 0 when going up an edge (gold) to obtain its corresponding Dyck word.

(b) A swap in a Dyck word can change an arbitrary number of `parent` pointers. Specifically, the root of each gold subtree has a new parent.

Figure 3 With the standard bijection between Dyck words and ordered trees in (a), a swap in a Dyck word can cause a non-constant amount of change in the corresponding ordered tree as shown in (b). This illustrates the difficulty in constructing simultaneous Gray codes.

Cool-lex order is a variation of co-lexicographic order that was first introduced for (s, t)-combinations by Ruskey and Williams [25] in the latter's PhD thesis [37]. Since then it has been used to create Gray codes and efficient algorithms for a variety of combinatorial objects [35, 23, 28, 24, 5, 3], with recent examples including multiset necklaces [29] and fixed-content Łukasiewicz words [13]. In particular, the application of cool-lex order to Dyck words [25] is relevant later in the article. Besides its familiarity, cool-lex order also provides efficient *ranking* and *unranking* of Dyck words [25] (and k-ary [6] and $\frac{1}{k}$-ary Dyck words [5]), which means that the position of any word can be determined, as can the word in any position, using only $\mathcal{O}(n)$ arithmetic operations. Hence, our Gray code of ordered trees can be ranked and unranked just as efficiently, since the standard bijection runs in linear-time.

1.2 Outline

Section 2 discusses ordered trees in more detail. Section 3 provides our Gray code, and Section 4 generates it efficiently. Correctness is proven in Section 5 and Section 6 has final remarks. C implementations with or without `parent` pointers are in the appendices and [12].

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.

and *two-close transpositions*, which replace a substring 100 with 001 (or vice versa) and which can be realized using two swaps. This would be a good candidate for simultaneous Gray code, so long as parent pointers could be avoided. Its loopless pseudocode is relatively compact, containing only 17 `if` and `else` keywords, but this is still significantly more complicated than our result.

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 6. 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.

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, o.first.right, 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 `o` has `o.num == 0` and `o.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 $\mathrm{pop}_{\mathcal{T}}(A)$ 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 $\mathrm{pop}_{\mathcal{T}}(A)$.

If $\mathcal{S}$ is a non-empty ordered tree, then let $\mathrm{push}_{\mathcal{T}}(A, \mathcal{S})$ 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 $\mathrm{push}_{\mathcal{T}}(A, \mathcal{S})$.

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

$$\mathrm{pull}_{\mathcal{T}}(A, B) \text{ is } \mathrm{push}_{\mathcal{T}'}(A, \mathrm{pop}_{\mathcal{T}}(B)) \text{ where } \mathcal{T}' \text{ is the modification of } \mathcal{T} \text{ after popping.} \quad (4)$$

We think of $\mathrm{pull}_{\mathcal{T}}(A, B)$ 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 B to A that shows how the subtree is pulled toward A.

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.

$$\mathrm{pull}_{\mathcal{T}}(A; C, B) \text{ is } \mathrm{pull}_{\mathcal{T}}(A, B) \text{ then } \mathrm{pull}_{\mathcal{T}'}(C, B), \text{ where } \mathcal{T}' \text{ is created by the first pull.} \quad (5)$$

```
▷ $A$ 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 = B.first;
  B.first = temp.right;
  temp.right = A.first;
  A.first = temp;
  temp.parent = A;
}
```

Figure 4 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.

In other words, A and C both pull B: B's first subtree becomes A's new first subtree, then B's new first subtree becomes C's new first subtree. When illustrating the double pull in Figure 5, we use gold for the first pull (from G), and purple for the second (from *root*).

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 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

This section defines the order in which we generate ordered trees with n nodes. The order is built one tree at a time by a successor rule whose origin is based on Theorem 5 in Section 5.

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 6.

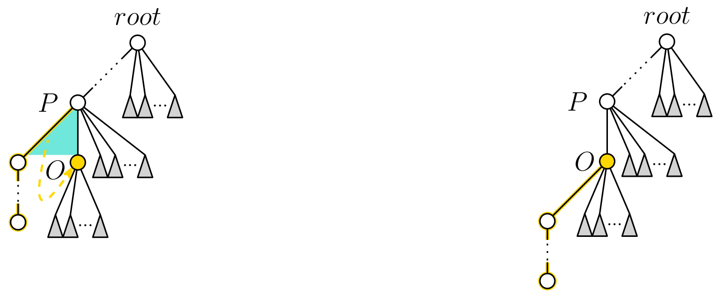
The rest of the order is generated by a successor rule, which transform any ordered tree – other than $\mathcal{T}_0$ – into the next ordered tree.

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 G is only well-defined when the parent P is not the *root* of the tree $\mathcal{T}$. Given these nodes, the successor rule can now be stated.

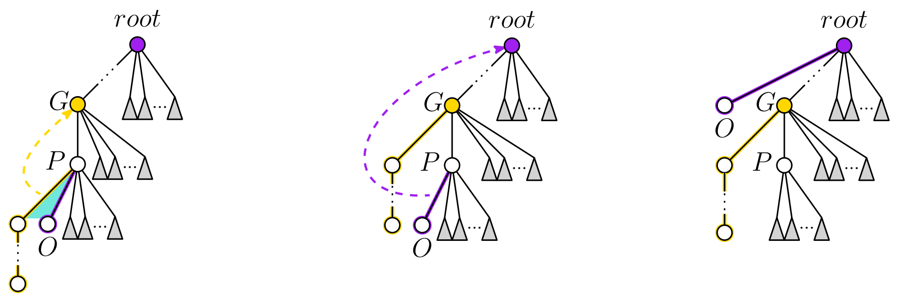
$$\text{next}(\mathcal{T}) = \begin{cases} \text{pull}_{\mathcal{T}}(O, P) & \text{if } P = \mathit{root} \text{ or } O \text{ has a child} \quad (6c) \\ \text{pull}_{\mathcal{T}}(G; \mathit{root}, P) & \text{otherwise (i.e. } P \neq \mathit{root} \text{ and } O \text{ is a leaf)} \quad (6d) \end{cases}$$

Figure 5 illustrates the pulls used in (6), while Figure 6 illustrates the full order for $n = 6$.



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

(a) Case 6c. After the pull, the new value of 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 push$(O, \text{pop}(P))$.



Current tree $\mathcal{T}$ where $P \neq \mathit{root}$ 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 $\text{pull}_{\mathcal{I}}(\mathit{root}, P)$.

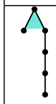
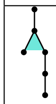
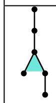
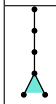
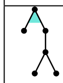
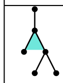
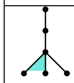
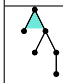
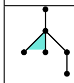
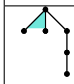
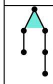
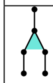
(b) Case 6d. After the pull, the new value of node O is updated to be the new second-child of *root*. The pull operations are equivalent to push$(G, \text{pop}(P))$ followed by push$(\mathit{root}, \text{pop}(P))$.

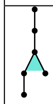
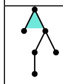
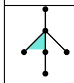
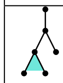
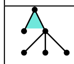
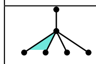
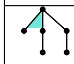
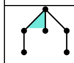
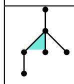
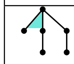
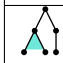
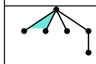
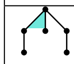
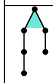
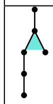
Figure 5 Our pull Gray code is generated by the two case successor rule in (6), which transforms the current ordered tree $\mathcal{T}$ into the new next ordered tree $\mathcal{T}'$ using one or two pulls. In these figures, O is the first node in a preorder traversal that is not on the path from the 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*.

Algorithm O first creates an initial tree $\mathcal{T}_1$. Then it repeatedly applies the successor rule in (6). To apply the rule, the algorithm tracks node O, with Figure 5 illustrating how its location changes after each application of (6). Given O's location, the pointer updates can be easily inferred from Figure 5. The 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.

Tree	Dyck word	Next
	1011110000	(6c)
	1101110000	(6c)
	1110110000	(6c)
	1111010000	(6d)
	1011101000	(6c)
	1101101000	(6c)
	1110101000	(6d)
	1011011000	(6c)
	1101011000	(6d)
	1010111000	(6c)
	1100111000	(6c)
	1110011000	(6c)

Tree	Dyck word	Next
	1111001000	(6d)
	1011100100	(6c)
	1101100100	(6c)
	1110100100	(6d)
	1011010100	(6c)
	1101010100	(6d)
	1010110100	(6c)
	1100110100	(6c)
	1110010100	(6d)
	1011001100	(6c)
	1101001100	(6d)
	1010101100	(6c)
	1100101100	(6c)
	1110010100	(6c)
	1111000100	(6d)

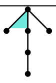
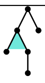
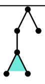
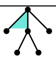
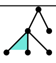
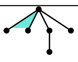
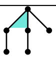
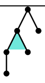
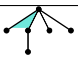
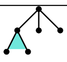
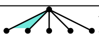
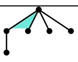
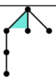
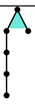
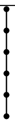
Tree	Dyck word	Next
	1011100010	(6c)
	1101100010	(6c)
	1110100010	(6d)
	1011010010	(6c)
	1101010010	(6d)
	1010110010	(6c)
	1100110010	(6c)
	1110010010	(6d)
	1011001010	(6c)
	1101001010	(6d)
	1010101010	(6c)
	1100101010	(6c)
	1110001010	(6c)
	1111000010	(6c)
	1111100000	

Figure 6 The ordered trees with $n = 6$ nodes as generated by our algorithm. The next tree is obtained by $\mathrm{pull}(O, P)$ when (6c) is specified, or by $\mathrm{pull}(G; root, P)$ when (6d) 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. Each tree is displayed beside its corresponding Dyck word. Each Dyck word can be transformed into its successor by left-shifting the highlighted bit into the second position, as discussed in Theorem 5.

Figure 7 gives the algorithm in pseudocode and C. A complete C implementation (with arguments parsing, initialization, and visit routines) can be found in the appendix or [12].

5 Connection to Cool-lex Order for Dyck Words

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 showing

```
function COOL-ORDERED-TREES($n$)
    ▷ Generate initial tree
    $O \leftarrow root.first.right$
    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($P.parent, P$)
                pull($root, P$)
            $O \leftarrow O.right$
        visit(root)
```

(a) Generate all ordered trees with $n+1$ 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
      pull(o,p);
      o = o->first->right;
    } else {
      if (p == root) { // if o's parent
                       // is root
        pull(o,p);
      } else {
        pull(p->parent,p);
        pull(root,p);
      }
      o = o->right;
    }
    visit(root);
  }
}
```

(b) C implementation of coolOtree.

Figure 7 Algorithm O is presented in pseudocode in (a), and its main function in C is in (b).

how certain pull operations in ordered trees translate to changes in the associated Dyck words in Section 5.1. In Section 5.2, we recall the successor rule for generating Dyck words in cool-lex order from [25]. Finally, in Section 5.3, 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 Pulls in Ordered Trees and Shifts in Dyck Words

Now we attempt to understand how the path-pulls in (6) 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 (6) and their depictions in Figure 8. Our first lemma is associated with the single pull used in (6c).

▶ **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*

$$\text{Dyck}(\mathcal{T}) = 1^p0^q1\alpha \text{ and } \text{Dyck}(\mathcal{T}') = 11^p0^q\alpha. \tag{7}$$

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

Proof. From Figure 8a, we can see that $\text{pull}_{\mathcal{T}}(O,P)$ transforms $\mathcal{T}$ into $\mathcal{T}'$ with

$$\text{Dyck}(\mathcal{T}) = 1^a11^b0^b01\alpha \text{ and } \text{Dyck}(\mathcal{T}') = 1^a111^b0^b0\alpha.$$

Therefore, the result follows by setting $p = a+b+1$ and $q = b+1$. This is because $\text{Dyck}(\mathcal{T}) = 1^a11^b0^b01\alpha = 1^p0^q1\alpha$ and $\text{Dyck}(\mathcal{T}') = 1^a111^b0^b0\alpha = 11^p0^q\alpha$. ◀

The second lemma considers the double pull used in (6d).

▶ **Lemma 4.** *Suppose that* (6d) *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* $\mathrm{pull}_{\mathcal{T}}(G, P)$*, and* $\mathcal{T}'$ *is the result of* $\mathrm{pull}_{\mathcal{I}}(root, P)$*. Then there exists* $p < q$ *and an* $\alpha \in \{0,1\}^*$ *in which*

$$\mathrm{Dyck}(\mathcal{T}) = 1^p0^q10\alpha \textit{ and } \mathrm{Dyck}(\mathcal{T}') = 101^{p-1}0^q1\alpha. \tag{8}$$

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

Proof. From Figure 8b, we can see that $\mathrm{pull}_{\mathcal{T}}(G, P)$ transforms $\mathcal{T}$ into $\mathcal{I}$ with

$$\mathrm{Dyck}(\mathcal{T}) = 1^a111^b0^b010\alpha \text{ and } \mathrm{Dyck}(\mathcal{I}) = 1^a11^b0^b0110\alpha.$$

Then Figure 8c, shows that $\mathrm{pull}_{\mathcal{I}}(root, P)$ transforms $\mathcal{I}$ into $\mathcal{T}'$ with

$$\mathrm{Dyck}(\mathcal{I}) = 1^a11^b0^b0110\alpha \text{ and } \mathrm{Dyck}(\mathcal{T}') = 101^a11^b0^b01\alpha.$$

Therefore, the result follows by setting $p = a + b + 2$ and $q = b + 1$. This is because $\mathrm{Dyck}(\mathcal{T}) = 1^a111^b0^b010\alpha = 1^p0^q10\alpha$ and $\mathrm{Dyck}(\mathcal{T}') = 101^a11^b0^b01\alpha = 101^{p-1}0^q1\alpha$. ◀

5.2 Cool-lex Order for Dyck Words

Dyck words of order n can be generated by a simple successor rule [25]. The resulting order can be understood as a sublist of the cool-lex order of (n, n)-combinations [26] or binary strings of length $2n$ [32], and as a special case of the order for bubble languages [23] or fixed-content Łukasiewicz words [13] both of which can be generated efficiently [28, 13]. However, these string results have no immediate implications for generating ordered trees.

▶ **Theorem 5** ([25])**.** *Dyck words of order* n *are generated in cool-lex order starting from* $101^{n-1}0^{n-1}$ *by the following successor rule, where* $\alpha \in \{0,1\}^*$ *denotes an unchanged suffix.*

(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*.*

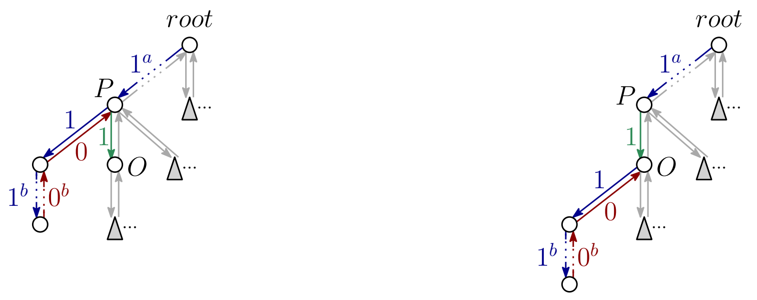
The only string that is not covered by one of these cases is the last Dyck word, 1^n0^n*.*

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

5.3 Proof of Correctness

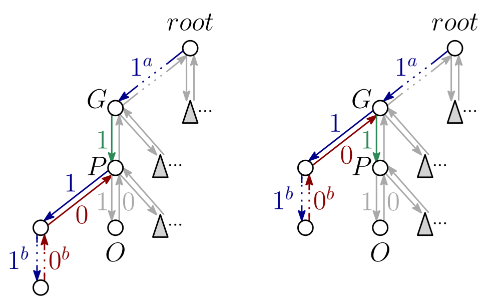
Proof of Theorem 1. It is clear that Algorithm O in Figure 7 is loopless and correctly implements the order defined in Section 3. What remains to be proven is that the algorithm actually generates all of the ordered trees with n nodes. The proof equates the successor (6) for ordered trees into the cool-lex rule for Dyck words of order $n-1$.

Observe that the first ordered tree $\mathcal{T}_1$ has $\mathrm{Dyck}(\mathcal{T}_1) = 101^{n-2}0^{n-2}$, which matches the first string in the cool-lex order of Dyck words. Similarly, the last ordered tree $\mathcal{T}_0$ has $\mathrm{Dyck}(\mathcal{T}_0) = 1^{n-1}0^{n-1}$, which matches the last string. It remains to be proven that the successor rule for ordered trees in (6) matches the cool-lex successor rule for Dyck words. Fortunately, this follows immediately from Lemmas 3–4 and Theorem 5. ◀

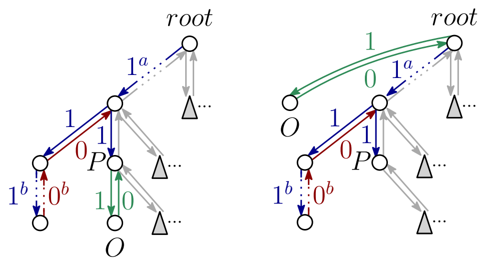


Ordered tree $\mathcal{T}$ with Dyck($\mathcal{T}$) = $1^a11^b0^b01\alpha$. Ordered tree $\mathcal{T}'$ with Dyck($\mathcal{T}'$) = $1^a111^b0^b0\alpha$.

(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 Dyck($\mathcal{T}$) = $1^a111^b0^b0\alpha$. Ordered tree $\mathcal{I}$ with Dyck($\mathcal{I}$) = $1^a11^b0^b01\alpha$.



Ordered tree $\mathcal{I}$ with Dyck($\mathcal{I}$) = $1^a11^b0^b0110\alpha$. Ordered tree $\mathcal{T}'$ with Dyck($\mathcal{T}'$) = $101^a11^b0^b01\alpha$.

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

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

Figure 8 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 (6c) and Figure 5a, and the latter two mirroring the double pulls in (6d) and Figure 5b, 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.

6 Final Remarks

In this article, we provided a 2-pull Gray code for ordered trees and a loopless algorithm for generating it. Our work also completes a simultaneous Gray code for Dyck words, ordered trees, and binary trees [25], which are three of the foremost Catalan structures.

6.1 Additional Results

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 the end of the array.

The cool-lex order of Dyck words was also used to create a loopless algorithm for binary trees in [25]. By storing 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

We showed that two pulls are sufficient for listing ordered trees, even when the pulled subtrees are always paths. This raises two follow-up questions.

- Is there a 1-pull Gray code for ordered trees?
- Is there a 1-path-pull Gray code for ordered trees?

Ideally, a positive answer would also be accompanied by an efficient algorithm.

Finally, we mention that generalizations of Catalan objects were recently ordered and generated by Cardinal, Merino and Mütze [4] using Algorithm J. This opens the door to considering further generalizations of the results found in this article.

References

1 Eyal Ackerman, Gill Barequet, and Ron Y Pinter. A bijection between permutations and floorplans, and its applications. *Discrete Applied Mathematics*, 154(12):1674–1684, 2006.

2 Jörg Arndt. *Matters Computational: ideas, algorithms, source code.* Springer Science & Business Media, 2010.

3 Péter Burcsi, Gabriele Fici, Zsuzsanna Lipták, Frank Ruskey, and Joe Sawada. On combinatorial generation of prefix normal words. In *Symposium on Combinatorial Pattern Matching*, pages 60–69. Springer, 2014.

4 Jean Cardinal, Arturo Merino, and Torsten Mütze. Efficient generation of elimination trees and graph associahedra. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2128–2140. SIAM, 2022.

5 Stephane Durocher, Pak Ching Li, Debajyoti Mondal, Frank Ruskey, and Aaron Williams. Cool-lex order and k-ary Catalan structures. *Journal of Discrete Algorithms*, 16:287–307, 2012.

6 Stephane Durocher, Pak Ching Li, Debajyoti Mondal, and Aaron Williams. Ranking and loopless generation of k-ary Dyck words in cool-lex order. In *International Workshop on Combinatorial Algorithms*, pages 182–194. Springer, 2011.

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

8 Frank Gray. Pulse code communication. *United States Patent Number 2632058*, 1953.

9 Elizabeth Hartung, Hung Hoang, Torsten Mütze, and Aaron Williams. Combinatorial generation via permutation languages. i. fundamentals. *Transactions of the American Mathematical Society*, 375(04):2255–2291, 2022.

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

11 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.

12 Paul W Lapey and Aaron Williams. Ordered tree iteration, 2022. URL: `https://gitlab.com/combinatronics/ordered-tree-iteration`.

13 Paul W Lapey and Aaron Williams. A shift Gray code for fixed-content Łukasiewicz words. In *International Workshop on Combinatorial Algorithms*, pages 383–397. Springer, 2022.

14 Joan M Lucas, Dominique Roelants van Baronaigien, and Frank Ruskey. On rotations and the generation of binary trees. *Journal of Algorithms*, 15(3):343–366, 1993.

15 Arturo Merino and Torsten Mütze. Efficient generation of rectangulations via permutation languages. In *37th International Symposium on Computational Geometry (SoCG 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

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

17 Torsten Mütze. Cos++. the combinatorial object server, 2022. URL: `http://combos.org`.

18 Shin-ichi Nakano. A gray code of ordered trees. *arXiv preprint*, 2022. `arXiv:2207.01129`.

19 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.

20 Andrzej Proskurowski and Frank Ruskey. Binary tree Gray codes. *Journal of Algorithms*, 6(2):225–238, 1985.

21 Frank Ruskey. Combinatorial generation. *Preliminary working draft. University of Victoria, Victoria, BC, Canada*, 11:20, 2003.

22 Frank Ruskey and Andrzej Proskurowski. Generating binary trees by transpositions. *Journal of Algorithms*, 11(1):68–84, 1990.

23 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.

24 Frank Ruskey, Joe Sawada, and Aaron Williams. De Bruijn sequences for fixed-weight binary strings. *SIAM Journal on Discrete Mathematics*, 26(2):605–617, 2012.

25 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.

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

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

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

29 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.

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

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

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

33 Vincent Vajnovszki and Timothy Walsh. A loop-free two-close Gray-code algorithm for listing k-ary Dyck words. *Journal of Discrete Algorithms*, 4(4):633–648, 2006.

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

35 Aaron Williams. Loopless generation of multiset permutations by prefix shifts. In *SODA 2009, Symposium on Discrete Algorithms*, 2009.

36 Aaron Williams. The greedy gray code algorithm. In *Workshop on Algorithms and Data Structures*, pages 525–536. Springer, 2013.

37 Aaron Michael Williams. *Shift Gray codes.* PhD thesis, University of Victoria, 2009.

38 Bo Yao, Hongyu Chen, Chung-Kuan Cheng, and Ronald Graham. Floorplan representations: Complexity and connections. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 8(1):55–80, 2003.

A C Implementation – With Parent Pointer

```c
#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, o pulls p
      pull(o,p);
      o = o->first->right;
    } else {
      if (p == root) { // if p is the root, o pulls p
        pull(o,p);
      } else { // otherwise, g pulls p and root pulls p
        pull(p->parent,p);
        pull(root,p);
      }
      o = o->right;
    }
    visit(root);
  }
}

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

B C Implementation – Without Parent Pointer

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

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

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

void coolOtree(int n) {
  node *root = (node*)calloc(sizeof(node),1);
  node *prev, *curr = root;
  for(int i = 0; i < n; i++){
    curr->first = (node*)calloc(sizeof(node),1);
    prev=curr;
    curr=curr->first;
  }
  pull(root,prev);
  node *p=root, *g=NULL, *o = root->first->right;
  visit(root);
  while (o) {
    if (o->first) { // if o has a child, o pulls p
      pull(o,p);
      g=p;
      p=o;
      o = o->first->right;
    } else {
      if (p == root) { // if p is the root, o pulls p
        pull(o,p);
      } else { // otherwise, g pulls p and root pulls p
        pull(g,p);
        pull(root,p);
        p=root; //g could be set to null
      }
      o = o->right;
    }
    visit(root);
  }
}
int main(int argc, char **argv) { coolOtree(atoi(argv[1])); }
```