Cooler than Cool:
Cool-Lex Order for Generating New Combinatorial Objects

Paul Lapey

March 27, 2022

Chapter 1

Introduction

1.1 Combinatorial Generation: Let's Look at all the Possibilities

Combinatorial generation is defined as the exhaustive listing of combinatorial objects of various types. Frank Ruskey duly notes in his book *Combinatorial Generation* that the phrase "Let's look at all the possibilities" sums up the outlook of his book and the field as a whole [Rus03]. Examining all possibilities fitting certain criteria is frequently necessary in fields ranging from mathematics to chemistry to operations research. Combinatorial generation as an area of study seeks to find an underlying combinatorial structure to these possibilities and utilize it to obtain an algorithm to efficiently enumerate an appropriate representation of them [Rus03].

A quintessential result of the combinatorial generation in practice is Frank Gray's reflected binary code, or Gray code. Gray codes give a "reflected" ordering of binary strings such that each successive string in the ordering differs from the previous string by exactly one bit. This is notably different from a lexicographic ordering of binary strings, in which a n-digit binary string can differ by up to n digits from its predecessor and will differ by approximately two (more precisely $\sum_{i=0}^{n} 2^i$, which is 1.9375 for 4 bit values and 1.996 for 8 bit values) bits on average[1]. The binary reflected Gray code, therefore, provides an ordering that requires as many bit switches as the more intuitive lexicographic order. Binary reflected Gray codes are widely used in electromechanical switches to reduce error and prevent spurious output associated with asynchronous bit switches. Crucially, Frank Gray's reflected binary code achieved a tangible benefit in error reduction through the use of an alternative method of enumerating binary strings. The technique of reflecting all or certain parts of a string to generate new strings has become one of the most widely used techniques in combinatorial generation.

[1] Consecutive pairs of binary digits in lexicographic order will differ in the bit at position i with probability $\frac{1}{2^i}$. Therefore, the average number of differing bits between two binary strings of length n is $\sum_{i=0}^{n} 2^i$, which converges to 2 as n grows large.

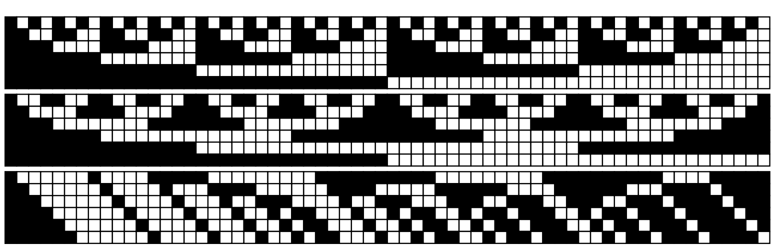


Figure 1.1: Lexicographic (top), binary reflected Gray code (middle), and cool-lex (bottom) enumerations of 6-bit binary strings. Individual strings are read vertically from top to bottom.

1.2 Cool-Lex Order

More recently, cool-lex order has introduced the idea of rotating sublists to enumerate languages. Different versions of cool-lex order have been shown to enumerate several sets of combinatorial objects, including binary strings, fixed weight binary strings, Dyck words, and multiset permutations. Cool-lex orders often lead to algorithms that are faster and simpler than standard lexicographic order. For example, the "multicool" package in R uses a loopless cool-lex algorithm to efficiently enumerate multiset permutations. The package started using cool-lex order for multiset permutations in versoin 1.1 and as of version 1.12 has been downloaded nearly a million times [CWKB21].

1.3 Goals of this Thesis

Cool-lex has been shown to provide a minimal-change cyclic ordering for the sets of fixed-weight binary strings, multiset permutations, binary and k-ary Dyck words, and other languages [Wil09b]. A common thread in the cool-lex algorithms for combinatorial generation is their focus on the *first increase* of string, or the longest prefix of a string such that each successive symbol in the prefix is less than or equal to the previous symbol in the string.

This thesis will examine the use of cool-lex orders to enumerate other languages. Among these are Lukasiewicz, Motzkin, and Schröder paths, which are lattice paths that share similarities with Dyck paths. Shift Gray codes for enumerating these languages have been developed and are given in 1.3.

Dyck, Motzkin, Schröder, and Lukasiewicz paths all share bijections with various combinatorial objects. For example, Dyck paths of length $2n$ share a bijection with binary trees with n nodes. Ruskey and Williams found that the cool-lex successor rule for enumerating Dyck words corresponded directly to a loopless successor rule for enumerating binary trees with a constant number of pointer changes [RW08]. This thesis will examine the efficiency of using cool-lex order to enumerate other sets of combinatorial objects in bijective correspondence with these languages.

TODO: Describe new results of thesis?

Chapter 2

Background

2.1 Dyck Words

The language of binary Dyck words is the set of sequences of binary digits that satisfy the following conditions: The sequence has an equal number of ones and zeroes and there is no prefix of the sequence in which the number of zeroes exceeds the number of ones. The Dyck language can equivalently be thought of as the set of balanced parentheses, with ones representing open parentheses and zeroes representing closing parentheses. In addition to balanced parentheses, Dyck words of length $2n$ are also in bijective correspondence with extended binary trees with n internal nodes. Given an extended binary tree B with n internal nodes, a Dyck word can be obtained by traversing B in preorder and recording each internal node as a 1 and each leaf with a 0, ignoring the final leaf of the tree.

2.1.1 Generalizations of Dyck words: Motzkin, Schröder, and Łukasiewicz paths

Motzkin, Schröder, and Łukasiewicz paths provide generalizations of Dyck words.

In addition to representing balanced parentheses, Dyck paths can be thought of as paths on a cartesian plane. Dyck paths are paths from $(0,0)$ to $(2n,0)$ that use $2n$ steps of either $(1,1)$ (northeast) or $(1,-1)$ (southeast) and never cross below the x axis. In the binary string representation of Dyck words, ones correspond to $(1,1)$ steps and zerores correspond to $(1,-1)$ steps.

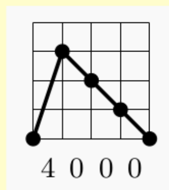
Motzkin paths allow for (1,0) horizontal steps in addition to (1,1) and (1,-1) steps. Schröder paths are identical to Motzkin paths except they allow for $(2,0)$ horizontal steps instead of $(1,0)$. Łukasiewicz paths allow (1,-1) steps, (1,0) steps and any (1,k) step where k is a positive integer. All three languages retain the requirement that the path start at the origin, end on the x axis, and never step below the x axis.

These paths can be encoded in a number of different ways. In a *-1-based encoding*, each $(1,i)$ step is encoded as i, and every prefix must have a nonnegative sum. In a *0-based encoding*, each $(1,i)$ step is encoded as $i+1$, and the sum of every prefix must be as large as its length. We primarily use the 0-based encoding. See Fig. 2.1 for examples of these paths using the 0-based encoding.

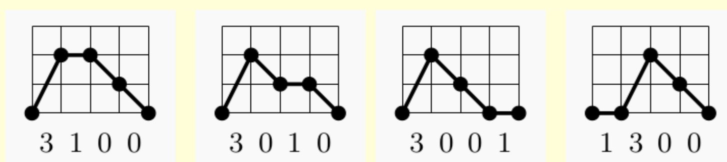
We refer to Motzkin, Schröder, and Lukasiewicz paths ending at $(n,0)$ as paths of *order n*. This contrasts slightly with the classification of Dyck words of order n, which terminate at $(2n,0)$

In the context of fixed-content generation, Motzkin and Schröder paths are identical: Both will have northeast steps encoded as twos, horizontal steps encoded as ones, and southeast steps encoded as zeroes. However, their graphical representations Notably, Łukasiewicz are a generalization of Motzkin and Schröder paths, as any Motzkin or Schröder path is also a Lukasiewicz path.

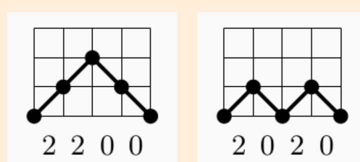
The number of Dyck words with n zeroes and n ones are counted by the nth Catalan number. Similarly, the number of Motzkin and Schröder paths of order n are counted by the nth Motzkin and big Schröder number respectively. The number of Lukasiewicz paths of order n are counted by the n Motzkin, Schröder, and Lukasiewicz paths bear a number of interesting bijective correspondences with other combinatorial objects. Richard Stanely's *Catalan Objects* outlines hundreds of interesting examples.



The Łukasiewicz paths with content $\{0, 0, 0, 4\}$ using 0-based strings.
Note: The content is $\{\text{-1}, \text{-1}, \text{-1}, 3\}$ when using (-1)-based meander strings,
or $\{0, 0, 1\}$ for 3-ary Dyck words.



The Łukasiewicz paths with content $\{0, 0, 1, 3\}$.



The Łukasiewicz / Schröder / Motzkin / Dyck paths with content $\{0, 0, 2, 2\}$.
Note: The content is $\{[,\ [,\ ],\ ]\}$ for Dyck words or $\{0, 0, 1, 1\}$ for 2-ary Dyck words.

Figure 2.1

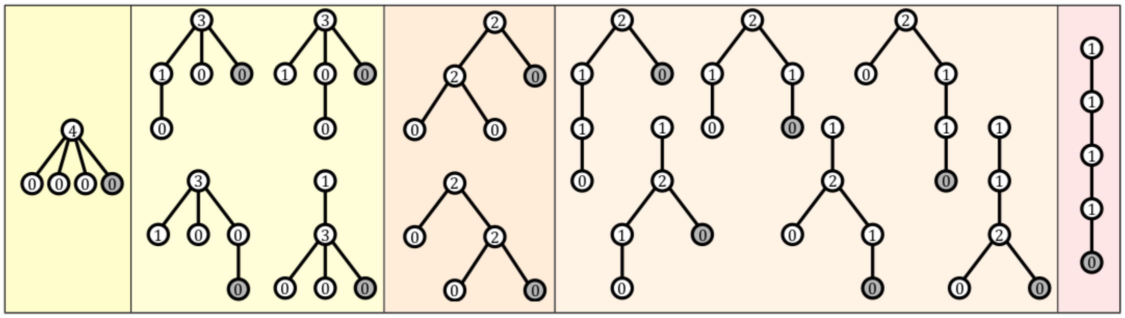


Figure 2.2: The $\mathcal{C}_4$=14 Lukasiewicz paths of order $n = 4$ are in bijective correspondence with the 14 rooted ordered trees with $n+1 = 5$ nodes. Given a tree, the corresponding word is obtained by recording the number of children of each node in preorder traversal; the zero from the rightmost leaf is omitted. For example, the two trees in the middle section correspond to 2200 (top) and 2020 (bottom) respectively.

Lukasiewicz paths of order n bear a particularly nice correspondence to rooted ordered trees with $n + 1$ nodes. See Fig. 2.2 for an illustration of this.

2.1.2 Cool-Lex Order on Different Combinatorial Objects

Combinations: Fixed-Weight Binary Strings

Generating all binary strings with s zeroes and t ones is often referred to as combinations, since each string can be used to represent a choice of t elements from a set of size of $s+t$. The cool-lex successor rule for generating all fixed-weight binary strings was given by Aaron Williams in his Ph. D thesis and is as follows[Wil09b]:
Let S be a binary string of length n.
Let y be the position of the leftmost zero in S and x be the position of the leftmost 1 in S such that $x \geq y$. Additionally, note that $S_1 ... S_{x-1}$ is the non-increasing prefix of S.

Let $\mathsf{left}(S, x)$ be a function that rotates the first i bits of a string S left circularly by one.

More formally, $\mathsf{left}(S, x) = S_2, S_3, ..., S_i, S_1, S_{i-1}, S_{i+1}, S_{i+2}, ..., S_{2n}$

$$\overleftarrow{\text{cool}}(S) = \begin{cases} \mathsf{left}(S, x) & \text{if } S_{x+1} = 1 \\ \mathsf{left}(S, x+1) & \textit{otherwise} \end{cases}$$

Note that $S_1 ... S_{x-1}$ must be exactly $1^{y-1}0^{x-y}$, where exponentiation denotes repeated symbols. Because of this, the two left-shift operations can be replaced with can be replaced with either one or two symbol transpositions.
Let $\mathsf{transpose}(S, i, j)$ with $1 \leq i \leq j \leq n$ be a function that swaps S_i an S_j. More formally, $\mathsf{transpose}(S, i, j) = S_1, S_2, \ldots, S_{i-1}, S_j S_{i+1} \ldots S_{j-1} S_i S_{j+1} \ldots S_n$ The left-shift rule can be re-stated as follows:

$$\overleftarrow{\text{cool}}(S) = \begin{cases} \mathsf{transpose}(S, y, x) & \text{if } S_{x+1} = 1 \\ \mathsf{transpose}(\mathsf{transpose}(S, y, x), 1, x+1) & \textit{otherwise} \end{cases}$$

Cool Lex Order on Dyck Paths and Binary Trees

Ruskey and Williams found the following successor rule for enumerating binary Dyck words, dubbed "CoolCat" due to its cool-lex order and (cat)alan numbers [RW08]: We will use $\mathbf{B}_n$ to denote binary Dyck words with n ones and n zeroes. Note that the length of any string in $\mathbf{B}_n$ is thus $2n$.
Let $S \in \mathbf{B}_n$
Let the ith prefix shift of S, denoted by $\mathsf{preshift}(S, i)$, be a function that rotates the second through ith symbols of S one to the right circularly. More formally,
$\mathsf{preshift}(S, i) = S_1, S_i, S_2, ..., S_{i-1}, S_{i+1}, S_{i+2}, ..., S_{2n}$

Let k be the index of the 1 in the leftmost 01 substring in S if it exists. Note that if S has no 01 substring, then $S = 1^n0^n$. The successor rule for S is as follows:

$$\overrightarrow{\mathsf{coolCat}}(S) = \begin{cases} \mathsf{preshift}(S, 2n) & \text{if } S \text{ has no 01 substring} \quad (2.1a) \\ \mathsf{preshift}(S, k+1) & \text{if } \mathsf{preshift}(S, k+1) \in \mathbf{B}_n \quad (2.1b) \\ \mathsf{preshift}(S, k) & \text{otherwise} \quad (2.1c) \end{cases}$$

Ruskey and Williams's algorithm can also enumerate a broader set of strings: The algorithm enumerates any set $\mathbf{B}_{s,t}$ where for any $S \in \mathbf{B}_{s,t}$ satisfies the constraint that each prefix of S has as many ones as zeroes. This is slightly broader than the language of Dyck words, as it does not have the requirement that a string have an equal number of ones and zeroes. We will focus on $\mathbf{B}_n$ languages due to their correspondce with Dyck words.

Evaluating whether $\mathsf{preshift}(S, k+1) \in B$ can be determined by looking S_{k+1} and the sum of the first k symbols of S:

The above algorithm can be Let $S' = \mathsf{preshift}(S, k+1)$

Note that we know $S \in \mathbf{B}_n$.

Since preshift only rotates symbols, S' will automaticallly satisfy the requirement that strings in $\mathbf{B}_n$ must have an equal number of zeroes and ones since S satisfied that requirement. Thus, $S' \in \mathbf{B}_n$ will be determined by whether or not all prefixes of S' have at least as many ones as zeroes.

If S_{k+1} is a 1, then every prefix i of S' will have at least as many ones as the corresponding ith prefix of S. Thus, S' must be $\in \mathbf{B}_n$, as rotating a 1 to earlier in the string will never invalidate the requirement that every prefix of the string has at least as many ones as zeroes.

Note that the kth prefix of S must be of the form 1^a0^b1, as otherwise there would be an earlier 01 prefix. Fruthermore, $a \geq b$ as otherwise the bth prefix of S would have more zeroes than ones and S would not be a valid Dyck word.

If S_{k+1} is a 0, then $S' \notin \mathbf{B}_n$ if and only if rotating a 0 to index 2 creates a prefix of S with more zeroes than ones. This will only happen if the k-1th prefix is exactly $1^{\frac{k-1}{2}}0^{\frac{k-1}{2}}$.

Therefore, $\mathsf{preshift}(S, k+1) \in \mathbf{B}_n \iff S_{k+1} = 1$ or S starts with more than $\lfloor \frac{k-1}{2} \rfloor$ ones

$$\overrightarrow{\mathsf{coolCat}}(S) = \begin{cases} \mathsf{preshift}(S, 2n) & \text{if } S \text{ has no 01 substring} \quad (2.2a) \\ \mathsf{preshift}(S, k+1) & S_{k+1} = 1 \text{ or } S \text{ starts with more than } \lfloor \frac{k-1}{2} \rfloor \text{ ones} \quad (2.2b) \\ \mathsf{preshift}(S, k) & \text{otherwise} \quad (2.2c) \end{cases}$$

Since k is the index of the first 01 substring in S, $\sum_{i=1}^{k} S_i$ is actually just the number of consecutive ones to start S, which simplifies the evaluation of this conditional even further.

Ruskey and Williams provided a pseudocode implementation of CoolCat that utilized this fact to enumerate any $\mathbf{B}_{s,t}$ using at most 2 conditionals per successor [RW08].

Due to its simplicity and efficienty, Don Knuth included the cool-lex algorithm for Dyck words in his 4th volume of *The Art of Computer Programming* and also provided an implementation of it for his theoretical MMIX processor architecture [Knu15].

Multiset Permutations

Cool-lex order has also been shown to enumerate multiset permutations via prefix shifts. The rule given by Williams is as follows [Wil09a]:

Let S be a multiset of length n.

Let i be the maximum value such that $S_{j-1} \geq s_j$ for all $2 \leq j \leq i$. In other words, i is the length of the non-increasing prefix of S.

Let $\sigma_j(S)$ be a function that shifts the ith value of S into the first position, or equivalently rotates the first i elements of S right circularly. More formally,

$\sigma_j(S) = S_j, S_1, S_1, \ldots, S_{j-1}, S_j + 1, \ldots, S_n$

Then

$$\text{nextPerm}(S) = \begin{cases} \sigma_{i+1}(S) & \text{if } i \leq n-2 \text{ and } s_{i+2} > s_i \\ \sigma_{i+2}(S) & \text{if } i \leq n-2 \text{ and } s_{i+2} \leq s_i \\ \sigma_n(S) & \textit{otherwise} \end{cases}$$

See Fig. **??** for an example comparison of cool-lex and lexicographic order for two multisets.

This successor rule has the nice property of ensuring that length of the successor's non-increasing prefix is easy to find.

In particular, if S_{i+2} is shifted, then the length of the non-increasing prefix is either 1 if $S_{i+2} \leq S_1$ or $i+1$ otherwise.

Similarly, if S_{i+1} is shifted, then the length of the non-increasing prefix is either 1 if $S_{i+1} \leq S_1$ or $i+1$ otherwise.

This allows for a loopless implementation of the successor rule, as scanning the string to find the length of the non-increasing prefix is not required. Due to the simplicity and efficiency of this rule, it is used in the "multicool" package in R, which is used for generating multiset permutations, Bell numbers, and other combinatorial objects [CWKB21]. Further information on the package is available here: https://www.rdocumentation.org/packages/multicool/versions/0.1-12

TODO: Common threads among cool-lex order. Non-increasing prefix.

Chapter 3

New Results

This thesis provides successor rules and implementations for enumerating the following languages: Ordered trees with a fixed number of nodes, Lukasiewicz words with fixed content, and Motzkin/Schroder words with fixed content. The algorithm for ordered trees is loopless and The algorithm for enumerating Lukasiewicz paths also provides a generalization of the cool-lex successor rule for multiset permutations, given in section 2.

3.1 Loopless Ordered Tree Generation

This chapter presents the first loopless algorithm for generating all ordered trees with n nodes.

Ruskey and Williams previously gave a cool-lex algorithm for looplessly generating all Dyck words of a given length via prefix shifts [RW08]. In the same paper, Ruskey and Williams also gave a loopless algorithm for generating all binary trees with a fixed number in the same order.

This thesis provides a new algorithm that generates ordered trees with a fixed number of nodes in a cool-lex order. The algorithm generates a minimal change ordering of ordered trees in the same order as their corresponding Dyck words in Ruskey and Williams's paper. Like the cool-lex algorithms for Dyck words and binary trees, this algorithm can be implemented looplessly: each ordered tree takes worst-case constant time to generate. This is faster than other algorithms for generating ordered trees which take constant amortized time [PM21] [Er85] [Zak80] [Ska88]. Moreover, taken in conjunction with Ruskey and Williams's algorithms for Dyck words and binary trees, this algorithm completes a trio of loopless cool-lex algorithms for enumerating the three foremost Catalan structures.

Like the cool-lex algorithm for binary trees, this algorithm generates ordered trees stored as pointer structures. This contrasts from other efficient gray codes for enumerating ordered trees, which use either bit-strings or integer sequences to represent ordered trees [PM21] [Zak80] [Er85] as representations of ordered trees. Skarbek's 1988 paper *Generating Ordered Trees* gives a constant amortized time algorithm for generating ordered trees stored as pointer structures and is therefore a a noable exception to this [Ska88]. Generating ordered trees via a pointer structure facilitates the practical use of the trees generated by this algorithm, as a translation step between an alternative representation and a tree structure to traverse the tree is not necessary.

3.1.1 Ordered Trees $\Longleftrightarrow$ Dyck Words

This algorithm will use the bijection between ordered trees and Dyck words specified in [Sta15]. The bijection described by Stanely is as follows:[1]

Given an ordered tree T with $n+1$ nodes: Traverse T in preorder. Whenever going "down" an edge, or away from the root, record a 1. Whenever going "up" an edge, or towards the root, record a 0. The resulting binary sequence is a Dyck word D corresponding to the ordered tree T.

This process can be inverted as follows:

As before, let $D = d_1 ... d_{2n}$ be a dyck word of order n with $n > 0$. Construct an ordered tree T via the following steps.

Create a root node of T. Keep track of a current node *curr*; set $curr = root$.

[1] Stanley's text refers to ordered trees as *plane trees* and Dyck words as *ballot sequences*

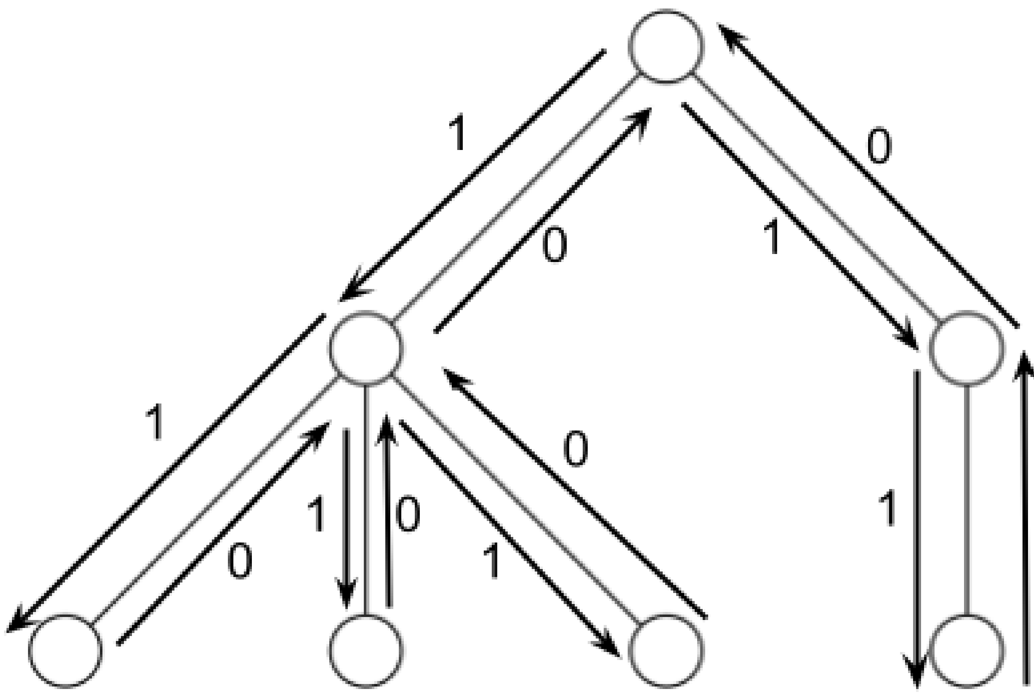


Figure 3.1: An ordered tree with $6+1=7$ nodes corresponding to the order 6 Dyck word 110101001100.

- For each d_i such that $1 \leq i \leq 2n$
  - if $d_i = 1$: append a rightmost child ch to $curr$'s children; set $curr = ch$
  - if $d_i = 0$, set $curr$ equal to $curr$'s parent.

Figure 3.1 demonstrates both directions of this process. Note that each t_i with $1 \leq i \leq n$ in a preorder traversal of T corresponds to the $i^{\underline{th}}$ 1 in D.

In addition to the above bijection, we define the following functions relating to ordered trees, Dyck words, and the correspondence between them.

- Let $\mathsf{OTree}(D)$ and $\mathsf{Dyck}(T)$ be functions that convert a Dyck word to an ordered tree and an ordered tree to a Dyck word respectively via the above process.
- Let $\mathsf{Depth}(t_i)$ = length of path between root and t_i. $\mathsf{Depth}(root) = 0$
- Let $\mathsf{oneindex}(D, i)$ = be the index of the $i^{\underline{th}}$ one in D.

The following remarks can be derived from the bijection between ordered trees and Dyck words.

Remark 1. *t_i corresponds to the ith one in D for $1 \leq i \leq n$*

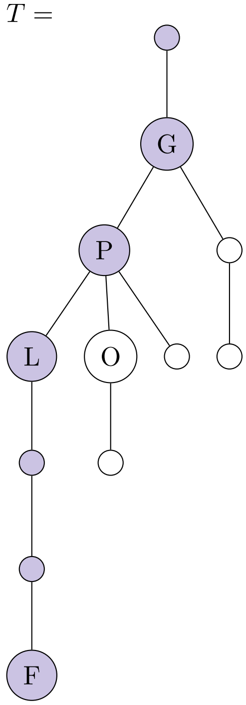
Proof. Recall the method of constructing an ordered tree from a Dyck word. Each one in D creates a new node; zeroes in D do not create nodes. Generating an ordered tree from a Dyck word generates the nodes of the tree in preorder. Thus, t_i corresponds to the ith one in D for $1 \leq i \leq n$. □

Remark 2. *The difference in depths between nodes t_i and t_{i-1} is equal to one minus the number of zeroes between the $(i-1)^{\underline{st}}$ and $i^{\underline{th}}$ and ones in D*

Proof. This remark can be stated formally as

$$\mathsf{Depth}(t_i) - \mathsf{Depth}(t_{i-1}) = 1 - (\mathsf{oneindex}(D, i) - \mathsf{oneindex}(D, i-1) - 1) \quad (3.1)$$

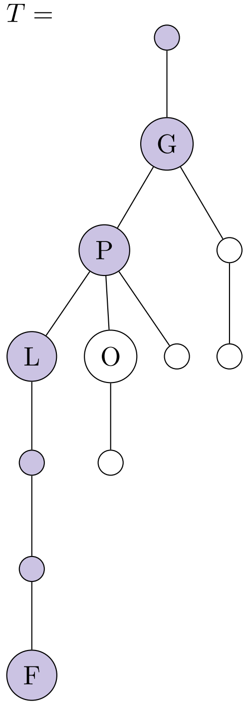
$T =$



$D = [1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0]$

Figure 3.2: An ordered tree with 12 nodes corresponding to the Dyck word 1111110000110010011000. The left down path of T is highlighted in purple.

$T =$



$D = [1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0]$

Figure 3.3: An ordered tree with 12 nodes corresponding to the Dyck word 1111110000110010011000. The left down path of T is highlighted in purple.

Note that ($\mathsf{oneindex}(D,i) - \mathsf{oneindex}(D,i-1) - 1$) is equal to the number of zeroes between the $i^{\underline{th}}$ and $(i-1)^{\underline{st}}$ ones in D.

This follows naturally from the bijection between Dyck words and ordered trees. Each zero corresponds to a step up in the tree before adding the next child.

If there are zero zeroes between the $i^{\underline{th}}$ and $(i-1)^{\underline{st}}$ ones in D, t_i is a child of t_{i-1}; $\mathsf{Depth}(t_i) = \mathsf{Depth}(t_{i-1}) + 1$

If there is one zero between the $i^{\underline{th}}$ and $(i-1)^{\underline{st}}$ ones in D, t_i is a child of t_{i-1}'s parent; $\mathsf{Depth}(t_i) = \mathsf{Depth}(t_{i-1}) + 1$.

Each subsequent zero between t_{i-1} and t_i decreases $\mathsf{Depth}(t_i)$ by one. Thus, the depth of t_i is the depth of t_{i-1} plus 1 minus the number of zeroes between t_{i-1} and t_i. □

Remark 3. *A preorder listing of* $\mathsf{Depth}(t_i)$ *for each* $t_i \in T$ *can be used to construct a Dyck word.*

Proof. Let $T = t_0, t_1, ...t_n$ be a preorder traversal of T. Note that t_0 is the root of T

Construct D as follows:

- Let $D = \epsilon$
- For each t_i, $1 \le i \le n$
  - Append a 1 to D
  - Append $1 - \mathsf{Depth}(t_i) + \mathsf{Depth}(t_{i-1})$ zeroes to D.
- Append $\mathsf{Depth}(t_n)$ zeroes to D.

□

3.1.2 Successor Rule

Let F be the leftmost leaf of T, or equivalently the leftmost descendant of the root. Consider the unique path between the root of T and F, denoted $\mathsf{path}(\mathsf{T}, \mathsf{root}, \mathsf{F})$. We will refer to this path as the left-down path of T $\mathsf{leftpath}(T)$

Given an ordered tree T, let O be the first node in a preorder traversal of T that is not in the $\mathsf{path}(\mathsf{T}, \mathsf{root}, \mathsf{F})$. Let P be O's parent. Let G be P's parent, and let L be P's leftmost child (or, equivalently, O's left sibling). The labels P, G, and L are mnemonics for O's (p)arent, (g)randparent, and (l)eft sibling. Fig. 3.5 gives an example illustrating O,P,G,L,F, and the left-down path in an tree.

The successor rule for enumerating ordered trees with n nodes can be stated as folows:

$$\mathsf{nextree}(T) = \begin{cases} \mathsf{shiftree}(\mathsf{shiftree}(T, L, G), O, root) & \text{if } P \neq root \text{ and O has no children} \quad (3.2a) \\ \mathsf{shiftree}(T, L, O) & \text{otherwise} \quad (3.2b) \end{cases}$$

Figure 3.4 gives a demonstration of the shifts in cases 3.2a and 3.2b

To make the order cyclic, an additional rule can be added, modifying the successor rule to be:

$$\mathsf{nextree}(T) = \begin{cases} \mathsf{shiftree}(T, F, root) & \mathsf{leftpath}(T) = T \quad (3.3a) \\ \mathsf{shiftree}(\mathsf{shiftree}(T, L, G), O, root) & \text{if } P \neq root \text{ and O has no children} \quad (3.3b) \\ \mathsf{shiftree}(T, L, O) & \text{otherwise} \quad (3.3c) \end{cases}$$

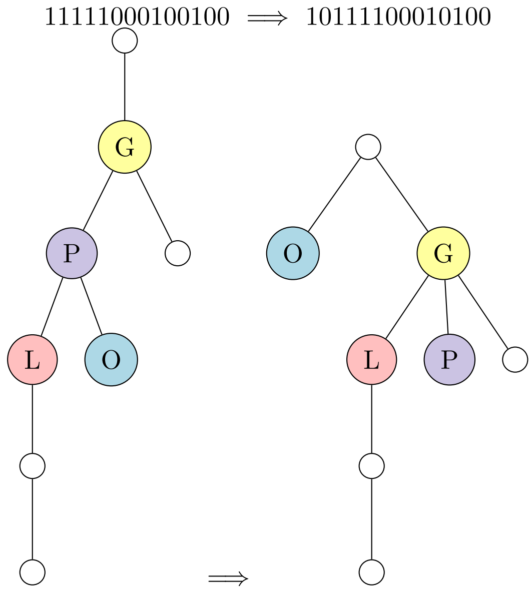
The following remarks can be derived from from the definition of the successor role and the nodes O,G,L, and T.

Let $D = \mathsf{Dyck}(T)$; s be the number of consecutive ones to start D, and z be the number of consecutive zeroes starting at d_{s+1}. Note that $z = (k - s - 1)$; $d_k = 1$

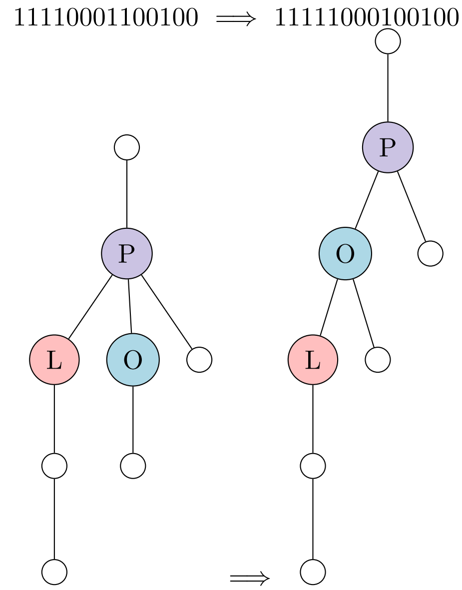
Remark 4. $\mathsf{Depth}(O) = s - z + 1$

Proof. t_s is the last node in the $\mathsf{leftpath}(T)$, as the left-down path has $s+1$ nodes starting at t_0. t_s has depth s, as it is exactly s steps from the root. Note that $O = t_{s+1}$. The number of zeroes between t_s and t_{s+1} is the number of zeroes between the $s^{\underline{th}}$ and $(s+1)^{\underline{st}}$ ones in D_i.

□



(a) O has no children, $P \neq root$:
$\textsf{shiftree}(\textsf{shiftree}(T, L, G), O, root)$
L becomes G's first child;
O becomes the first child of $root$



(b) O has at least 1 child
$\textsf{shiftree}(T, L, O)$
L becomes O's first child

Figure 3.4: Illustrations of cases 3.2a and 3.2b

Remark 5. *O corresponds to D_k, i.e.* $\textsf{oneindex}(D, s+1) = k$

Proof. Let $D = \textsf{Dyck}(T)$ and let k be the index of the 1 in the leftmost 01 substring of D. Let $t_0 ... t_s = \textsf{leftpath}(T)$; $O = t_{s+1}$.

Note that each 1 in D corresponds to a step down; each 0 to a step up. Consequently, $\textsf{leftpath}(T)$ corresponds to the "all-one" prefix of D. In other words, $\textsf{leftpath}(T) = t_0, t_1, ... t_s$ such that $i = 0$ or $D_i = 1$. Note that t_{s+1} is therefore the first node in a preorder traversal of T such that $D_{\textsf{oneindex}(D,s+1)} = 1$ and $D_{\textsf{oneindex}(D,s+1)-1} = 0$. O is therefore also the first node in a preorder traversal of T such that $t_{s+1} \notin \textsf{leftpath}(T)$. Therfore, $\textsf{oneindex}(D, s+1) = k$, i.e., $t_{s+1} = O$ corresponds to the 1 in the leftmost 01 substring of D.

□

Remark 6. *Every non-leaf node below P in* $\textsf{leftpath}(T)$ *has exactly 1 child.*

Proof. Suppose by way of contradiction that a node below P in $\textsf{leftpath}(T)$ had a second child. That child would not be in $\textsf{leftpath}(T)$ and would be be traversed before O in preorder. O was specified to be the first node in a preorder traversal of T that is not in $\textsf{leftpath}(T)$, which generates a contradiction.

□

Remark 7. *L corresponds to D_{s-z+1}, i.e.* $\textsf{oneindex}(D, s+1) = k$

Proof. $\textsf{Depth}(L) = \textsf{Depth}(O) = s - z - 1$ since L and O are siblings. Therefore, L must be $s - z - 1$ steps down from the root $\Longrightarrow$ L is the $s - j - 1$th node in a preorder traversal of T $\Longrightarrow$ T corresponds to D_{s-z-1}.

□

3.1.3 Proof of Correctness

Ruskey and Williams proved that, given a Dyck word of order n, 2.2 iteratively generates all Dyck words of order n. This proof will use the bijection between Dyck words of order n and ordered trees with $n + 1$ nodes to show that that 3.2 generates all ordered trees with a given number of nodes.

Recall that the successor rule $\overrightarrow{\mathsf{coolCat}}(D)$ generates all Dyck words. Therefore, To prove that $\mathsf{nextree}(T)$ generates all ordered trees with $|T|$ nodes, it is sufficient to show that, given an arbitrary ordered tree T,

Theorem 3.1.1. *Given an ordered tree T,* $\mathsf{nextree}(T) = \mathsf{OTree}(\overrightarrow{\mathsf{coolCat}}(\mathsf{Dyck}(T)))$

Proof. $\overrightarrow{\mathsf{coolCat}}(D)$ and $\mathsf{nextree}(T)$ are each broken down into 3 cases in equations 2.2 and 3.2 respectively.

For convenience, equations 3.4 and 3.5 give the expanded restatemtents of the successor rules for $\mathsf{nextree}(T)$ and $\overrightarrow{\mathsf{coolCat}}(D)$ to facilitate comparisons between the two.

$$\mathsf{nextree}(T) = \begin{cases} \mathsf{shiftree}(T, F, root) & \mathsf{leftpath}(T) = T \quad (3.4a) \\ \mathsf{shiftree}(T, L, O) & \text{if O has at least 1 child} \quad (3.4b) \\ \mathsf{shiftree}(\mathsf{shiftree}(T, L, G), O, root) & \text{if } P \neq root \text{ and O has no children} \quad (3.4c) \\ \mathsf{shiftree}(T, L, O) & \text{if O has no children and } P = root \quad (3.4d) \end{cases}$$

$$\overrightarrow{\mathsf{coolCat}}(D) = \begin{cases} \mathsf{preshift}(D, 2n) & \text{if } D \text{ has no 01 substring} \quad (3.5a) \\ \mathsf{preshift}(D, k+1) & D_{k+1} = 1 \quad (3.5b) \\ \mathsf{preshift}(D, k+1) & D_{k+1} = 0 \text{ and } s > \frac{k-1}{2} \quad (3.5c) \\ \mathsf{preshift}(D, k) & D_{k+1} = 0 \text{ and } s = \frac{k-1}{2} \quad (3.5d) \end{cases}$$

We will show the following equivalences:

- 3.4a corresponds to 3.5a
- 3.4b corresponds to 3.5b
- 3.4c corresponds to 3.5c
- 3.4d corresponds to 3.5d

To accomplish this, we will first prove a few auxillary lemmas to be used to show equivalency between cases.

Let $D = \mathsf{Dyck}(T)$, s be the number of consecutive ones to start D, and z be the number of consecutive zeroes starting at d_{s+1}. Note that $z = (k - s - 1)$; $d_k = 1$

Lemma 3.1.2. *D has no 01 substring* $\iff$ *$LD(T) = T$*

Proof. If D has no 01 substring, $D = 1^n0^n$, and T is $n+1$ nodes where t_0 is the root and each t_i for $1 \leq i \leq n$ is a child of t_{i-1} In this case, T is a single path of $n+1$ nodes, and the left-down path of T is the entire tree. □

Lemma 3.1.3. *$D_{k+1} = 0 \iff$ O has no children*

Proof. This follows logically from the bijection between Dyck words and ordered trees. D_k corresponds to O. If $D_{k+1} = 0$, an "upward" step is taken after O and consequently the next node after O cannot be a child of O. Since the ones in D give the nodes of T in preorder, O must have no children.

Informally, once you go "up" from O, the bijection between Dyck words and ordered trees gives no way to go "back down" to give O an additional child. □

Lemma 3.1.4. *$P = root \iff s = z = \frac{k-1}{2}$.*

Proof. First, note that $P = root$ simply means that O is a child of the root. O is a child of the root $\iff \mathsf{Depth}(O) = 1$. Additionally, note that $s + z = k - 1$

As shown in remark 4, $\mathsf{Depth}(O) = s - z + 1$. Therefore, $P = root \iff s = z = \frac{k-1}{2}$ i.e. the first $k-1$ symbols of D are $\frac{k-1}{2}$ ones followed by $\frac{k-1}{2}$ zeroes.

□

Lemma 3.1.5. *3.4a corresponds to 3.5a*

Proof. Let $D = \mathsf{Dyck}(T)$

Per lemma 3.1.2 D has no 01 substring $\iff$ $\mathsf{leftpath}(T) = T$.

Thus, $\mathsf{nextree}(T)$ executes case 3.4a if and only if $\overrightarrow{\mathsf{coolCat}}(D)$ executes case 3.5a

Note that since D has no 01 substring, $D = 1^n 0^n$.

Additionally, since $\mathsf{leftpath}(T) = T$, T can be specified as follows.

$T =$

node	t_0	t_1	t_2	$\dots$	t_{n-1}	$F = t_s = t_n$
depth	0	1	2	$\dots$	$n-1$	n
Dyck		$1^n 0^n$				

The third row of this table illustrates the construction of $\mathsf{Dyck}(T)$ via the process specified in remark 3.

Shifting F to be the first child of the root changes $\mathsf{Depth}(F)$ to 1 and does not affect the depth of any other nodes. Thus, if $T' = \mathsf{nextree}(T)$,

$T =$

node	t_0	$F = t_s = t_n$	t_1	t_2	$\dots$	t_{n-1}
depth	0	1	1	2	$\dots$	$n-1$
Dyck		1	$01^{n-1}0^{n-1}$			

Recall that $\overrightarrow{\mathsf{coolCat}}(D) = \mathsf{preshift}(D, 2n)$ if D has no 01 substring. $D_{2n} = 0$, and therefore $\overrightarrow{\mathsf{coolCat}}(D) = 101^{n-1}0^{n-1}$

Note that this is exactly the Dyck word constructed from T'. Therefore, if D has no 01 substring or $\mathsf{leftpath}(T) = T$,

$\mathsf{OTree}(\overrightarrow{\mathsf{coolCat}}(D)) = \mathsf{nextree}(T)$

□

Lemma 3.1.6. *3.4c corresponds to 3.5c*

Proof. Let $D = \mathsf{Dyck}(T)$

Per lemma 3.1.4 $P = root$ $\iff$ D starts with exactly $\frac{k-1}{2}$ ones.

It was also previously shown that $D_{k+1} = 0 \iff O$ has no children. Thus, $\mathsf{nextree}(T)$ executes case 3.2a if and only if $\overrightarrow{\mathsf{coolCat}}(D)$ executes case 3.5c

We now show that the execution of 3.2a is equivalent to the execution of 3.5c given case a. Given $\mathsf{Dyck}(T) = D = 1^s 0^z 10 d_{k+2} d_{k+3} \dots d_{2n}$, we aim to show that

$\mathsf{Dyck}(\mathsf{nextree}(T)) = \overrightarrow{\mathsf{coolCat}}(\mathsf{Dyck}(T))$

Note that $\mathsf{nextree}(T) = \mathsf{shiftree}(\mathsf{shiftree}(T, L, G), O, root)$.

Let $T' = \mathsf{shiftree}(T, L, G)$; $T'' = \mathsf{shiftree}(T', O, root)$

Note that $\mathsf{nextree}(T) = T''$

Since $P \neq root$, we know that G, the parent of P, exists. Thus, we can assume that $G, P, L \in \mathsf{leftpath}(T)$. T can therefore be specified as follows:

$T =$

node	t_0	t_1	$\dots$	$G = t_{s-z-1}$	$P = t_{s-z}$	$L = t_{s-z+1}$	$\dots$	$F = t_s$	$O = t_{s+1}$	$\dots$
depth	0	1	$\dots$	$(s-z-1)$	$(s-z)$	$(s-z+1)$	$\dots$	s	$(s-z+1)$	$\dots$
Dyck		1^s							$0^z 1$	$0 \dots$

Furthermore, recall that L (and all other non-leaf nodes $\in \mathsf{leftpath}(T)$ must have exactly one child. Therefore, every node below L in $\mathsf{leftpath}(T)$ has its depth reduced by one; no other nodes have their depth affected by this shift. Therefore, T' can be written as follows:

$T' =$

node	t_0	t_1	$\dots$	$G = t_{s-z-1}$	$L = t_{s-z+1}$	$\dots$	$F = t_s$	$P = t_{s-z}$	$O = t_{s+1}$	$\dots$
depth	0	1	$\dots$	$(s-z-1)$	$(s-z)$	$\dots$	$s-1$	$(s-z)$	$(s-z+1)$	$\dots$
Dyck		1^{s-1}						0^z1	1	$0\dots$

Since L is now G's first child, P changes from being G's first child to G's second child. P is therefore removed from the left-down path of T', thereby making P the first node in a preorder traversal of T' that is not in the left-down path of T'. Therefore, $|\mathsf{leftpath}(T')| = s$; $O' = P$.

Recovering a Dyck word from T', we obtain

D'=$1^{s-1}0^z110d_{k+2}d_{k+3},\dots,d_{2n}$

Next, we use $\mathsf{shiftree}(T', O, root)$ to obtain $T'' = \mathsf{nextree}(T)$

$\mathsf{shiftree}(T', O, root)$ shifts O to become the first child of the root. Note that we know that O has no children. Consequently, no nodes other than O have their depth affected by this shift. Thus,

$T'' =$

node	t_0	$O = t_{s+1}$	t_1	t_2	$\dots$	$G = t_{s-z-1}$	$L = t_{s-z+1}$	$\dots$	$F = t_s$	$P = t_{s-z}$	$\dots$
depth	0	1	1	2	$\dots$	$(s-z-1)$	$(s-z)$	$\dots$	$s-1$	$(s-z)$	$\dots$
Dyck		1	01^{s-1}							0^z1	$\dots$

Therefore, since $T'' = \mathsf{nextree}(T)$, $\mathsf{Dyck}(\mathsf{nextree}(T)) = 101^{s-1}0^z1\dots$

Since $\mathsf{Dyck}(T) = D = 1^s0^z10\dots$ 3.5b gives that

$\overrightarrow{\mathsf{coolCat}}(\mathsf{Dyck}(T)) = 101^{s-1}0^z1\dots$

Therefore, we have shown that $\mathsf{Dyck}(\mathsf{nextree}(T)) = \overrightarrow{\mathsf{coolCat}}(\mathsf{Dyck}(T)) = 101^{s-1}0^z1\dots$

∎

Lemma 3.1.7. *3.4b corresponds to 3.5b*

Proof. Per 3.1.3, as O has at least 1 child $\iff D_{k+1} = 1$.

Thus, $\mathsf{nextree}(T)$ will execute case 3.4b if and only if $\overrightarrow{\mathsf{coolCat}}(D)$ executes case 3.5b

Therefore, we aim to show that, given O has at least child and $D_{k+1} = 1$,

$\mathsf{preshift}(\mathsf{Dyck}(T), k+1) = \mathsf{Dyck}(\mathsf{shiftree}(T, L, O))$

Since $D_{k+1} = 1$, we can rewrite D as. $D = 1^s0^z11$

$T =$

node	t_0	t_1	$\dots$	$G = t_{s-z-1}$	$P = t_{s-z}$	$L = t_{s-z+1}$	$\dots$	$F = t_s$	$O = t_{s+1}$	$t_{s+2}\dots$
depth	0	1	$\dots$	$(s-z-1)$	$(s-z)$	$(s-z+1)$	$\dots$	s	$(s-z+1)$	$s-z+2\dots$
Dyck		1^s							0^z1	$1\dots$

Shift L to be O's first child:

Nodes $L = t_{s-z+1}$ through $F = t_s$ will now come after O in preorder traversal. Additionally, $\mathsf{leftpath}(T)$ will now go through O; every node in $\mathsf{path}(\mathsf{T}, \mathsf{L}, \mathsf{F})$ will have its depth increased by one.

Therefore, $T' = \mathsf{nextree}(T)$ can be specified as follows:

$T' =$

node	t_0	t_1	$\dots$	$G = t_{s-z-1}$	$P = t_{s-z}$	$O = t_{s+1}$	$L = t_{s-z+1}$	$\dots$	$F = t_s$	$t_{s+2}\dots$
depth	0	1	$\dots$	$(s-z-1)$	$(s-z)$	$(s-z+1)$	$(s-z+2)$	$\dots$	$s+1$	$s-z+2\dots$
Dyck		1^{s+1}								$0^z1\dots$

Note that $z \geq 1$, so z zeroes occur between the one corresponding to t_s and the one corresponding to t_{s+2}.

Next, recall that $D = \mathsf{Dyck}(T) = D = 1^s0^z11\dots$ and that $k = s+z+1$

Therefore, $\overrightarrow{\mathsf{coolCat}}(D) = \mathsf{preshift}(D, k+1) = 1^{s+1}0^z1\dots$, which is the same as the Dyck word resulting from translating $T' = \mathsf{nextree}(T)$ to the Dyck word $1^{s+1}0^z1\dots$

∎

Lemma 3.1.8. *3.4d corresponds to 3.5d*

Proof. $T \neq \mathsf{leftpath}(T) \iff D$ has a 01 substring.
$D_{k+1} = 1 \iff O$ has at least one child.
$D_{k+1} = 0$ and $s = \frac{k-1}{2} \iff O$ has no children and O is a child of the root.
O has no children and P=root. Therefore $s = z$, $k = 2s+1$
We can thus rewrite $D = \mathsf{Dyck}(T) = 1^s0^s101\ldots$
Furthermore, since $s = z$, O has depth 1.
$T =$

node	$P = t_0$	$L = t_1$	$\ldots$	$F = t_s$	$O = t_{s+1}$	$t_{s+2}\ldots$
depth	0	1	$\ldots$	s	1	1
Dyck		1^s			0^s1	$01\ldots$

Nodes $L = t_1$ through $F = t_s$ will now come after O in preorder traversal. Additionally, $\mathsf{leftpath}(T)$ will now go through O; every node in $\mathsf{path}(\mathsf{T},\mathsf{L},\mathsf{F})$ will have its depth increased by one.

Therefore, $T' = \mathsf{nextree}(T)$ can be specified as follows:
$T' =$

node	$P = t_0$	$O = t_{s+1}$	$L = t_1$	$\ldots$	$F = t_s$	$t_{s+2}\ldots$
depth	0	1	2	$\ldots$	$s+1$	1
Dyck		1^{s+1}				$0^{s+1}1\ldots$

Since $D = \mathsf{Dyck}(T) = 1^s0^s101\ldots$, $\overrightarrow{\mathsf{coolCat}}(D) = 1^{s+1}0^{s+1}1\ldots$ as per case *3.4d*. This is identical to the Dyck word constructed from $T' = \mathsf{nextree}(T)$. Therefore, cases 3.4d and 3.5d are equivalent. ∎

Since these 4 cases cover all cases for the two successor rules, we have shown that $\mathsf{nextree}(T) = \mathsf{OTree}(\overrightarrow{\mathsf{coolCat}}(\mathsf{Dyck}(T)))$ in all cases. ∎

3.1.4 Loopless Implementation

The algorithm described in this section has been implemented in C using a tree node struct that contains fields for each node's parent, left child, and right sibling.

The functions `shift_tree_a` and `shift_tree_b` perform the shifts outlined in cases 3.2a and 3.2b respectively; the function `get_initial_tree` generates the first tree in the ordering.

`t` is a parameter equal to the number of non-root nodes in the tree; `visit` is a user-supplied function for visiting a tree.

```
void coolOtree(int t, void (*visit)(node*)){
    node* root = get_initial_tree(t);
    node* o=root->left_child->right_sibling;

    visit(root);
    //o is NULL for the final tree
    while(o){
        if(o->left_child){ //if o has a child
            shift_tree_b(root,o); 3.2b
            o=o->left_child->right_sibling;
        }else{
            if(o->parent == root){
                shift_tree_b(root,o); 3.2b
                o=o->right_sibling;
            }else{ //if the string isn't tight, shift a zero
                shift_tree_a(root,o); 3.2a
                o=o->right_sibling;
            }
        }
        visit(root);
```

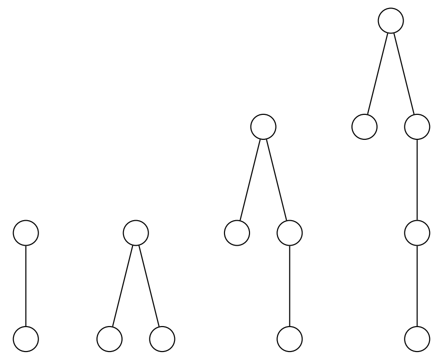


Figure 3.5: The initial trees returned by `get_initial_tree` in `cool0tree` with $t = 1, 2, 3$, and 4

```
    }
}
```

Algorithm 1 Generate all ordered trees with $t+1$ nodes

function COOL-ORDERED-TREES(t)
    ▷ Generate initial tree
     TODO
    visit(root)
    $O \leftarrow root.lchild$
    `visit`(root)
    while $O \neq NULL$ **do**
        if $O.lchild \neq NULL$ **then**
            $P \leftarrow O.parent$
            $L \leftarrow P.lchild$
            $P.lchild \leftarrow O$
            $L.parent \leftarrow O$
    ▷ Initialize instance variables
    Initialize $incs$ as an empty stack
    $m \leftarrow |a|$
    $prefix\_sum \leftarrow |a| - 1$

    ▷ Loop through all permutations
    while True **do**
        ▷ Find indices for the shift
        if $m == |a|$ **then**
            $insert\_index \leftarrow 1$
            $shift\_index \leftarrow |a| - 1$
        else if $a_{m+1} > a_m - 1$ **then**
            if $a_{m+1} > a_m$ **then**
                $incs.pop()$
            $shift\_index \leftarrow m$
            $insert\_index \leftarrow 0$
            $incs.append(m+1)$
        else if $a\_m + 1 > 0$ or $prefix\_sum > prefix\_len$ **then**
            if $a_{m+2} > a_{m+1}$ and $a_{m+2} <= a_m$ **then**
                $incs.pop()$
            $shift\_index \leftarrow m - 1$
            if $a_{shift\_index} > 0$ **then**
                $insert\_index \leftarrow 0$
            else
                $insert\_index \leftarrow 1$
            $incs.append(m+1)$
        else
            $shift\_index \leftarrow m$
            $insert\_index \leftarrow 0$

        ▷ Do shift and visit
        $a.insert(insert\_index, a.remove(shift\_index))$
        `visit`(a)

        ▷ Update variables for next iteration
        if $a_{insert\_index} < a_{insert\_index+1}$ **then**
            $prefix\_sum \leftarrow a_0$
            if $insert\_index \neq m$ **then**
                $incs.append(m+1)$
        else
            $prefix\_sum \leftarrow prefix\_sum + a_{insert\_index}$
        if $|incs| = 0$ **then**
            return
        else
            $m \leftarrow incs.pop()$

Bibliography

[CWKB21] James Curran, Aaron Williams, Jerome Kelleher, and Dave Barber. Package 'multicool', Jun 2021.

[Er85] MC Er. Enumerating ordered trees lexicographically. *The Computer Journal*, 28(5):538–542, 1985.

[Knu15] Donald Ervin Knuth. The art of computer programming: Combinatorial algorithms, vol. 4, 2015.

[PM21] 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.

[Rus03] Frank Ruskey. Combinatorial generation. *Preliminary working draft. University of Victoria, Victoria, BC, Canada*, 11:20, 2003.

[RW08] 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.

[Ska88] Wladyslaw Skarbek. Generating ordered trees. *Theoretical Computer Science*, 57(1):153–159, 1988.

[Sta15] Richard P Stanley. *Catalan numbers*. Cambridge University Press, 2015.

[Wil09a] Aaron Williams. Loopless generation of multiset permutations by prefix shifts. In *SODA 2009, Symposium on Discrete Algorithms*, 2009.

[Wil09b] Aaron Michael Williams. *Shift gray codes*. PhD thesis, 2009.

[Zak80] Shmuel Zaks. Lexicographic generation of ordered trees. *Theoretical Computer Science*, 10(1):63–82, 1980.