Space Efficient Construction of Lyndon Arrays in Linear Time

Philip Bille
DTU Compute, Technical University of Denmark, Lyngby, Denmark
phbi@dtu.dk

Jonas Ellert
Department of Computer Science, Technical University of Dortmund, Germany
jonas.ellert@tu-dortmund.de

Johannes Fischer
Department of Computer Science, Technical University of Dortmund, Germany
johannes.fischer@cs.tu-dortmund.de

Inge Li Gørtz
DTU Compute, Technical University of Denmark, Lyngby, Denmark
inge@dtu.dk

Florian Kurpicz
Department of Computer Science, Technical University of Dortmund, Germany
florian.kurpicz@tu-dortmund.de

J. Ian Munro
Cheriton School of Computer Science, University of Waterloo, Canada
imunro@uwaterloo.ca

Eva Rotenberg
DTU Compute, Technical University of Denmark, Lyngby, Denmark
erot@dtu.dk

Abstract

Given a string S of length n, its *Lyndon array* identifies for each suffix $S[i..n]$ the next lexicographically smaller suffix $S[j..n]$, i.e. the minimal index $j > i$ with $S[i..n] \succ S[j..n]$. Apart from its plain $(n \log_2 n)$-bit array representation, the Lyndon array can also be encoded as a succinct parentheses sequence that requires only $2n$ bits of space. While linear time construction algorithms for both representations exist, it has previously been unknown if the same time bound can be achieved with less than $\Omega(n \lg n)$ bits of additional working space. We show that, in fact, $o(n)$ additional bits are sufficient to compute the succinct $2n$-bit version of the Lyndon array in linear time. For the plain $(n \log_2 n)$-bit version, we only need $\mathcal{O}(1)$ additional words to achieve linear time. Our space efficient construction algorithm makes the Lyndon array more accessible as a fundamental data structure in applications like full-text indexing.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases String algorithms, string suffixes, succinct data structures, Lyndon word, Lyndon array, nearest smaller values, nearest smaller suffixes

Digital Object Identifier 10.4230/LIPIcs.ICALP.2020.14

Category Track A: Algorithms, Complexity and Games

Related Version `https://arxiv.org/abs/1911.03542`

Supplementary Material `https://github.com/jonas-ellert/nearest-smaller-suffixes`

Funding *Philip Bille*: Supported by the Danish Research Council (DFF-9131-00069B).
Inge Li Gørtz: Partially supported by the Danish Research Council (DFF-8021-002498).
Florian Kurpicz: Supported by the German Research Foundation DFG SPP 1736 "Algorithms for Big Data".
J. Ian Munro: Supported by the Canada Research Chairs Programme and NSERC Discovery Grant 8237.
Eva Rotenberg: Partially supported by Independent Research Fund Denmark grants 2020-2023 (9131-00044B), 2020-2023 (8021-002498), and 2018-2021 (8021-00249B).

47th International Colloquium on Automata, Languages, and Programming (ICALP 2020).
Editors: Artur Czumaj, Anuj Dawar, and Emanuela Merelli; Article No. 14; pp. 14:1–14:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction & Related Work

The Lyndon array [5] is a well-known combinatorial object on strings and has gained renewed attention [6, 7, 14, 18, 19, 23] due to its recently discovered central role in combinatorics on strings, e.g., when computing all the runs in a string [2]. It is known [18, 13] that the Lyndon array can be computed in linear time from the list of lexicographically sorted suffixes (the *suffix array*). Baier [1] introduced the first *direct* algorithm for computing the Lyndon array – interestingly as a *preliminary* step for his new suffix sorting algorithm. However, it requires $\Theta(n \lg n)$ bits of additional working space even for just computing the Lyndon array. Other Lyndon array construction algorithms have been introduced [13, 19, 20], but they all have the same space bounds.

The Lyndon array has some structural properties that allow for a more space efficient representation, namely using only $2n + 2$ bits [19]. Thus, it would be desirable to compute this succinct representation using less than $\Theta(n \lg n)$ bits of working space, without sacrificing the linear running time. Previously, no such algorithm was known.

Our Contributions. We introduce the first algorithm that computes the succinct Lyndon array in $\mathcal{O}(n)$ time, using only $\mathcal{O}(n \lg \lg n / \lg n)$ bits of additional working space. Alternatively, our algorithm can construct the plain ($\mathcal{O}(n \lg n)$-bits) Lyndon array using only $\mathcal{O}(1)$ words of additional working space, i.e., directly without precomputing the suffix array. In practice, our approach is up to 10 times faster than previous algorithms for the Lyndon array.

The rest of the paper is organized as follows: In Section 2 we introduce the notation and definitions that we use throughout the paper. Section 3 provides a new intuitive definition of the succinct Lyndon array. We introduce our construction algorithm for the succinct Lyndon array in Sections 4 and 5, and adapt it such that it computes the plain ($\mathcal{O}(n \lg n)$-bits) Lyndon array in Section 6. Finally, we present experimental results for both versions (Section 7), and summarize our findings (Section 8).

2 Preliminaries

We write $\lg x$ for $\log_2 x$. For $i, j \in \mathbb{N}$, the interval $[i, j]$ represents $\{x \mid x \in \mathbb{N} \land i \leq x \leq j\}$. We use the notation $[i, j+1) = (i-1, j] = (i-1, j+1) = [i, j]$ for open and half-open discrete intervals. Our analysis is performed in the word RAM model [17], where we can perform fundamental operations (logical shifts, basic arithmetic operations etc.) on words of size w bits in constant time. For the input size n of our problems we assume $\lceil \lg n \rceil \leq w$.

A *string* (also called *text*) over the *alphabet* Σ is a finite sequence of *symbols* from the finite and totally ordered set Σ. We say that a string $\mathcal{S}$ has length n and write $|\mathcal{S}| = n$, iff $\mathcal{S}$ is a sequence of exactly n symbols. The i-th symbol of a string $\mathcal{S}$ is denoted by $\mathcal{S}[i]$, while the *substring* from the i-th to the j-th symbol is denoted by $\mathcal{S}[i..j]$. For convenience we use the interval notations $\mathcal{S}[i..j+1) = \mathcal{S}(i-1..j] = \mathcal{S}(i-1..j+1) = \mathcal{S}[i..j]$. The i-th *suffix* of $\mathcal{S}$ is defined as $\mathcal{S}_i = \mathcal{S}[i..n]$, while the substring $\mathcal{S}[1..i]$ is called *prefix* of $\mathcal{S}$. A prefix or suffix of $\mathcal{S}$ is called *proper*, iff its length is at most $n-1$. The concatenation of two strings $\mathcal{S}$ and $\mathcal{T}$ is denoted by $\mathcal{S} \cdot \mathcal{T}$. The length of the *longest common prefix (LCP)* between $\mathcal{S}$ and $\mathcal{T}$ is defined as $\textsc{lcp}(\mathcal{S}, \mathcal{T}) = \max\{\ell \mid \mathcal{S}[1..\ell] = \mathcal{T}[1..\ell]\}$. The *longest common extension (LCE)* of indices i and j is the length of the LCP between $\mathcal{S}_i$ and $\mathcal{S}_j$, i.e. $\textsc{lce}(i, j) = \textsc{lcp}(\mathcal{S}_i, \mathcal{S}_j)$. We can simplify the description of our algorithm by introducing a special symbol $\$ \notin \Sigma$ that is smaller than all symbols from Σ. For a string $\mathcal{S}$ of length n we define the 0-*th suffix* $\mathcal{S}_0 = \$$ as well as the $(n+1)$-*st suffix and position* $\mathcal{S}_{n+1} = \mathcal{S}[n+1] = \$$. The total order on Σ

induces a total order on the set Σ^* of strings over Σ. Let $\mathcal{S}$ and $\mathcal{T}$ be strings over Σ, and let $\ell = \textsc{lcp}(\mathcal{S}, \mathcal{T})$. We say that $\mathcal{S}$ is lexicographically smaller than $\mathcal{T}$ and write $\mathcal{S} \prec \mathcal{T}$, iff we have $\mathcal{S} \neq \mathcal{T}$ and $\mathcal{S}[\ell+1] < \mathcal{T}[\ell+1]$. Analogously, we say that $\mathcal{S}$ is lexicographically larger than $\mathcal{T}$ and write $\mathcal{S} \succ \mathcal{T}$, iff we have $\mathcal{S} \neq \mathcal{T}$ and $\mathcal{S}[\ell+1] > \mathcal{T}[\ell+1]$.

2.1 The Lyndon Array & Nearest Smaller Suffixes

A *Lyndon word* is a string that is lexicographically smaller than all of its proper suffixes, i.e. $\mathcal{S}$ is a Lyndon word, iff $\forall i \in [2..n] : \mathcal{S}_i \succ \mathcal{S}$ holds [8]. For example, the string `northamerica` is not a Lyndon word because its suffix `america` is lexicographically smaller than itself. On the other hand, its substring `americ` is a Lyndon word. The Lyndon array of $\mathcal{S}$ identifies the longest Lyndon word at each position of $\mathcal{S}$:

▶ **Definition 1** (Lyndon Array). *Given a string $\mathcal{S}$ of length n, the Lyndon array is an array λ of n integers with $\lambda[i] = \max\{\ell \mid \mathcal{S}[i..i+\ell) \text{ is a Lyndon word}\}$.*

▶ **Definition 2** (Nearest Smaller Suffixes). *Given a string $\mathcal{S}$ and a suffix $\mathcal{S}_i$, the* next smaller suffix of $\mathcal{S}_i$ *is $\mathcal{S}_j$, where j is the smallest index that is larger than i and satisfies $\mathcal{S}_i \succ \mathcal{S}_j$. The* previous smaller suffix of $\mathcal{S}_i$ *is defined analogously. The* next-smaller-suffix array (NSS array) and previous-smaller-suffix array (PSS array) *are arrays of size n defined as follows:*

$$\mathsf{nss}[i] = \min\{j \mid j \in (i, n+1] \wedge \mathcal{S}_i \succ \mathcal{S}_j\} \qquad \mathsf{pss}[i] = \max\{j \mid j \in [0, i) \wedge \mathcal{S}_j \prec \mathcal{S}_i\}$$

The Lyndon array and nearest smaller suffixes are highly related to each other. In fact, the NSS array is merely a different representation of the Lyndon array (Lemma 3), as visualized in Figure 1a. We conclude the preliminaries by showing a slightly weaker connection between the PSS array and Lyndon words (Lemma 4).

▶ **Lemma 3** (Lemma 15 [13]). *The longest Lyndon word at position i ends at the starting position of the NSS of $\mathcal{S}_i$, i.e. $\lambda[i] = \mathsf{nss}[i] - i$.*

▶ **Lemma 4.** *Let $\mathsf{pss}[j] = i > 0$, then $\mathcal{S}[i..j)$ is a Lyndon word.*

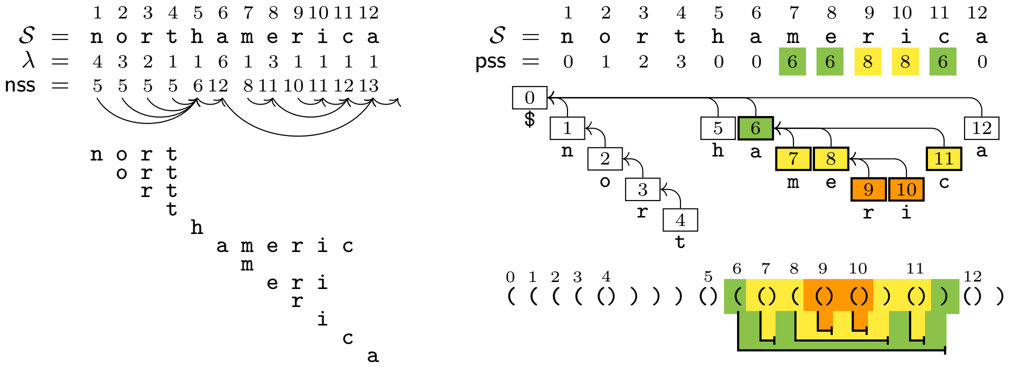
Proof. By definition, the string $\mathcal{S}[i..j)$ is a Lyndon word iff there exists no $k \in (i, j)$ with $\mathcal{S}[k..j) \prec \mathcal{S}[i..j)$. Assume that such a k exists. Since $i = \mathsf{pss}[j]$, we know that (a) $\mathcal{S}_k \succ \mathcal{S}_i$. Now assume there is a mismatching character between $\mathcal{S}[k..j)$ and $\mathcal{S}[i..j)$. Then appending $\mathcal{S}_j$ to both strings preserves this mismatch. This implies that we have $\mathcal{S}[k..j) \prec \mathcal{S}[i..j) \iff \mathcal{S}[k..j) \cdot \mathcal{S}_j \prec \mathcal{S}[i..j) \cdot \mathcal{S}_j$, and thus $\mathcal{S}_k \prec \mathcal{S}_i$, which contradicts (a). Therefore, we know that (b) $\mathcal{S}[k..j) = \mathcal{S}[i..i+(j-k))$. Then

$$\mathcal{S}_k \overset{\text{(a)}}{\succ} \mathcal{S}_i \quad\iff\quad \mathcal{S}[k..j) \cdot \mathcal{S}_j \succ \mathcal{S}[i..i+(j-k)) \cdot \mathcal{S}_{i+(j-k)} \quad\overset{\text{(b)}}{\iff}\quad \mathcal{S}_j \succ \mathcal{S}_{i+(j-k)}$$

which contradicts the fact that $\mathsf{pss}[j] = i < i + (j-k)$. Hence, the described k cannot exist, and $\mathcal{S}[i..j)$ must be a Lyndon word. ◀

3 Previous-Smaller-Suffix Trees

In this section we introduce the *previous-smaller-suffix tree*, which simulates access to the Lyndon array, the NSS array, and the PSS array. The PSS array inherently forms a tree in which each index i is represented by a node whose parent is $\mathsf{pss}[i]$. The root is the artificial index 0, which is parent of all indices that do not have a PSS (see Figure 1b for an example).



(a) Lyndon array, NSS array, and maximal Lyndon words at all indices of $\mathcal{S}$.

(b) PSS array, PSS tree, and BPS representation of the PSS tree of $\mathcal{S}$. (Best viewed in color.)

Figure 1 Data structures for $\mathcal{S} = \texttt{northamerica}$.

▶ **Definition 5** (Previous-Smaller-Suffix Tree $\mathcal{T}_{\mathsf{pss}}$). *Let $\mathcal{S}$ be a string of length n. The* previous-smaller-suffix tree (PSS tree) *of $\mathcal{S}$ is an ordinal tree $\mathcal{T}_{\mathsf{pss}}$ with nodes $[0, n]$ and root 0. For $i \in [1, n]$, we define* $\textsc{Parent}(i) = \mathsf{pss}[i]$. *The children are arranged in ascending order, i.e. if i is a left-side sibling of j, then $i < j$ holds.*

The PSS tree is highly similar to the Left-to-Right-Minima (LRM) tree [3, 9, 22], which we will briefly explain now. Given an array $A[1, n]$ with artificial minimum $A[0] = -\infty$, let $\mathsf{psv}[i] = \max\{j \mid j \in [0, i) \wedge A[j] < A[i]\}$ be the index of the *previous smaller value (PSV)* of $A[i]$. The LRM tree of A is an ordinal tree in which each index i is a child of $\mathsf{psv}[i]$, and the children are ordered ascendingly (i.e. the only difference to the PSS tree is that we consider previous smaller *values* instead of previous smaller *suffixes*). If A is the inverse suffix array of $\mathcal{S}$, then by definition of the inverse suffix array we have $\forall i, j \in [1, n] : \mathcal{S}_i \prec \mathcal{S}_j \Leftrightarrow A[i] < A[j]$. It follows that the PSS tree of a string is identical to the LRM tree of its inverse suffix array. Consequently, an important property of the LRM tree also applies to the PSS tree:

▶ **Corollary 6** (Lemma 1 [9]). *The nodes of the PSS tree directly correspond to the preorder-numbers, i.e. node i has preorder-number i (if the first preorder-number is 0).*

The corollary allows us to simulate the NSS array with the PSS tree:

▶ **Lemma 7.** *Given the PSS tree, NSS array and Lyndon array of the same string we have* $\mathsf{nss}[i] = i + \textsc{SubtreeSize}(i)$ *and thus* $\lambda[i] = \textsc{SubtreeSize}(i)$.[1]

Proof. Since the nodes directly correspond to the preorder-numbers (Corollary 6), it follows that the descendants of i form a consecutive interval $(i, r]$. Since $i + \textsc{SubtreeSize}(i) = i + (r - i + 1) = r + 1$ holds, we only have to show $\mathsf{nss}[i] = r + 1$. Assume $r = n$, then there is no index larger than i that is not a descendant of i. Clearly, in this case i does not have an NSS, and thus it follows $\mathsf{nss}[i] = n + 1 = r + 1$. Assume $r < n$ instead, then $\mathcal{S}_{r+1}$ must be lexicographically smaller than all suffixes that begin at positions from $[i, r]$, since otherwise $r + 1$ would be a descendant of i. Therefore, $\mathcal{S}_{r+1}$ is the first suffix that starts right of i and is lexicographically smaller than $\mathcal{S}_i$, which means $\mathsf{nss}[i] = r + 1$. ◀

[1] $\textsc{SubtreeSize}(i)$ denotes the number of nodes in the subtree that is rooted in i, including i itself.

3.1 Storing the PSS Tree as a BPS

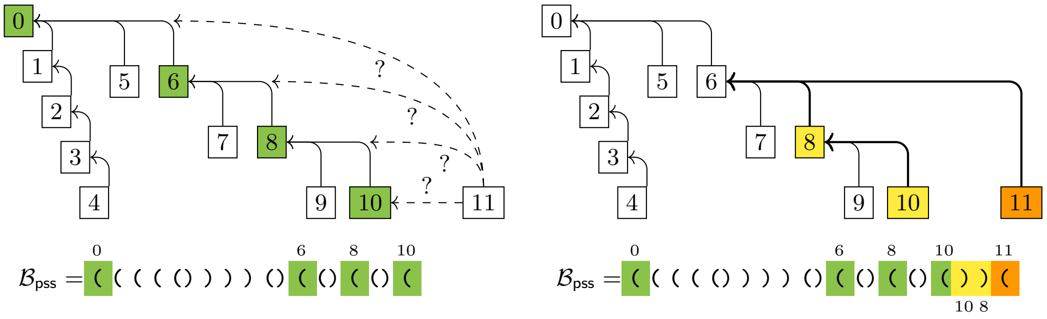
We store the PSS tree as a balanced parentheses sequence (BPS, [21]) of length $2n+2$, which takes $2n+2$ bits. Note that this is less than the $\approx 2.54n$ bits that are necessary to encode previous and next smaller *values* [10] because different suffixes of a text cannot be equal. As a shorthand for the BPS of the PSS tree we write $\mathcal{B}_{\textsf{pss}}$. The sequence is algorithmically defined by a preorder-traversal of the PSS tree, where we write an opening parenthesis whenever we *walk down* an edge, and a closing one whenever we *walk up* an edge. An example is provided in Figure 1b. Note that the BPS of the PSS tree is identical to the succinct Lyndon array presentation from [19]. While the BPS itself does not support fast tree operations, it can be used to construct the data structure from [22], which takes $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ bits of working space. This data structure is of size $2n + \mathcal{O}(n/\lg^c n)$ bits (for any $c \in \mathbb{N}^+$ of our choice) and supports $\textsc{Parent}(\cdot)$ and $\textsc{SubtreeSize}(\cdot)$ operations in $\mathcal{O}(c^2)$ time, and thus allows us to simulate access to the Lyndon array in $\mathcal{O}(c^2)$ time using Lemma 7.

Operations on a BPS Prefix. Since we will be building $\mathcal{B}_{\textsf{pss}}$ from left to right, at any given point of the algorithm execution we know a prefix of $\mathcal{B}_{\textsf{pss}}$. It is crucial that we maintain support for the following queries in constant time:

- Given the index o_i of an opening parenthesis in $\mathcal{B}_{\textsf{pss}}$, determine the node i that belongs to the parenthesis. We have $i = \text{rank}_{\text{open}}(o_i) - 1$, where $\text{rank}_{\text{open}}(o_i)$ is the number of opening parentheses in $\mathcal{B}_{\textsf{pss}}[1..o_i]$.
- Given a preorder-number i, find the index o_i of the corresponding opening parenthesis in $\mathcal{B}_{\textsf{pss}}$. We have $o_i = \text{select}_{\text{open}}(i) = \min\{o \mid \text{rank}_{\text{open}}(o) > i\}$.
- Given an integer $k \geq 1$, find the index $o_{\text{uncl}(k)} = \text{select}_{\text{uncl}}(k)$ of the k-th *unclosed* parenthesis in $\mathcal{B}_{\textsf{pss}}$. An opening parenthesis is called unclosed, if we have not written the matching closing parenthesis yet. For example, there are five opening parentheses in (()((), but only the first and the third one are unclosed.

There are support data structures of size $\mathcal{O}(n \lg\lg n/\lg n)$ bits that answer $\text{rank}_{\text{open}}$ and $\text{select}_{\text{open}}$ queries in constant time [16]. Since these data structures can be constructed in linear time by scanning the BPS from left to right, clearly we can maintain them with no significant time overhead when writing the BPS in an append-only manner. The structure for $\text{select}_{\text{uncl}}$ is a simple modification of the structure for $\text{select}_{\text{open}}$. Consider the (not necessarily balanced) parentheses sequence $\hat{\mathcal{B}}$ with $\hat{\mathcal{B}}[i] = ($, iff $\mathcal{B}_{\textsf{pss}}[i]$ is an unclosed parenthesis, and otherwise $\hat{\mathcal{B}}[i] =)$. Clearly, answering $\text{select}_{\text{uncl}}$ on $\mathcal{B}_{\textsf{pss}}$ is equivalent to answering $\text{select}_{\text{open}}$ on $\hat{\mathcal{B}}$. Thus, if we construct the $\text{select}_{\text{open}}$ data structure by Golynski [16, Section 2.1] for $\hat{\mathcal{B}}$, then we already have a working index for $\text{select}_{\text{uncl}}$ on $\mathcal{B}_{\textsf{pss}}$. However, this approach comes at the cost of additional $2n$ bits of space because we need to explicitly store $\hat{\mathcal{B}}$. In the following paragraph, we outline how to modify the index such that queries can be answered directly on $\mathcal{B}_{\textsf{pss}}$, i.e. without $\hat{\mathcal{B}}$. The reader should be familiar with [16, Section 2.1].

Assume that we want to answer $\text{select}_{\text{open}}(i)$ on $\hat{\mathcal{B}}$. Golynski's index conceptually splits $\hat{\mathcal{B}}$ into chunks of size $\lg n - 3\lg\lg n$. At the time we actually need to access $\hat{\mathcal{B}}$, we have already identified the chunk $\hat{C} = \hat{\mathcal{B}}[c_x..c_x + \lg n - 3\lg\lg n)$ and the value p such that the i-th opening parenthesis in $\hat{\mathcal{B}}$ is exactly the p-th opening parenthesis in $\hat{C}$. Answering the query is realized by simply retrieving the index j of the p-th opening parenthesis within $\hat{C}$ from a precomputed lookup table. Then, the result of the query is $c_x + j - 1$. We can answer the query without $\hat{\mathcal{B}}$, if we retrieve the chunk $C = \mathcal{B}_{\textsf{pss}}[c_x..c_x + \lg n - 3\lg\lg n)$ directly from $\mathcal{B}_{\textsf{pss}}$ (instead of retrieving the chunk $\hat{C}$ from $\hat{\mathcal{B}}$). We only need to change the precomputed lookup table such that it returns the index of the p-th unclosed parenthesis within the chunk instead of the index of the p-th opening parenthesis.


0
1
2
3
4
5
6
7
8
9
10
11
?
?
?
?
0 6 8 10
$\mathcal{B}_{\mathsf{pss}}$ = ((((( )))) ( ) (( ) (( ) (
0 6 8 10 11
$\mathcal{B}_{\mathsf{pss}}$ = ((((( )))) ( ) (( ) (( ) ( )) (
10 8

(a) Before.

(b) After.

Figure 2 The partial PSS tree before and after processing index 11 of $\mathcal{S} = \texttt{northamerica}$ during the execution of Algorithm 1. We have $p_1 = 10$, $p_2 = 8$, $p_3 = 6$, $p_4 = 0$, and $p_m = p_3$. (Best viewed in color.)

Appending parentheses to $\mathcal{B}_{\mathsf{pss}}$ may invalidate parts of the support data structure for $\text{select}_{\text{uncl}}$. The reason for this is that we essentially emulate $\text{select}_{\text{uncl}}$ on $\mathcal{B}_{\mathsf{pss}}$ by answering $\text{select}_{\text{open}}$ on $\hat{\mathcal{B}}$. Appending a closing parenthesis to $\mathcal{B}_{\mathsf{pss}}$ not only translates to appending a closing parenthesis to $\hat{\mathcal{B}}$, but also means that we have to replace the rightmost opening parenthesis in $\hat{\mathcal{B}}$ with a closing one. Thus, we may have to change previously computed parts of the support data structure. This can easily be handled without significant time overhead by only periodically updating the data structure, and naively keeping track of newly appended parentheses in between updates. We omit the details.

4 Constructing the PSS Tree

In this section we introduce our construction algorithm for the BPS of the PSS tree, which processes the indices of the input text in left-to-right order. Processing index i essentially means that we attach i to a partial PSS tree that is induced by the nodes from $[0, i)$. An example is provided in Figure 2a. But how can we efficiently determine i's parent $\mathsf{pss}[i]$? Consider the nodes on the rightmost path of the partial tree, which starts at $i - 1$ and ends at the root 0. We call the set of these nodes *PSS closure* $\mathcal{P}_{i-1}$ of $i - 1$ because it contains exactly the nodes that can be obtained by repeated application of the PSS function on $i - 1$. For $j \in [1, n]$ we recursively define $\mathcal{P}_0 = \{0\}$ and $\mathcal{P}_j = \{j\} \cup \mathcal{P}_{\mathsf{pss}[j]}$. Interestingly, $\mathsf{pss}[i]$ is a member of $\mathcal{P}_{i-1}$:

▶ **Lemma 8.** *For any index* $i \in [1, n]$ *we have* $\mathsf{pss}[i] = \max\{j \mid j \in \mathcal{P}_{i-1} \wedge \mathcal{S}_j \prec \mathcal{S}_i\}$.

Proof. If we show $\mathsf{pss}[i] \in \mathcal{P}_{i-1}$, then the correctness of the lemma follows from Definition 2. Assume $\mathsf{pss}[i] \notin \mathcal{P}_{i-1}$, then there is some index $j \in \mathcal{P}_{i-1}$ with $\mathsf{pss}[i] \in (\mathsf{pss}[j], j)$. By Definition 2, this implies $\mathcal{S}_{\mathsf{pss}[i]} \succ \mathcal{S}_j$. However, we also have $j \in (\mathsf{pss}[i], i)$, which leads to the contradiction $\mathcal{S}_{\mathsf{pss}[i]} \prec \mathcal{S}_j$. ◀

Let $p_1 = i - 1, p_2, \ldots, p_k = 0$ be the elements of $\mathcal{P}_{i-1}$ in descending order, then it follows from Lemma 8 that there is some $m \in [1, k]$ with $\mathsf{pss}[i] = p_m$, i.e. node i has to become a child of p_m in the partial PSS tree. In terms of the BPS, we have to append $m - 1$ closing parentheses to the BPS prefix. Then, we can simply write the opening parenthesis of node i. Once again, an example is provided in Figure 2b.

Algorithm 1 BuildPssBps.

```
Input: String $\mathcal{S}$ of length $n$
Output: BPS of the PSS Tree of $\mathcal{S}$
1: $\mathcal{B}_{\mathsf{pss}} \leftarrow$ (                                        ▷ Open node 0
2: for $i = 1$ to $n$ do
3:     Let $\mathcal{P}_{i-1} = \{p_1, \dots, p_k\}$ with $\mathsf{pss}[p_x] = p_{x+1}$
4:     Determine $p_m = \mathsf{pss}[i]$
5:     Append $m-1$ closing parentheses to $\mathcal{B}_{\mathsf{pss}}$        ▷ Close nodes $p_1, \dots, p_{m-1}$
6:     Append one opening parenthesis to $\mathcal{B}_{\mathsf{pss}}$          ▷ Open node $i$
7: Append $|\mathcal{P}_n|$ closing parentheses to $\mathcal{B}_{\mathsf{pss}}$           ▷ Close rightmost path
```

Our construction algorithm for $\mathcal{B}_{\mathsf{pss}}$ is based on this simple idea, as outlined by Algorithm 1. Initially, the BPS only contains the opening parenthesis of the root 0 (line 1). Then, whenever we process an index i, we use $\mathcal{P}_{i-1}$ to determine p_m (lines 3–4) and extend $\mathcal{B}_{\mathsf{pss}}$ by appending $m-1$ closing parentheses and one opening one (lines 5–6). Finally, once all nodes have been added to the PSS tree, we only have to close all remaining unclosed parentheses (line 7). The algorithm has two black boxes: How do we determine $\mathcal{P}_{i-1}$ (line 3), and how do we use it to find p_m (line 4)? The first question is easily answered, since the operations that we support on the BPS prefix at all times (see Section 3.1) are already sufficient to access each element of $\mathcal{P}_{i-1}$ in constant time. Let $p_1, \dots, p_k$ be exactly these elements in descending order. As explained earlier, they directly correspond to the unclosed parentheses of the BPS prefix, such that p_k corresponds to the leftmost unclosed parenthesis, and p_1 to the rightmost one. Therefore, we have $p_x = \mathrm{rank}_{\mathrm{open}}(\mathrm{select}_{\mathrm{uncl}}(k - x + 1)) - 1$. It remains to be shown how to efficiently find p_m.

4.1 Efficiently Computing p_m

Consider the following naive approach for computing p_m: Iterate over the indices $p_1, \dots, p_k$ in descending order (i.e. p_1 first, p_k last). For each index p_x, evaluate whether $\mathcal{S}_{p_x} \prec \mathcal{S}_i$ holds. As soon as this is the case, we have found p_m. The cost of this approach is high: A naive suffix comparison between $\mathcal{S}_{p_x}$ and $\mathcal{S}_i$ takes $\textsc{lce}(p_x, i) + 1$ individual character comparisons, which means that we spend $\mathcal{O}(m + \sum_{x=1}^{m} \textsc{lce}(p_x, i))$ time to determine m. However, the following property will allow us to decrease this time bound significantly:

▶ **Corollary 9** (Bitonic LCE Values). *Let $p_1, \dots, p_k$ be exactly the elements of $\mathcal{P}_{i-1}$ in descending order and let $p_m = \mathsf{pss}[i]$. Furthermore, let $\ell_x = \textsc{lce}(p_x, i)$ for all $x \in [1, k]$. We have $\ell_1 \leq \ell_2 \leq \dots \leq \ell_{m-1}$ as well as $\ell_m \geq \ell_{m+1} \geq \dots \geq \ell_k$.*

Proof. Follows from $\mathcal{S}_{p_1} \succ \ldots \succ \mathcal{S}_{p_{m-1}} \succ \mathcal{S}_i \succ \mathcal{S}_{p_m} \succ \ldots \succ \mathcal{S}_{p_k}$ and simple properties of the lexicographical order. ◀

From now on, we continue using the notation $\ell_x = \textsc{lce}(p_x, i)$ from the corollary. Note that the longest LCE between i and any of the p_x occurs either with p_m or with p_{m-1}. Let $\ell_{\max} = \max(\ell_{m-1}, \ell_m)$ be this largest LCE value, then our more sophisticated approach for determining m only takes $\mathcal{O}(m + \ell_{\max})$ time. It consists of two steps: First, we determine a candidate interval $(u, w] \subseteq [1, k]$ of size at most $\ell_{\max}$ that contains m. In the second step we gradually narrow down the borders of the candidate interval until the exact value of m is known.

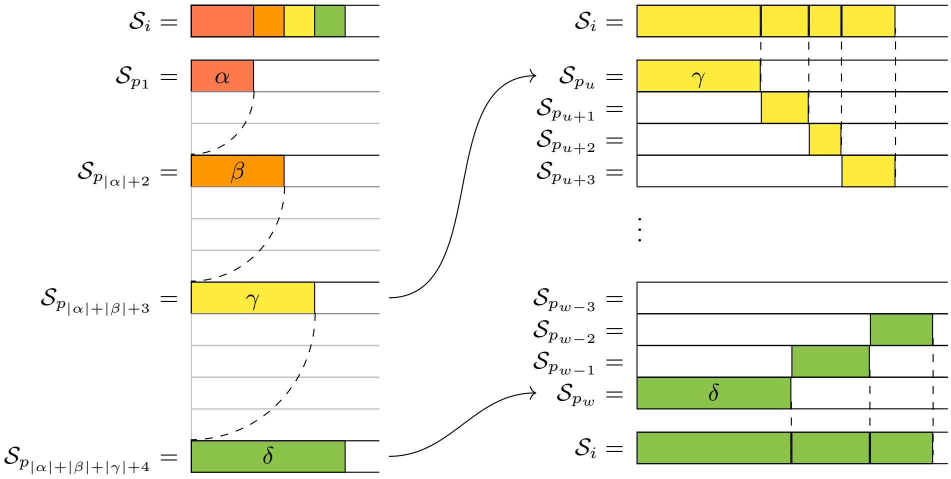

$\mathcal{S}_i =$
$\mathcal{S}_{p_1} =$
α
$\mathcal{S}_{p_{|\alpha|+2}} =$
β
$\mathcal{S}_{p_{|\alpha|+|\beta|+3}} =$
γ
$\mathcal{S}_{p_{|\alpha|+|\beta|+|\gamma|+4}} =$
δ
$\mathcal{S}_i =$
$\mathcal{S}_{p_u} =$
γ
$\mathcal{S}_{p_{u+1}} =$
$\mathcal{S}_{p_{u+2}} =$
$\mathcal{S}_{p_{u+3}} =$
$\mathcal{S}_{p_{w-3}} =$
$\mathcal{S}_{p_{w-2}} =$
$\mathcal{S}_{p_{w-1}} =$
$\mathcal{S}_{p_w} =$
δ
$\mathcal{S}_i =$

Figure 3 Matching character comparisons when determining p_m. On the left we have the suffix $\mathcal{S}_i$ as well as $\mathcal{S}_{p_1}, \mathcal{S}_{p_2}, \ldots, \mathcal{S}_{p_w}$, which are relevant for the first step. Each prefix $\alpha, \beta, \gamma, \delta$ highlights the LCP between the respective suffix $\mathcal{S}_{p_x}$ and $\mathcal{S}_i$. On the right side we have the suffixes $\mathcal{S}_{p_u}, \mathcal{S}_{p_{u+1}}, \ldots, \mathcal{S}_{p_w}$, which are relevant for the second step. (Best viewed in color.)

Step 1: Find a candidate interval. Our goal is to find $(u, w] = (u, u + \ell_u + 1]$ with $m \in (u, w]$. Initially, we naively compute $\ell_1 = \text{LCE}(p_1, i)$, allowing us to evaluate $\mathcal{S}_{p_1} \prec \mathcal{S}_i$ in constant time. If this holds, then we have $m = 1$ and no further steps are necessary. Otherwise, let $u \leftarrow 1$ and (i) let $w \leftarrow u + \ell_u + 1$. We already know that $u < m$ holds. Now we have to evaluate if $m \le w$ also holds. Therefore, we compute $\ell_w = \text{LCE}(p_w, i)$ naively, which allows us to check in constant time if $\mathcal{S}_{p_w} \prec \mathcal{S}_i$ and decide if $m \le w$ holds. If this is not the case, then we assign $u \leftarrow w$ as well as $\ell_u \leftarrow \ell_w$ and continue at (i). If however $\mathcal{S}_{p_w} \prec \mathcal{S}_i$ holds, then we have $m \le w$ and therefore $m \in (u, w]$. Figure 3 (left) outlines the procedure.

Step 2: Narrow down $(u, w]$ to the exact value of m. Now we gradually tighten the borders of the candidate interval. If ℓ_u is smaller than ℓ_w, then we try to increase u by one. Otherwise, we try to decrease w by one.

Assume that we have $\ell_u < \ell_w$, then it follows from Corollary 9 that $\ell_{u+1} \ge \ell_u$ holds. Therefore, when computing ℓ_{u+1} we can simply skip the first ℓ_u character comparisons. Now we use ℓ_{u+1} to evaluate in constant time if $\mathcal{S}_{p_{u+1}} \succ \mathcal{S}_i$ holds. If that is the case, then we have $u + 1 < m$ and thus we can assign $u \leftarrow u + 1$ and start Step 2 from the beginning. If however $\mathcal{S}_{p_{u+1}} \prec \mathcal{S}_i$ holds, then we have $m = u + 1$ and no further steps are necessary. In case of $\ell_u \ge \ell_w$ we proceed analogously. Once again, Figure 3 (right) visualizes the procedure.

Time Complexity. Step 1 is dominated by computing LCE values. Determining the final LCE value ℓ_w takes ℓ_w+1 individual character comparisons and thus $\Theta(\ell_w+1)$ time. Whenever we compute any previous value of ℓ_w, we increase w by $\ell_w + 1$ afterwards. Therefore, the time for computing all LCE values is bound by $\Theta(w + \ell_w) = \Theta(u + \ell_u + \ell_w) \subseteq \mathcal{O}(m + \ell_{\max})$. Since initially $(u, w]$ has size at most $\ell_{\max}$, we call Step 2 at most $\mathcal{O}(\ell_{\max})$ times. With every call we increase ℓ_u or ℓ_w by exactly the number of matching character comparisons that we perform. Therefore, the total number of matching character comparisons is bound by $2\ell_{\max}$.

Thus, the total time needed for Step 2 is bound by $\mathcal{O}(\ell_{\max})$. In sum, processing index i takes $\mathcal{O}(m + \ell_{\max})$ time. For the total processing time of all indices (and thus the execution time of Algorithm 1) we get:

$$\begin{aligned}
&\sum_{i=1}^{n} \mathcal{O}(\ \overbrace{|\mathcal{P}_{i-1} \cap [\mathsf{pss}[i], i]|}^{m}\ ) + \sum_{i=1}^{n} \mathcal{O}(\ \overbrace{\max_{p_x \in \mathcal{P}_{i-1}} \text{LCE}(p_x, i)}^{\ell_{\max}}\ )\\
= \quad &\mathcal{O}(n) \qquad\qquad\qquad + \qquad \mathcal{O}(n^2)
\end{aligned}$$

(The $\mathcal{O}(m)$-terms sum to $\mathcal{O}(n)$ since $m-1$ is exactly the number of closing parentheses that we write while processing i, and there are exactly $n+1$ closing parentheses in the entire BPS.) As it appears, the total time bound of the algorithm is still far from linear time. However, it is easy to identify the crucial time component that makes the algorithm too expensive. From now on we call the $\mathcal{O}(m)$ term of the processing time *negligible*, while the $\mathcal{O}(\ell_{\max})$ term is called *critical*.

Clearly, if we could somehow remove the critical terms, we would already achieve linear time. There exists a variety of data structures that could help us to achieve this goal by accelerating suffix comparisons, e.g. the (compressed or sparse) suffix tree, the (compressed) suffix array, or dedicated data structures for fast LCE queries. However, all of theses data structures either require more than $\mathcal{O}(n)$ bits of construction space, or more than $\mathcal{O}(n)$ construction time, or they are non-deterministic, or their efficiency depends on the alphabet or the compressability of the text. For example, there exists a linear time construction algorithm for the sparse suffix tree [15], but it is non-deterministic. This motivates the techniques that we describe in the following sections, which directly remove the critical terms without relying on any of the aforementioned data structures. This way, the execution time of Algorithm 1 decreases to $\mathcal{O}(n)$, while the additional working space remains unchanged.

5 Achieving Linear Time

The critical time component for processing index i is $\ell_{\max} = \max_{p_x \in \mathcal{P}_{i-1}} \text{LCE}(p_x, i)$. When processing i with the technique from Section 4.1, we inherently find out the exact value of $\ell_{\max}$, and we also discover the index $p_{\max}$ for which we have $\text{LCE}(p_{\max}, i) = \ell_{\max}$. From now on, we simply use $\ell = \ell_{\max}$ and $j = p_{\max}$. While discovering a large LCE value ℓ is costly, it yields valuable structural information about the input text: There is a repeating substring of length ℓ with occurrences $\mathcal{S}[j..j+\ell)$ and $\mathcal{S}[i..i+\ell)$. Intuitively, there is also a large repeating structure in the PSS tree, and consequently a repeating substring in $\mathcal{B}_{\mathsf{pss}}$. This motivates the techniques shown in this section, which conceptually alter Algorithm 1 as follows: Whenever we finish processing an index i with critical cost ℓ, we skip the next $\Omega(\ell)$ iterations of the loop by simply extending the BPS prefix with the copy of an already computed part, which means that the amortized critical cost per index becomes constant.

Depending on j and ℓ we choose either the *run extension* (Section 5.1) or the *amortized look-ahead* (Section 5.2) to perform the extension. Algorithm 2 outlines our final construction algorithm on a higher level, and complements the written description by showing when the special cases arise. Before going into detail, we point out that $\mathcal{S}[j..i)$ is a Lyndon word. As mentioned earlier, it follows from Corollary 9 that j equals p_m or p_{m-1}. Since i is the first node that is not a descendant of p_{m-1}, we have $\mathsf{nss}[p_{m-1}] = i$. Therefore, if $j = p_{m-1}$ holds, we have $\mathsf{nss}[j] = i$, which by definition implies that $\mathcal{S}[j..i)$ is a Lyndon word. If however $j = p_m = \mathsf{pss}[i]$ holds, then $\mathcal{S}[j..i)$ is a Lyndon word because of Lemma 4.

Algorithm 2 BuildPssBpsLinear.

```
Input: String $\mathcal{S}$ of length $n$
Output: BPS of the PSS Tree of $\mathcal{S}$
 1: $\mathcal{B}_{\mathsf{pss}} \leftarrow ($
 2: for $i = 1$ to $n$ do
 3:     Let $\mathcal{P}_{i-1} = \{p_1, \ldots, p_k\}$ with $\mathsf{pss}[p_x] = p_{x+1}$
 4:     Determine $p_m = \mathsf{pss}[i]$,
            using the technique from Section 4.1, causing critical cost $\ell$ and
            discovering the index $j$ with $\textsc{lce}(j, i) = \ell$ as described in the
            beginning of Section 5.
 5:     Append $m - 1$ closing parentheses to $\mathcal{B}_{\mathsf{pss}}$
 6:     Append one opening parenthesis to $\mathcal{B}_{\mathsf{pss}}$
        (For any $x$, let $o_x$ be the opening parenthesis of node $x$.)
 7:     if $\ell \geq 2(i - j)$ then
 8:         Apply the run extension as described in Section 5.1.
                Let $t = \lfloor \ell/(i-j) \rfloor + 1$. Take $\mathcal{B}_{\mathsf{pss}}(o_j..o_i]$ and append it $(t-2)$
                times to $\mathcal{B}_{\mathsf{pss}}$. Continue in line 2 with $i \leftarrow i + (t-2) \cdot (i-j)$.
 9:     else
10:         Apply the amortized look-ahead as described in Section 5.2.
                Using Lemma 13, find the largest value $\chi \in [1, \lfloor \ell/4 \rfloor]$ that sat-
                isfies $\mathcal{B}_{\mathsf{pss}}[o_j..o_{j+\chi-1}] = \mathcal{B}_{\mathsf{pss}}[o_i..o_{i+\chi-1}]$, and append a copy of
                $\mathcal{B}_{\mathsf{pss}}(o_j..o_{j+\chi-1}]$ to $\mathcal{B}_{\mathsf{pss}}$. Continue in line 2 with $i \leftarrow i + \chi$. If
                $\chi < \lfloor \ell/4 \rfloor$, then iteration $i + \chi$ will automatically skip additional
                $\Omega(\ell)$ iterations by using the run extension.
11: Append $|\mathcal{P}_n|$ closing parentheses to $\mathcal{B}_{\mathsf{pss}}$
```

5.1 Run Extension

We apply the run extension iff we have $\ell \geq 2(i - j)$. It is easy to see that in this case $\mathcal{S}[j..j+\ell)$ and $\mathcal{S}[i..i+\ell)$ overlap such that the Lyndon word $\mu = \mathcal{S}[j..i)$ repeats itself at least three times, starting at index j. We call the substring $\mathcal{S}[j..i+\ell)$ *Lyndon run with period* $|\mu|$. The number of *repetitions* is $t = \lfloor \ell / |\mu| \rfloor + 1 \geq 3$, and the starting positions of the repetitions are $r_1, \ldots, r_t$ with $r_1 = j$, $r_2 = i$, and generally $r_x = r_{x-1} + |\mu|$. In a moment we will show that in this particular situation the following lemma holds:

▶ **Lemma 10.** *Let o_x be the index of the opening parenthesis of node x in $\mathcal{B}_{\mathsf{pss}}$. Then we have $\mathcal{B}_{\mathsf{pss}}[o_{r_1}..o_{r_2}] = \mathcal{B}_{\mathsf{pss}}[o_{r_2}..o_{r_3}] = \cdots = \mathcal{B}_{\mathsf{pss}}[o_{r_{t-1}}..o_{r_t}]$.*

Expressed less formally, each repetition of μ – except for the last one – induces the same substring in the BPS. Performing the run extension is as easy as taking the already written substring $\mathcal{B}_{\mathsf{pss}}(o_{r_1}..o_{r_2}] = \mathcal{B}_{\mathsf{pss}}(o_j..o_i]$, and appending it $t - 2$ times to $\mathcal{B}_{\mathsf{pss}}$. Afterwards, the last parenthesis that we have written is the opening parenthesis of node r_t, and we continue the execution of Algorithm 1 with iteration $r_t + 1$. Thus, we have skipped the processing of $r_t - i$ indices. Since

$$r_t - i \;=\; (t-2) \cdot |\mu| \;\geq\; \frac{(t-2) \cdot |\mu|}{t \cdot |\mu|} \cdot \ell \;\geq\; \frac{1}{3} \cdot \ell \;=\; \Omega(\ell) \;,$$

it follows that the average critical cost per index from $[i, r_t]$ is constant.

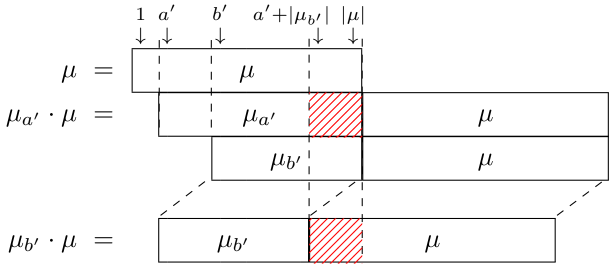
Proving the Lemma. It remains to be shown that Lemma 10 holds. It is sufficient to prove the correctness for $t = 3$, since the correctness for the general case follows by repeatedly applying the lemma with $t = 3$. Therefore, we only have to show $\mathcal{B}_{\mathsf{pss}}[o_{r_1}..o_{r_2}] = \mathcal{B}_{\mathsf{pss}}[o_{r_2}..o_{r_3}]$.

Isomorphic Subtrees. Since μ is a Lyndon word, it is easy to see that the suffixes at the starting positions of repetitions are lexicographically smaller than the suffixes that begin in between the starting positions of repetitions, i.e. we have $\forall x \in (r_1, r_2) : \mathcal{S}_{r_1} \prec \mathcal{S}_x$ and $\forall x \in (r_2, r_3) : \mathcal{S}_{r_2} \prec \mathcal{S}_x$. Consequently, the indices from (r_1, r_2) are descendants of r_1 in the PSS tree, and the indices from (r_2, r_3) are descendants of r_2 in the PSS tree, i.e. each of the intervals $[r_1, r_2)$ and $[r_2, r_3)$ induces a tree.

Next, we show that these trees are actually isomorphic. Clearly, the tree induced by $[r_1, r_2)$ solely depends on the lexicographical order of suffixes that begin within $[r_1, r_2)$, and the tree induced by $[r_2, r_3)$ solely depends on the lexicographical order of suffixes that begin within $[r_2, r_3)$. Assume that the trees are *not* isomorphic, then there must be a suffix comparison that yields different results in each interval, i.e. there must be offsets $a, b \in [0, |\mu|)$ with $a \neq b$ such that $\mathcal{S}_{r_1+a} \prec \mathcal{S}_{r_1+b} \iff \mathcal{S}_{r_2+a} \succ \mathcal{S}_{r_2+b}$ holds. However, this is impossible, as shown by the lemma below.

▶ **Lemma 11.** *For all $a, b \in [0, |\mu|)$ with $a \neq b$ we have $\mathcal{S}_{r_1+a} \prec \mathcal{S}_{r_1+b} \iff \mathcal{S}_{r_2+a} \prec \mathcal{S}_{r_2+b}$.*

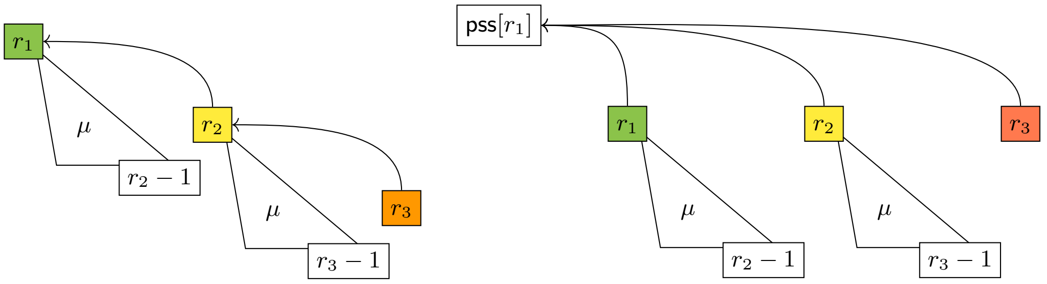
Proof. Assume w.l.o.g. $a < b$, and let $a' = a + 1$ and $b' = b + 1$. We can show that the strings $\mu_{a'} \cdot \mu$ and $\mu_{b'} \cdot \mu$ have a mismatch:



Consider the two hatched areas in the drawing above. The top area highlights the suffix $\mu_{a'+|\mu_{b'}|}$ of μ, which has length $c = |\mu| - (a' + |\mu_{b'}|) + 1$. The bottom area highlights the prefix $\mu[1..c]$ of μ. Since μ is a Lyndon word, there is no proper non-empty suffix of μ that is also a prefix of μ. It follows that the hatched areas cannot be equal, i.e. $\mu_{a'+|\mu_{b'}|} \neq \mu[1..c]$. This guarantees a mismatch between $\mu_{a'} \cdot \mu$ and $\mu_{b'} \cdot \mu$. Therefore, appending an arbitrary string to $\mu_{a'} \cdot \mu$ and $\mu_{b'} \cdot \mu$ does not influence the outcome of a lexicographical comparison. The statement of the lemma directly follows by appending $\mathcal{S}_{r_3}$ and $\mathcal{S}_{r_4}$ respectively:

$$\mu_{a'} \cdot \mu \prec \mu_{b'} \cdot \mu \iff \underbrace{\mu_{a'} \cdot \mu \cdot \mathcal{S}_{r_3}}_{=\ \mathcal{S}_{r_1+a}} \prec \underbrace{\mu_{b'} \cdot \mu \cdot \mathcal{S}_{r_3}}_{=\ \mathcal{S}_{r_1+b}} \iff \underbrace{\mu_{a'} \cdot \mu \cdot \mathcal{S}_{r_4}}_{=\ \mathcal{S}_{r_2+a}} \prec \underbrace{\mu_{b'} \cdot \mu \cdot \mathcal{S}_{r_4}}_{=\ \mathcal{S}_{r_2+b}}$$

◀



(a) Increasing run.

(b) Decreasing run.

Figure 4 The run of the Lyndon word $\mu = \mathcal{S}[r_1, r_2) = \mathcal{S}[r_2, r_3) = \mathcal{S}[r_3, r_3{+}|\mu|)$ induces isomorphic subtrees in the PSS tree. If $\mathcal{S}_{r_1} \prec \mathcal{S}_{r_2}$, then the roots of the subtrees form a chain **(a)**. Otherwise, they are siblings **(b)**.

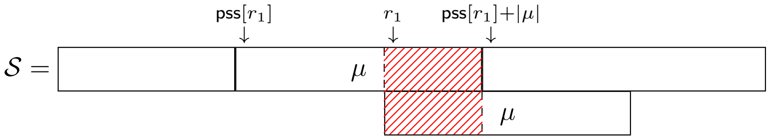
Finally, we show that in the PSS tree the induced isomorphic trees are connected in a way that ultimately implies $\mathcal{B}_{\mathsf{pss}}[o_{r_1}..o_{r_2}] = \mathcal{B}_{\mathsf{pss}}[o_{r_2}..o_{r_3}]$. There are two possible scenarios for this connection, which depend on the so called *direction* of the Lyndon run. We call a run *increasing* iff $\mathcal{S}_{r_1} \prec \mathcal{S}_{r_2}$ holds, and *decreasing* otherwise.

Increasing Runs. First, we focus on increasing runs. It follows from $\mathcal{S}_{r_1} \prec \mathcal{S}_{r_2} \iff \mu \cdot \mathcal{S}_{r_2} \prec \mu \cdot \mathcal{S}_{r_3} \iff \mathcal{S}_{r_2} \prec \mathcal{S}_{r_3}$ that $\mathcal{S}_{r_1} \prec \mathcal{S}_{r_2} \prec \mathcal{S}_{r_3}$. Since μ is a Lyndon word, we have $\forall x \in (r_1, r_2) : \mathcal{S}_{r_2} \prec \mathcal{S}_x$ as well as $\forall x \in (r_2, r_3) : \mathcal{S}_{r_3} \prec \mathcal{S}_x$. Therefore, we have $\mathsf{pss}[r_2] = r_1$ and $\mathsf{pss}[r_3] = r_2$, and the isomorphic subtrees are connected as visualized in Figure 4a. It is easy to see that a preorder-traversal from r_1 to r_2 yields the same sequence of parentheses as a preorder-traversal from r_2 to r_3. Therefore we have $\mathcal{B}_{\mathsf{pss}}[o_{r_1}..o_{r_2}] = \mathcal{B}_{\mathsf{pss}}[o_{r_2}..o_{r_3}]$, which means that Lemma 10 holds for increasing runs.

Decreasing Runs. With the same argument as for increasing runs, we have $\mathcal{S}_{r_1} \succ \mathcal{S}_{r_2} \succ \mathcal{S}_{r_3}$ in decreasing runs. We also have $\forall x \in (r_1, r_2) : \mathcal{S}_{r_2} \prec \mathcal{S}_x$ as well as $\forall x \in (r_2, r_3) : \mathcal{S}_{r_3} \prec \mathcal{S}_x$, which means that $\mathsf{pss}[r_2] \leq \mathsf{pss}[r_1]$ and $\mathsf{pss}[r_3] \leq \mathsf{pss}[r_1]$ hold. In Lemma 12 we will show that in fact $\mathsf{pss}[r_1] = \mathsf{pss}[r_2] = \mathsf{pss}[r_3]$ holds, such that the isomorphic subtrees are connected as visualized in Figure 4b. A preorder-traversal from r_1 to r_2 yields the same sequence of parentheses as a preorder-traversal from r_2 to r_3. Therefore we have $\mathcal{B}_{\mathsf{pss}}[o_{r_1}..o_{r_2}] = \mathcal{B}_{\mathsf{pss}}[o_{r_2}..o_{r_3}]$, which means that Lemma 10 holds for decreasing runs.

▶ **Lemma 12.** *In decreasing runs we have* $\mathsf{pss}[r_1] = \mathsf{pss}[r_2] = \mathsf{pss}[r_3]$.

Proof. As explained previously, we have $\mathsf{pss}[r_2] \leq \mathsf{pss}[r_1]$ and $\mathsf{pss}[r_3] \leq \mathsf{pss}[r_1]$, and thus only need to show $\mathcal{S}_{\mathsf{pss}[r_1]} \prec \mathcal{S}_{r_2}$ and $\mathcal{S}_{\mathsf{pss}[r_1]} \prec \mathcal{S}_{r_3}$. We will show below that μ cannot be a prefix of $\mathcal{S}_{\mathsf{pss}[r_1]}$, from which the statement of the lemma can be deduced easily since the suffixes $\mathcal{S}_{r_2}$ and $\mathcal{S}_{r_3}$ begin with the prefix μ. Assume for the sake of contradiction that μ is a prefix of $\mathcal{S}_{\mathsf{pss}[r_1]}$. If we also assume $\mathsf{pss}[r_1] + |\mu| > r_1$, we get:



As indicated by the hatched area, this implies that there is a proper non-empty suffix of μ that is also a prefix of μ, which is impossible because μ is a Lyndon word. Thus we have $\mathsf{pss}[r_1] + |\mu| \not> r_1$. Also, we cannot have $\mathsf{pss}[r_1] + |\mu| = r_1$, because then $\mathsf{pss}[r_1]$ would be the starting position of another repetition of μ, which would imply $\mathcal{S}_{\mathsf{pss}[r_1]} \succ \mathcal{S}_{r_1}$. It follows $\mathsf{pss}[r_1] + |\mu| < r_1$, i.e. $\mathsf{pss}[r_1] + |\mu| \in (\mathsf{pss}[r_1], r_1)$ and thus $\mathcal{S}_{\mathsf{pss}[r_1]+|\mu|} \succ \mathcal{S}_{r_1}$. However, this leads to a contradiction:

$$\begin{aligned}\mathcal{S}_{\mathsf{pss}[r_1]} \prec \mathcal{S}_{r_1} &\iff \mu \cdot \mathcal{S}_{\mathsf{pss}[r_1]+|\mu|} \prec \mu \cdot \mathcal{S}_{r_2} \\ &\iff \mathcal{S}_{\mathsf{pss}[r_1]+|\mu|} \prec \mathcal{S}_{r_2} \\ &\underset{\mathcal{S}_{r_1} \succ \mathcal{S}_{r_2}}{\implies} \mathcal{S}_{\mathsf{pss}[r_1]+|\mu|} \prec \mathcal{S}_{r_1}\end{aligned}$$ ◀

5.2 Amortized Look-Ahead

Finally, we show how to amortize the critical cost $\mathcal{O}(\ell)$ of processing index i if the run extension is not applicable, i.e. if we have $\ell < 2(i-j)$. Unfortunately, the trees induced by the nodes from $[j, j+\ell)$ and $[i, i+\ell)$ are not necessarily isomorphic. However, we can still identify a sufficiently large isomorphic structure. In a moment we will show that the following lemma holds:

▶ **Lemma 13.** *Let o_x be the index of the opening parenthesis of node x in $\mathcal{B}_{\mathsf{pss}}$. We either have $\mathcal{B}_{\mathsf{pss}}[o_j..o_{j+\lfloor \ell/4 \rfloor -1}] = \mathcal{B}_{\mathsf{pss}}[o_i..o_{i+\lfloor \ell/4 \rfloor -1}]$, or there is an integer $\chi < \lfloor \ell/4 \rfloor$ with $\mathcal{B}_{\mathsf{pss}}[o_j..o_{j+\chi-1}] = \mathcal{B}_{\mathsf{pss}}[o_i..o_{i+\chi-1}]$ and an index $h \in [i, i+\chi)$ such that $\mathcal{S}[h..i+\ell)$ is a Lyndon run of the Lyndon word $\mathcal{S}[h..i+\chi)$. We can determine which case applies, and also determine the value of χ (if applicable) in $\mathcal{O}(\ell)$ time and $\mathcal{O}(1)$ words of additional space.*

When performing the amortized look-ahead we first determine which case of the lemma applies. Then, if $\mathcal{B}_{\mathsf{pss}}[o_j..o_{j+\lfloor \ell/4 \rfloor -1}] = \mathcal{B}_{\mathsf{pss}}[o_i..o_{i+\lfloor \ell/4 \rfloor -1}]$, we extend the known prefix of the BPS by appending a copy of $\mathcal{B}_{\mathsf{pss}}(o_j..o_{j+\lfloor \ell/4 \rfloor -1}]$, and continue the execution of Algorithm 1 with iteration $i + \lfloor \ell/4 \rfloor$. Since this way we skip the processing of $\lfloor \ell/4 \rfloor - 1 = \Omega(\ell)$ indices, the average critical cost per index from $[i, i + \lfloor \ell/4 \rfloor)$ is constant. If, however, the second case applies, then we determine the value of χ and extend the known prefix of the BPS by appending a copy of $\mathcal{B}_{\mathsf{pss}}(o_j..o_{j+\chi-1}]$, allowing us to continue the execution of Algorithm 1 with iteration $i+\chi$. We know that there is some $h \in [i, i+\chi)$ such that $\mathcal{S}[h..i+\ell)$ is a Lyndon run of the Lyndon word $\mu = \mathcal{S}[h..i+\chi)$. This run might even be longer: Let $\ell' = \textsc{lce}(h, i+\chi)$ (computed naively), then $\mathcal{S}[h..i+\chi+\ell')$ is the longest run of μ that starts at index h. If the run is increasing, then $\mathsf{pss}[i+\chi] = h$ holds (see Section 5.1), and the longest LCE that we discover when processing index $i+\chi$ is ℓ'. If the run is decreasing, then $\mathsf{pss}[i+\chi] = \mathsf{pss}[h]$ holds. Also in this case, the longest LCE that we discover when processing index $i+\chi$ is ℓ', since $\textsc{lce}(\mathsf{pss}[i+\chi], i+\chi)$ is less than $|\mu|$ (see proof of Lemma 12). Therefore, the critical cost of processing index $i+\chi$ will be $\mathcal{O}(\ell')$. However, since the Lyndon run has at least three repetitions, we will also skip the processing of $\Omega(\ell')$ indices by using the run extension. The algorithmic procedure for the second case can be summarized as follows: We process index i with critical cost $\mathcal{O}(\ell)$ and skip $\chi - 1$ indices afterwards. Then we process index $i+\chi$ with critical cost $\mathcal{O}(\ell')$ and skip another $\Omega(\ell')$ indices by using the run extension. Since we have $\ell' = \Omega(\ell)$, the total critical cost is $\mathcal{O}(\ell')$, and the total number of processed or skipped indices is $\Omega(\ell')$. Thus, the average critical cost per index is constant.

Proving Lemma 13. It remains to be shown that Lemma 13 holds. For this purpose, assume $\mathcal{B}_{\mathsf{pss}}[o_j..o_{j+\lfloor \ell/4 \rfloor -1}] \neq \mathcal{B}_{\mathsf{pss}}[o_i..o_{i+\lfloor \ell/4 \rfloor -1}]$. From now on we refer to $\mathcal{B}_{\mathsf{pss}}[o_j..o_{j+\lfloor \ell/4 \rfloor -1}]$ and $\mathcal{B}_{\mathsf{pss}}[o_i..o_{i+\lfloor \ell/4 \rfloor -1}]$ as *left* and *right side*, respectively. Consider the first mismatch between the two, where w.l.o.g. we assume that the mismatch has an opening parenthesis on the left

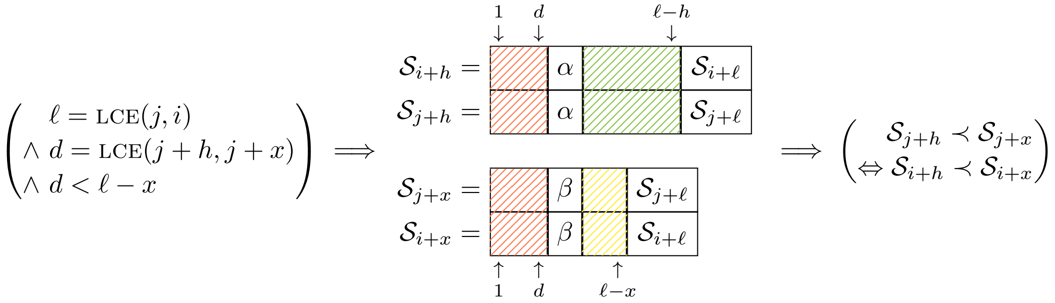


Figure 5 Proving Lemma 13. Equal colors indicate equal substrings. (Best viewed in color.)

side, and a closing one on the right side. On the left side, the opening parenthesis corresponds to a node $j+x$ with $x \in [1, \lfloor \ell/4 \rfloor)$ that is a child of another node $j+h$. Since $\mathcal{S}[j..j+\ell)$ is a Lyndon word, all nodes from $(j, j+\ell)$ are descendants of j. Consequently, we have $h \in [0, x)$. Now we look at the right side: Since we have a closing parenthesis instead of an opening one, we know that $i+x$ is not attached to $i+h$, but to a smaller node, i.e. we have $\mathsf{pss}[i+x] < i+h$. It follows that $\mathcal{S}_{j+h} \prec \mathcal{S}_{j+x}$ and $\mathcal{S}_{i+h} \succ \mathcal{S}_{i+x}$ hold. Let $d = \textsc{lce}(j+h, j+x)$ and assume $d \leq \ell - x$. Then due to $\mathcal{S}_{j+h} \prec \mathcal{S}_{j+x}$ we have $\mathcal{S}[j+h+d] < \mathcal{S}[j+x+d]$. However, since we have $\mathcal{S}[j..j+\ell] = \mathcal{S}[i..i+\ell]$, it follows $\textsc{lce}(i+h, i+x) = d$ and $\mathcal{S}[i+h+d] < \mathcal{S}[i+x+d]$, which contradicts $\mathcal{S}_{i+h} \succ \mathcal{S}_{i+x}$ (see Figure 5). Thus, it holds $d = \textsc{lce}(j+h, j+x) \geq \ell - x$, allowing us to show that $\mathcal{S}[j+h..j+\ell)$ is a Lyndon run with period $x-h$. Since $\mathsf{pss}[j+x] = j+h$ holds, it follows from Lemma 4 that $\mathcal{S}[j+h..j+x)$ is a Lyndon word. Due to $\textsc{lce}(j+h, j+x) > \ell - x \geq 3(\ell/4) \geq 3(x-h)$ we know that the Lyndon word repeats at least four times, and the run extends all the way to the end of $\mathcal{S}[j..j+\ell)$. Note that since the opening parenthesis of node $j+x$ causes the first mismatch between $\mathcal{B}_{\mathsf{pss}}[o_j..o_{j+\lfloor \ell/4 \rfloor - 1}]$ and $\mathcal{B}_{\mathsf{pss}}[o_i..o_{i+\lfloor \ell/4 \rfloor - 1}]$, we have $\mathcal{B}_{\mathsf{pss}}[o_j..o_{j+x-1}] = \mathcal{B}_{\mathsf{pss}}[o_i..o_{i+x-1}]$. Therefore, $\chi \leftarrow x$ already satisfies Lemma 13.

Finally, we explain how to determine $\chi = x$ in $\mathcal{O}(\ell)$ time. As described above, if $\mathcal{B}_{\mathsf{pss}}[o_j..o_{j+\lfloor \ell/4 \rfloor - 1}] \neq \mathcal{B}_{\mathsf{pss}}[o_i..o_{i+\lfloor \ell/4 \rfloor - 1}]$, then there is some offset $h < \lfloor \ell/4 \rfloor$ such that $\mathcal{S}[j+h..j+\ell)$ is a Lyndon run of at least four repetitions of a Lyndon word μ. Consequently, $\mathcal{S}[j + \lfloor \ell/4 \rfloor..j+\ell)$ has the form $\mathrm{suf}(\mu) \cdot \mu^t \cdot \mathrm{pre}(\mu)$ with $t \geq 2$, where $\mathrm{suf}(\mu)$ and $\mathrm{pre}(\mu)$ are a proper suffix and a proper prefix of μ. A string of this form is called *extended Lyndon run*. In Section 5.2.1 we propose an algorithm that checks whether or not $\mathcal{S}[j + \lfloor \ell/4 \rfloor..j+\ell)$ is an extended Lyndon run in $\mathcal{O}(\ell)$ time and constant additional space. If $\mathcal{S}[j + \lfloor \ell/4 \rfloor..j+\ell)$ is not an extended Lyndon run, then we have $\mathcal{B}_{\mathsf{pss}}[o_j..o_{j+\lfloor \ell/4 \rfloor - 1}] = \mathcal{B}_{\mathsf{pss}}[o_i..o_{i+\lfloor \ell/4 \rfloor - 1}]$ and no further steps are needed to satisfy Lemma 13. Otherwise, the algorithm from Section 5.2.1 also provides the period $|\mu|$ of the run, as well as $|\mathrm{suf}(\mu)|$. In this case, we try to extend the extended Lyndon run to the left: We are now not only considering $\mathcal{S}[j + \lfloor \ell/4 \rfloor..j+\ell)$, but $\mathcal{S}[j..j+\ell)$. We want to find the leftmost index $j+h$ that is the starting position of a repetition of μ. Given $|\mu|$ and $|\mathrm{suf}(\mu)|$, this can be done naively by scanning $\mathcal{S}[j..j+\lfloor \ell/4 \rfloor]$ from right to left, which takes $\mathcal{O}(\ell)$ time. If we have $h \geq \lfloor \ell/4 \rfloor - |\mu|$, then the first case of Lemma 13 applies and no further steps are necessary. Otherwise, we let $\chi \leftarrow h + |\mu|$. This concludes the proof of Lemma 13 and the description of our construction algorithm.

5.2.1 Detecting Extended Lyndon Runs

In this section, we propose a linear time algorithm that identifies extended Lyndon runs, i.e. strings of the form $\mathrm{suf}(\mu) \cdot \mu^t \cdot \mathrm{pre}(\mu)$ with $t \geq 2$, where $\mathrm{suf}(\mu)$ and $\mathrm{pre}(\mu)$ are a proper suffix and a proper prefix of μ. Our approach exploits properties of the Lyndon factorization, which is defined as follows:

▶ **Lemma 14** (Lyndon Factorization [5]). *Every non-empty string $\mathcal{S}$ can be decomposed into non-empty Lyndon words $s_1, s_2, \ldots, s_m$ such that $\mathcal{S} = s_1 \cdot s_2 \cdot \ldots \cdot s_m$ and $\forall i \in [2, m] : s_{i-1} \succeq s_i$. There is exactly one such factorization for each string.*

▶ **Lemma 15.** *Let $\mathcal{S} = \mathrm{suf}(\mu) \cdot \mu^t \cdot \mathrm{pre}(\mu)$ be an extended Lyndon run. Let $x_1, \ldots, x_{k_1}$ be the Lyndon factorization of $\mathrm{suf}(\mu)$, and let $y_1, \ldots, y_{k_2}$ be the Lyndon factorization of $\mathrm{pre}(\mu)$. Then the Lyndon factorization of $\mathcal{S}$ is $x_1, \ldots, x_{k_1}, \underbrace{\mu, \mu, \ldots, \mu}_{t \text{ times}}, y_1, \ldots, y_{k_2}$.*

Proof. Clearly, the factorization given by the lemma consists solely of Lyndon words. Thus, we only have to show $x_1 \succeq \ldots \succeq x_{k_1} \succeq \mu \succeq y_1 \succeq \ldots \succeq y_{k_2}$. Since we defined $x_1, \ldots, x_{k_1}$ and $y_1, \ldots, y_{k_2}$ to be the Lyndon factorizations of $\mathrm{suf}(\mu)$ and $\mathrm{pre}(\mu)$ respectively, we already know that $\forall i \in [2, k_1] : x_{i-1} \succeq x_i$ and $\forall i \in [2, k_2] : y_{i-1} \succeq y_i$ hold. It remains to be shown that $x_{k_1} \succeq \mu \succeq y_1$ holds. Since x_{k_1} is a non-empty suffix of $\mathrm{suf}(\mu)$ and thus also a non-empty proper suffix of μ, it follows that $x_{k_1} \succ \mu$ holds. Since y_1 is a prefix of $\mathrm{pre}(\mu)$ and thus also a prefix of μ, it follows (by definition of the lexicographical order) that $\mu \succ y_1$ holds. ◀

Lemma 15 implies that the longest factor of the Lyndon factorization of an extended Lyndon run is exactly the repeating Lyndon word μ. This is the key insight that we use to detect extended Lyndon runs:

▶ **Lemma 16.** *Let $\mathcal{S}$ be a string of length n. If $\mathcal{S}$ is an extended Lyndon run of the form $\mathcal{S} = \mathrm{suf}(\mu) \cdot \mu^t \cdot \mathrm{pre}(\mu)$, then we can determine $|\mu|$ and $|\mathrm{suf}(\mu)|$ in $\mathcal{O}(n)$ time and $\mathcal{O}(1)$ words of additional space.*

Proof. Using Duval's algorithm [8, Algorithm 2.1], we compute the Lyndon factorization of $\mathcal{S}$ in $\mathcal{O}(n)$ time and $\mathcal{O}(1)$ words of additional space. The algorithm computes and outputs the factors one-at-a time and in left-to-right order. Whenever it outputs a factor that is longer than all previous ones, we store its length l and its starting position d. Note that since we investigate each factor individually and then immediately discard it, we never need to store the entire factorization in memory. If $\mathcal{S}$ is an extended Lyndon run, then following Lemma 15 it must have the form $\mathcal{S} = \mathrm{suf}(\mu) \cdot \mu^t \cdot \mathrm{pre}(\mu)$ with $|\mathrm{suf}(\mu)| = d - 1$ and $|\mu| = l$. Since we know d and l, checking whether $\mathcal{S} = \mathrm{suf}(\mu) \cdot \mu^t \cdot \mathrm{pre}(\mu)$ holds can be achieved by performing a simple scan over $\mathcal{S}$. ◀

6 Algorithmic Summary & Adaptation to the Lyndon Array

We now summarize our construction algorithm for the PSS tree. We process the indices from left to right using the techniques from Section 4.1, where processing an index means attaching it to the PSS tree. Whenever the critical time of processing an index is $\mathcal{O}(\ell)$, we skip the next $\Omega(\ell)$ indices by using the run extension (Section 5.1) or the amortized look-ahead (Section 5.2). Thus, the critical time per index is constant, and the total worst-case execution time is $\mathcal{O}(n)$. In terms of working space, we only need $\mathcal{O}(n \lg\lg n / \lg n)$ bits to support the operations described in Section 3.1. The correctness of the algorithm follows from the description. We have shown:

▶ **Theorem 17.** *For a string $\mathcal{S}$ of length n we can compute its succinct Lyndon array $\mathcal{B}_{\mathsf{pss}}$ in $\mathcal{O}(n)$ time using $\mathcal{O}(n \lg\lg n / \lg n)$ bits of working space apart from the space needed for $\mathcal{S}$ and $\mathcal{B}_{\mathsf{pss}}$.*

The algorithm can easily be adapted to compute the Lyndon array instead of the PSS tree. For this purpose, we use a single array $\mathcal{A}$ (which later becomes the Lyndon array), and no further auxiliary data structures. We maintain the following invariant: At the time we start processing index i, we have $\mathcal{A}[j] = \mathsf{pss}[j]$ for $j \in \mathcal{P}_{i-1}$, and $\mathcal{A}[j] = \lambda[j]$ for $j \in [1, i) \setminus \mathcal{P}_{i-1}$. As before, we determine $p_m = \mathsf{pss}[i]$ with the techniques from Section 4.1. In Step 1 and Step 2 we require some access on elements of $\mathcal{P}_{i-1}$, which we can directly retrieve from $\mathcal{A}$. Apart from that, the algorithm remains unchanged. Once we computed p_m, we set $\mathcal{A}[i] \leftarrow p_m$ $(= \mathsf{pss}[i])$. Additionally, it follows that i is the first node that is not a descendant of any of the nodes $p_1, \ldots, p_{m-1}$, which means that we have $\mathsf{nss}[p_x] = i$ for any such node. Therefore, we assign $\mathcal{A}[p_x] \leftarrow i - p_x$ $(= \lambda[p_x])$. The run extension and the amortized look-ahead remain essentially unchanged, with the only difference being that we copy and append respective array intervals instead of BPS substrings (some trivial shifts on copied values are necessary). Once we have processed index n, we have $\mathcal{A}[j] = \mathsf{pss}[j]$ for $j \in \mathcal{P}_n$, and $\mathcal{A}[j] = \lambda[j]$ for $j \in [1, n] \setminus \mathcal{P}_n$. Clearly, all indices $p_x \in \mathcal{P}_n$ do not have a next smaller suffix, and we set $\mathcal{A}[p_x] \leftarrow n - p_x + 1 = \lambda[p_x]$. After this, we have $\mathcal{A} = \lambda$. Since at all times we only use $\mathcal{A}$ and no auxiliary data structures, the additional working space needed (apart from input and output) is constant. The linear execution time and correctness of the algorithm follow from the description. Thus we have shown:

▶ **Theorem 18.** *Given a string $\mathcal{S}$ of length n, we can compute its Lyndon array λ in $\mathcal{O}(n)$ time using $\mathcal{O}(1)$ words of working space apart from the space needed for $\mathcal{S}$ and λ.*

7 Experimental Results

We implemented our construction algorithm for both the succinct and the plain Lyndon array (LA-Succ and LA-Plain). The C++ implementation is publicly available at GitHub[2]. As a baseline we compared the throughput of our algorithms with the throughput of DivSufSort[3], which is known to be the fastest suffix array construction algorithm in practice [12]. Thus, it can be seen as a natural lower bound for any Lyndon array construction algorithm that depends on the suffix array. Additionally we consider LA-ISA-NSV, which builds the Lyndon array by computing next smaller values on the inverse suffix array (see [13], we use DivSufSort to construct the suffix array). For LA-Succ we only construct the succinct Lyndon array without the support data structure for fast queries. All experiments were conducted on the LiDO3 cluster[4], using an Intel Xeon E5-2640v4 processor and 64GiB of memory. We repeated each experiment five times and use the median as the final result. All texts are taken from the Pizza & Chili text corpus[5].

Table 1 shows the throughput of the different algorithms, i.e. the number of input bytes that can be processed per second. We are able to construct the plain Lyndon array at a speed of between 41 MiB/s (`fib41`) and 82 MiB/s (`xml`), which is on average 9.9 times faster than LA-ISA-NSV, and 8.1 times faster than DivSufSort. Even in the worst case, LA-Plain is still 6.8 times faster than LA-ISA-NSV, and 5.2 times faster than DivSufSort (`pitches`). When constructing the succinct Lyndon array we achieve around 86% of the throughput of LA-Plain on average, but never less than 81% (`pitches`). In terms of memory usage, we measured the additional working space needed apart from the space for the text and the

[2] `https://github.com/jonas-ellert/nearest-smaller-suffixes`
[3] `https://github.com/y-256/libdivsufsort`
[4] `https://www.lido.tu-dortmund.de/cms/de/LiDO3/index.html`
[5] `http://pizzachili.dcc.uchile.cl/`

Table 1 Throughput in MiB/s.

	(normal corpus)						(repetitive corpus)			
	english.1GiB	dna	pitches	proteins	sources	xml	cere	einstein.de	fib41	kernel
LA-Plain	60.57	50.83	60.58	62.18	66.13	82.10	53.08	59.09	41.71	62.27
LA-Succ	52.81	46.03	49.49	52.77	57.31	68.56	48.20	50.35	35.30	54.42
LA-ISA-NSV	4.61	4.86	9.13	4.40	7.41	7.11	5.44	6.72	3.81	6.79
DivSufSort	5.53	5.76	11.61	5.21	9.25	8.62	6.57	8.45	4.20	8.45

(succinct) Lyndon array. Both LA-Plain and LA-Succ never needed more than 0.002 bytes of additional memory per input character (or 770 KiB of additional memory in total), which is why we do not list the results in detail.

8 Summary

We showed how to construct the succinct Lyndon array in linear time using $\mathcal{O}(n \lg\lg n/\lg n)$ bits of working space. The construction algorithm can also produce the (non-succinct) Lyndon array in linear time using only $\mathcal{O}(1)$ words of working space. There are no other linear time algorithms achieving these bounds. Our algorithm performs also extremely well in practice. We envision applications of these practical algorithms in full-text indexing, such as an improved implementation of Baier's suffix array construction algorithm [1], or as a first step in sparse suffix sorting [11, 4].

References

1 Uwe Baier. Linear-time suffix sorting - A new approach for suffix array construction. In *Proceedings of the 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016)*, Tel Aviv, Israel, June 2016. doi:10.4230/LIPIcs.CPM.2016.23.

2 Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The "Runs" theorem. *SIAM Journal on Computing*, 46(5):1501–1514, 2017. doi:10.1137/15M1011032.

3 Jérémy Barbay, Johannes Fischer, and Gonzalo Navarro. LRM-trees: Compressed indices, adaptive sorting, and compressed permutations. In *Proceedings of the 22nd Annual Symposium on Combinatorial Pattern Matching (CPM 2011)*, pages 285–298, Palermo, Italy, June 2011. doi:10.1007/978-3-642-21458-5_25.

4 Philip Bille, Johannes Fischer, Inge Li Gørtz, Tsvi Kopelowitz, Benjamin Sach, and Hjalte Wedel Vildhøj. Sparse text indexing in small space. *ACM Transactions on Algorithms*, 12(3):Article No. 39, 2016. doi:10.1145/2836166.

5 K. T. Chen, R. H. Fox, and R. C. Lyndon. Free differential calculus, IV. the quotient groups of the lower central series. *Annals of Mathematics*, 68(1):81–95, 1958. doi:10.2307/1970044.

6 Maxime Crochemore and Luís M. S. Russo. Cartesian and lyndon trees. *Theor. Comput. Sci.*, 806:1–9, 2020.

7 Jacqueline W. Daykin, Frantisek Franek, Jan Holub, A. S. M. Sohidull Islam, and W. F. Smyth. Reconstructing a string from its lyndon arrays. *Theor. Comput. Sci.*, 710:44–51, 2018.

8 Jean Pierre Duval. Factorizing words over an ordered alphabet. *Journal of Algorithms*, 4(4):363–381, 1983. doi:10.1016/0196-6774(83)90017-2.

9 Johannes Fischer. Optimal succinctness for range minimum queries. In *Proceedings of the 9th Latin American Symposium on Theoretical Informatics (LATIN 2010)*, pages 158–169, Oaxaca, Mexico, April 2010. doi:10.1007/978-3-642-12200-2_16.

10 Johannes Fischer. Combined data structure for previous- and next-smaller-values. *Theoretical Computer Science*, 412(22):2451–2456, 2011.

11 Johannes Fischer, Tomohiro I, and Dominik Köppl. Deterministic sparse suffix sorting on rewritable texts. In *Proceedings of the 12th Latin American Theoretical Informatics Symposium (LATIN 2016)*, pages 483–496, Ensenada, México, April 2016. doi:10.1007/978-3-662-49529-2_36.

12 Johannes Fischer and Florian Kurpicz. Dismantling DivSufSort. In *Proceedings of the 25th Prague Stringology Conference (PSC 2017)*, pages 62–76, Prague, Czech Republic, August 2017.

13 Frantisek Franek, A. S. M. Sohidull Islam, Mohammad Sohel Rahman, and William F. Smyth. Algorithms to compute the Lyndon array. In *Proceedings of the 20th Prague Stringology Conference (PSC 2016)*, pages 172–184, Prague, Czech Republic, August 2016.

14 Frantisek Franek, Asma Paracha, and William F. Smyth. The linear equivalence of the suffix array and the partially sorted Lyndon array. In *Proceedings of the 21st Prague Stringology Conference (PSC 2017)*, pages 77–84, Prague, Czech Republic, August 2017.

15 Paweł Gawrychowski and Tomasz Kociumaka. Sparse suffix tree construction in optimal time and space. In *Proceedings of the 28th Annual Symposium on Discrete Algorithms (SODA 2017)*, pages 425–439, Barcelona, Spain, January 2017. doi:10.1137/1.9781611974782.27.

16 Alexander Golynski. Optimal lower bounds for rank and select indexes. *Theoretical Computer Science*, 387(3):348–359, 2007. doi:10.1016/j.tcs.2007.07.041.

17 Torben Hagerup. Sorting and searching on the word ram. In *Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science (STACS 1998)*, pages 366–398, Paris, France, February 1998. doi:10.1007/BFb0028575.

18 Christophe Hohlweg and Christophe Reutenauer. Lyndon words, permutations and trees. *Theoretical Computer Science*, 307(1):173–178, 2003. doi:10.1016/S0304-3975(03)00099-9.

19 Felipe A. Louza, W.F. Smyth, Giovanni Manzini, and Guilherme P. Telles. Lyndon array construction during Burrows–Wheeler inversion. *Journal of Discrete Algorithms*, 50:2–9, May 2018. doi:10.1016/j.jda.2018.08.001.

20 Felipe Alves Louza, Sabrina Mantaci, Giovanni Manzini, Marinella Sciortino, and Guilherme P. Telles. Inducing the Lyndon array. In *Proceedings of the 26th International Symposium on String Processing and Information Retrieval (SPIRE 2019)*, pages 138–151, Segovia, Spain, October 2019. doi:10.1007/978-3-030-32686-9_10.

21 J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001. doi:10.1137/s0097539799364092.

22 Kunihiko Sadakane and Gonzalo Navarro. Fully-functional succinct trees. In *Proceedings of the 21st Annual Symposium on Discrete Algorithms (SODA 2010)*, pages 134–149, Austin, TX, USA, January 2010. doi:10.1137/1.9781611973075.13.

23 Kazuya Tsuruta, Dominik Köppl, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Grammar-compressed self-index with Lyndon words. *CoRR*, abs/2004.05309, 2020. arXiv:2004.05309.