technische universität
dortmund

Master's Thesis

Efficient Computation of Nearest Smaller Suffixes

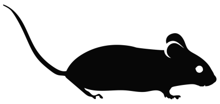
Jonas Ellert

July 8, 2019

Supervisors:
Prof. Dr. Johannes Fischer
M.Sc. Florian Kurpicz

Faculty of Computer Science
Algorithm Engineering (LS 11)
Technische Universität Dortmund
http://ls11-www.cs.tu-dortmund.de

Abstract. In this thesis we explore the possibilities of linear time Lyndon array construction. We propose an intuitive way of representing the Lyndon array as an ordered tree that can be stored in only $2n + o(n)$ bits of memory, which is asymptotically optimal. A novel algorithm allows the construction of this representation in $\mathcal{O}(\delta n)$ time, while using only $\lceil 4n/\delta \rceil + \mathcal{O}(\sqrt{n} \lg n)$ bits of additional memory (for a freely choosable parameter δ). It is the first algorithm that computes a representation of the Lyndon array in linear time without relying on other data structures like the suffix array. In practice, it is significantly faster than the algorithms that need the suffix array, while using only a fraction of the memory. Therefore, it makes the Lyndon array more accessible as a prerequisite for other algorithms and data structures.



My gratitude goes to my supervisors, my family, and my friends.

Contents

Chapter 1

Introduction

String processing is a fundamental discipline of computer science. Its applications are countless and influence our lives on a daily basis. For example, efficient string matching drives software like search engines, spell checkers, spam filters, and many more. The ability to index large collections of DNA sequences accelerates the research in genetics, improving our understanding of dangerous diseases and genetic disorders. Another important application of string processing is lossless text compression, which allows us to handle the steadily growing amounts of data that we encounter in the information age, for example during web mining, DNA sequencing, or when logging sensor data.

Many algorithms and data structures from the field of text indexing and compression utilize the same fundamental data structures as building blocks or prerequisites. Two common examples for this are the *suffix array* [Manber and Myers, 1990, 1993] and the *Burrows-Wheeler transform (BWT)* [Burrows and Wheeler, 1994]. In recent years, a new data structure called *Lyndon array* emerged, which could become another important building block. In this thesis we focus on the efficient computation of the Lyndon array, which makes it more practical as a prerequisite for other algorithms and data structures.

A *Lyndon word* (named after the American mathematician Roger Lyndon) is a string that is lexicographically smaller than all of its rotations. For example, `norway` is not a Lyndon word because it is lexicographically larger than its rotation `aynorw`. On the other hand, `belgium` is a Lyndon word because it is lexicographically smaller than all of its rotations. Given a string, the Lyndon array at position i contains the length of the longest Lyndon word that begins at position i of the string. While apparently it has been named Lyndon array for the first time in [Franek et al., 2016], it has previously been used without the explicit name in [Bannai et al., 2017] (preprint available since 2014). Since it was first introduced, the Lyndon array has become of great academic interest. In the aforementioned paper, Bannai et al. prove that there are less than n *runs* in any string of length n. A run is a maximal periodic subinterval of a string, like for example `ississi` in `mississippi`. Their algorithm to compute all runs utilizes the Lyndon array, showing its practical importance. Another use case of the Lyndon array emerged when Baier introduced a new linear-time suffix sorting algorithm [Baier, 2015, 2016]. In its first phase, this algorithm computes (a partially sorted version of) the Lyndon array, which is then uses to compute the suffix array in the second phase. Efficient Lyndon array construction algorithms could potentially speed up the first phase, and thus pave the way for faster suffix sorting algorithms.

In this thesis we present the *previous smaller suffix tree (PSS tree)*, a data structure of size $2n + \mathcal{O}(n/\lg^c n)$ bits that simulates access to the Lyndon array in $\mathcal{O}(c^2)$ time. Constructing the PSS tree requires two steps: First, we have to build the actual tree structure as a *balanced parentheses sequence (BPS)* of size $2n + 2$ bits. Then, we can add $\mathcal{O}(c^2)$ time query support by computing an auxiliary data structure of size $\mathcal{O}(n/\lg^c n)$ bits. We provide a parameterized construction algorithm that builds the PSS tree (or more precisely its BPS) in $\mathcal{O}(\delta n)$ time and uses only $\lceil 4n/\delta \rceil + \mathcal{O}(\sqrt{n} \cdot \lg n)$ bits of memory (apart from input and output). The algorithm is *elementary*, i.e. it uses no complex precomputed data structures like the suffix array or the Burrows-Wheeler transform. As it appears, this is the first elementary algorithm that computes a representation of the Lyndon array in linear time. The experimental evaluation confirms that the new algorithm is highly competitive in practice.

We also introduce two data structures that are strongly related to the Lyndon array: The *previous smaller suffix array (PSS array, denoted as* pss*)* and the *next smaller suffix array (NSS array, denoted as* nss*)*. Let S_i be the i-th suffix of a string S, then the next smaller suffix of S_i can be found by searching among the upcoming suffixes $S_{i+1}, \ldots, S_n$ for the first suffix S_j that is lexicographically smaller than S_i. The entry of the NSS array at position i is exactly the described index j, i.e. we have $\mathsf{nss}[i] = j$. The PSS array is defined analogously. An interesting property of next smaller suffixes is, that they inherently encode the Lyndon array. The longest Lyndon word starting at any text position i is exactly the string $S[i..\mathsf{nss}[i] - 1]$, which has been shown in [Franek et al., 2016, Lemma 15] and later also in [Louza et al., 2018, Lemma 1]. A similar property had previously been proven in [Hohlweg and Reutenauer, 2003, Corollary 3.1].

The PSS tree does not only simulate access to the Lyndon array, but also to the NSS array and the PSS array. In fact, it is merely a different representation of the PSS array. An additional useful feature of the PSS tree is its ability to answer *range minimum suffix queries (RMSQ)*: Given two indices i, j, we find the lexicographically smallest suffix S_k with $k \in [i, j]$ in $\mathcal{O}(c^2)$ time.

Finally, we prove that our data structure reaches optimal *succinctness*. The aim of succinct data structures is to store elements from a universe of size $L(n)$ in $\lg(L(n)) + o(\lg(L(n)))$

bits[1]. We will show that the information-theoretical lower bound for storing Lyndon arrays of length n is $\lg(L(n)) = 2n - \Theta(\lg n)$ bits, which means that our data structure of size $2n + o(n)$ bits is asymptotically optimal in terms of succinctness.

1.1 Overview

The paper is organized as follows: In the next section we discuss related work, focusing on existing approaches for Lyndon array construction. Chapter 2 establishes the basic definitions regarding string processing. Chapter 3 explains the PSS tree, its succinct representation, and how to simulate access to the various array. We present the construction algorithm for the PSS tree incrementally, starting in Chapter 4 with a solution that needs $\mathcal{O}(n^2)$ time and $\mathcal{O}(n)$ words of additional memory. In Chapter 5 we reduce the time bound to $\mathcal{O}(n)$. In Chapter 6 we propose novel space efficient stack representations. They allow us to introduce the parameter δ, reducing the memory bound of our algorithm to $\lceil 4n/\delta \rceil + \mathcal{O}(\sqrt{n} \cdot \lg n)$ bits for an execution time of $\mathcal{O}(\delta n)$. An extensive evaluation in Chapter 7 is followed by the conclusion and a discussion of future work in Chapter 8.

1.2 Related Work

1.2.1 Nearest Smaller Values

First, we discuss the *previous smaller value (PSV)* and *next smaller value (NSV)* problems. As the name suggests, they are in close relation to the previous and next smaller *suffix* array. For the PSS tree and its construction we utilize techniques that have previously been used in the context of PSVs and NSVs.

Given an array A, the PSV of $A[i]$ is the rightmost element $A[j]$ that is left of $A[i]$ and satisfies $A[j] < A[i]$. We write $\texttt{psv}[i] = j$. NSVs are defined analogously. The problem originates from parallel computing, where it has been shown that all PSVs and NSVs of an array of n entries can be computed in optimal $\mathcal{O}(\lg \lg n)$ time on a CRCW PRAM with $n/\lg \lg n$ processors [Berkman et al., 1993]. It is commonly known, that computing all PSVs and NSVs with a single processor takes $\mathcal{O}(n)$ time (for example [Goto and Bannai, 2013, Algorithm 4]). There is a wide variety of applications in which PSVs and NSVs can be used, for example in the context of compressed suffix trees [Fischer et al., 2008], as a prerequisite of Lempel-Ziv factorization [Goto and Bannai, 2013], and (for this thesis most importantly) as a simple way of computing the Lyndon array [Franek et al., 2016]. We will cover the Lyndon array construction with NSVs in Section 1.2.3. Most recently, PSVs

[1] In this thesis, lg denotes the binary logarithm.

have been used to define an alternative representation of *Cartesian trees* that enables new pattern matching techniques [Park et al., 2019].

There is an intuitive way of storing all PSVs of an array in an *ordinal tree*: Each index i is a child of $\texttt{psv}[i]$, and children of the same node are ordered ascendingly. This representation has been introduced under the name *LRM tree* as a navigational tool for succinct trees [Sadakane and Navarro, 2010], and under the name *2d-min-heap* in the context of range minimum queries [Fischer, 2010]. It is the inspiration for the PSS tree, which is a modified version of the LRM tree, where each index i of a string is a child of $\texttt{pss}[i]$.

1.2.2 Succinct Representations of Trees and the Lyndon Array

There exists a variety of succinct tree representations that store an unlabeled ordinal tree of n nodes in $2n$ bits. Most commonly known are the balanced parenthesis sequence (BPS), the depth-first unary degree sequence (DFUDS), and the level-order unary degree sequence (LOUDS) (see [Munro and Raman, 2001], [Benoit et al., 2005], and [Jacobson, 1989; Delpratt et al., 2006] respectively). Usually, these succinct representations are accompanied by an auxiliary support data structure of sublinear size that allows a wide variety of navigational operations on the tree. For the PSS tree, we use the BPS with Sadakane and Navarro's auxiliary data structure of size $\mathcal{O}(n/\lg^c n)$ bits, which supports all common operations (except for insertions and deletions) in $\mathcal{O}(c^2)$ time [Sadakane and Navarro, 2010].

Interestingly, while we use only $2n + o(n)$ bits of memory, we can simulate access to both the PSS *and* the NSS array. This is surprising because it has been shown that the number of bits needed for any combined NSV and PSV data structure is $\lg \mathcal{S}_n$, where $\mathcal{S}_n$ is the n-th Schröder–Hipparchus number with $\lg \mathcal{S}_n = 2n + \Omega(n)$ [Fischer, 2011]. We will see that the reason for this peculiarity is, that two entries at different positions of an array can be identical, while two suffixes starting at different positions of a string cannot.

Coincidentally, the BPS representation of the PSS tree is identical to the succinct Lyndon array representation shown in [Louza et al., 2018] (which also appears to be the only known succinct version of the Lyndon array). What differs is the *semantic interpretation*: Louza et al. do not interpret their representation as a tree, but identify each parenthesis pair as a maximal Lyndon word, where the distance between the opening parenthesis and the closing parenthesis determines the length of the word.

1.2.3 Lyndon Array Construction Algorithms

The literature differentiates between *elementary* and *non-elementary* Lyndon array construction algorithms. The former ones directly compute the Lyndon array from a given string without requiring precomputed data structures. The latter ones either require a precomputed data structure, or they produce the Lyndon array as a byproduct while computing another data structure. The exception is the first phase of Baier's suffix sorting algorithm [Baier, 2015, 2016], which cannot be clearly classified as elementary or non-elementary (we will cover it in a separate section).

In general, elementary algorithms offer less compelling worst-case guarantees in terms of execution time. For example, Duval's algorithm for Lyndon factorization can be applied to the input text in either iterative or recursive fashion, building the Lyndon array in $\mathcal{O}(n^2)$ time [Franek et al., 2016; Duval, 1983]. In the aforementioned paper by Franek et al., the authors provide an additional elementary algorithm that is based on the idea of next smaller suffixes. They believe that this algorithm builds the Lyndon array in $\mathcal{O}(\sigma n \lg n)$ time (where σ is the size of the alphabet), but do not provide a formal proof. It appears, that no linear time elementary algorithm has been discovered yet.

Non-elementary Lyndon array construction algorithms usually achieve linear time (even if we factor in the required precomputation time), but are slow in practice and have high memory requirements. This is due to the fact, that seemingly all of them either depend on the suffix array, or they simultaneously construct the Lyndon array *and* the suffix array. A surprisingly elegant and simple algorithm exploits the connection between next smaller suffixes and the Lyndon array by simply computing all NSVs of the inverse suffix array [Franek et al., 2016]. The same idea has been used in [Crochemore and Russo, 2018]. Another interesting approach is the Lyndon array construction during Burrows-Wheeler inversion. Given the Burrows-Wheeler transform of a string, it is possible to simultaneously restore the string and compute the Lyndon array in linear time [Louza et al., 2018]. This is very fast, if the BWT is already known. If only the plain text is given, then we first have to construct the BWT, which essentially requires the suffix array. Most recently, Nong's space efficient suffix array construction algorithm [Nong, 2013] has been modified to simultaneously compute the suffix array and the Lyndon array [Louza et al., 2019].

Baier's Suffix Sorting

Lastly, we want to briefly discuss Baier's linear time suffix sorting algorithm [Baier, 2015, 2016]. It has been thoroughly analyzed regarding its relation to the Lyndon array [Franek et al., 2017, 2018]. Essentially, the algorithm consists of two phases, which Franek et al. characterized as follows: In the first one, it computes a partially sorted version of the Lyndon array. The second phase exploits the partial sort to build the suffix array. This differs from other approaches because the suffix array is not used as prerequisite of computing the Lyndon array, and the Lyndon array is also not obtained as a byproduct of computing the suffix array. A more matching description would be, that the Lyndon array serves as a prerequisite of computing the suffix array. There exists a modification of the algorithm that outputs the Lyndon array (in its original form, i.e. not partially sorted) at the end of the first phase [Louza et al., 2018][2]. However, it still computes the partial sorting first, and then generates the non-sorted Lyndon array from it. Therefore, it cannot really be seen as a non-elementary algorithm.

[2] see also https://github.com/felipelouza/lyndon-array

Chapter 2

Preliminaries

In this chapter we introduce the notation and basic definitions that we use throughout this work. In terms of general notation, we simplify some recurring terms. Since we only use logarithms to base two, we simply write "$\lg x$" instead of "$\log_2 x$". The set $\mathbb{N}$ of natural numbers is defined as the non-negative integers $\{0, 1, 2, \dots\}$, whereas $\mathbb{N}^+$ denotes the positive integers $\{1, 2, 3 \dots\}$. Intervals are always to be considered discrete, i.e. for $x, y \in \mathbb{N}$ the interval $[x, y]$ represents the set $\{i \mid i \in \mathbb{N} \wedge x \leq i \leq y\}$. We use the notations $(x, y) = [x + 1, y - 1]$, $[x, y) = [x, y - 1]$, and $(x, y] = [x + 1, y]$ for open and half-open discrete intervals. Our research is situated in the word RAM model [Hagerup, 1998], 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 $\lceil \lg n \rceil \leq w$.

2.1 Strings & Lyndon Words

2.1.1 Definition (Alphabet). An *alphabet* Σ is a strictly totally ordered and finite set of *characters*. We assume that any two characters of the alphabet can be compared in constant time.

For figures and examples we use the lowercase English alphabet with the commonly used strict total order $\texttt{a} < \texttt{b} < \texttt{c} < \dots < \texttt{z}$.

2.1.2 Definition (String). A *string* $S \in \Sigma^*$ is a finite sequence of characters over an alphabet Σ. The length $|S|$ of the string is the number of characters it contains. We say that S is *empty* and write $S = \epsilon$, iff $|S| = 0$ holds. For $n = |S|$, $i \in [1, n]$ and $j \in [0, n]$ we define:

- $S[i]$ is the i-th character of S.
- $S[i..j]$ is the substring of length $j-i+1$ that starts at the i-th position and ends at the j-th position of S. For $j < i$ we define $S[i..j] = \epsilon$ instead.
- $S[i..j+1) = S(i-1..j] = S(i-1..j+1) = S[i..j]$.
- $S[1,j]$ is called *prefix* of S, and *proper prefix* of S, iff $j < n$.
- $S[j+1,n]$ is called *suffix* of S, and *proper suffix* of S, iff $j > 0$.
- $S_i = S[i..n]$ is the i-th suffix of S.

We use the notation $S \cdot T$ to express the concatenation of two strings S and T. The repeated concatenation of a string is denoted as $S^t = \underbrace{S \cdot S \cdot \ \ldots \ \cdot S}_{t \text{ times}}$ for $t \in \mathbb{N}^+$.

As a convention for this thesis, we use uppercase English letters $(S, T, U, \ldots)$ to denote strings. For better readability we sometimes use Greek lowercase letters $(\alpha, \beta, \gamma, \ldots)$ to denote substrings. For example, instead of repeatedly writing $S[i..j]$, we might simply define $\alpha = S[i..j]$ once, and then continue without having to use the cluttered index notation. The input string of a problem is also called *text*. Its indices can be referred to as *text positions* or *text indices*.

The strict total order on the alphabet naturally induces a strict total order on the set of strings over the alphabet:

2.1.3 Definition (Lexicographical Order). Let S and T be two strings over the same alphabet with $|S| = n$ and $|T| = m$. Without loss of generality we assume $n \leq m$. We say that the strings are equal and write $S = T$, iff $n = m$ and $\forall i \in [1,n] : S[i] = T[i]$ hold. We say that S is *lexicographically smaller* than T and write $S <_{\text{lex}} T$, iff the strings are not equal and one of the following conditions applies:

- S is a proper prefix of T, i.e. $S = T[1..n]$.
- S and T share a prefix of length $i-1$ for some $i \in [1,n]$, and the i-th character of S is smaller than the i-th character of T, i.e. $S[1,i) = T[1,i) \wedge S[i] < T[i]$.

Naturally, we also have the following operators:

- S is *lexicographically not larger* than T: $S \leq_{\text{lex}} T \Leftrightarrow S = T \vee S <_{\text{lex}} T$
- S is *lexicographically larger* than T: $S >_{\text{lex}} T \Leftrightarrow \neg(S \leq_{\text{lex}} T)$
- S is *lexicographically not smaller* than T: $S \geq_{\text{lex}} T \Leftrightarrow S = T \vee S >_{\text{lex}} T$

Below we see the lexicographical order of some strings over the English alphabet:

$$\texttt{five} <_{\text{lex}} \texttt{twenty} <_{\text{lex}} \texttt{twentyfive} <_{\text{lex}} \texttt{twohundred}$$

The first inequality holds because `five` and `twenty` do not share a non-empty common prefix and mismatch on the first character with $\texttt{f} < \texttt{t}$. The second inequality holds because `twenty` is a proper prefix of `twentyfive`. The third inequality holds because `twentyfive` and `twohundred` share a common prefix of length two and have a mismatch on the third character with $\texttt{e} < \texttt{o}$.

A fundamental subroutine in the field of string processing is the lexicographical comparison of suffixes. Comparing two suffixes essentially means, that we have to find the *longest common prefix* of the suffixes:

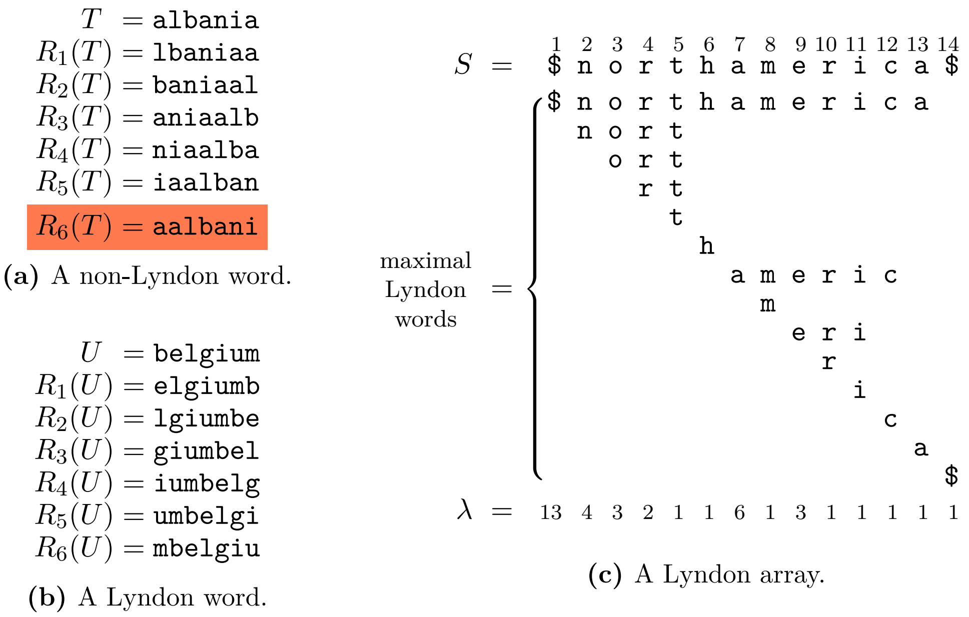
> **2.1.4 Definition (Longest Common Prefix).** Let S be a string of length n, and let $i, j \in [1, n]$ with $i < j$ be two indices. We define the *LCP value* of S_i and S_j as $\textsc{Lcp}_S(i, j) = \max\{l \mid j + l \leq n + 1 \land S[i..i + l) = S[j..j + l)\}$. The respective substring $S[i..i + \textsc{Lcp}_S(i, j))$ is called *longest common prefix (LCP)* of S_i and S_j.

When naively computing $\textsc{Lcp}_S(i, j)$, we have to compare S_i and S_j character by character until we find a mismatch or until we reach the end of S_j. In practice, checking if we reached the end of S_j for every single character comparison is expensive. We can avoid this corner case, as well as additional corner cases of our algorithms, if the input string fulfills the following criterion:

> **2.1.5 Definition.** Let S be a string of length $n > 1$. We say that S is *guarded by the sentinel* $\hat{s}$, iff both $S[1] = S[n] = \hat{s}$ and $\forall i \in (i, n) : \hat{s} < S[i]$ hold.

Being guarded also implies $S_n <_{\text{lex}} S_1 <_{\text{lex}} S_i$ for $i \in (1, n)$. Some strings are naturally guarded. For example, `austria` is guarded by the sentinel `a`. For strings that are not guarded we use the special character $\texttt{\$} \notin \Sigma$, which is smaller than all characters from Σ. We can make any string guarded by pre- and appending `$` at the front and the back of the string. For example, $S = \texttt{australia}$ is not guarded because of $S[1] \not< S[6]$, but `$australia$` is.

Finally we introduce Lyndon words, which are of particular interest for our research.



(a) A non-Lyndon word.

(b) A Lyndon word.

(c) A Lyndon array.

Figure 2.1: Strings, their rotations, and a Lyndon array.

2.1.6 Definition (Rotation and Lyndon Word). Let S be a non-empty string of length n. For $i \in [1, n)$ we call $R_i(S) = S_{i+1}S[1..i]$ the *i-th rotation* of S. We say that S is a *Lyndon word*, iff it is lexicographically smaller than all of its rotations, i.e. iff $\forall i \in [1, n) : S <_{\text{lex}} R_i(S)$ holds.

Figures 2.1a and 2.1b show two simple examples: The string $T =$ albania is not a Lyndon word, because it is lexicographically greater than its sixth rotation $R_6(T) =$ aalbani (Figure 2.1a). On the other hand, the string $U =$ belgium is a Lyndon word, because it is lexicographically smaller than all of its rotations (Figure 2.1b). Duval proposed a commonly used alternative characterization of Lyndon words:

2.1.7 Lemma (Duval, 1983 [Proposition 1.2]). *Let S be a non-empty string of length n. Then S is a Lyndon word, iff S is lexicographically smaller than all of its non-empty proper suffixes, i.e. iff $\forall i \in [2, n] : (S_i >_{\text{lex}} S)$ holds.*

Since each proper prefix of a string S is lexicographically smaller than the string itself, Lemma 2.1.7 implies that no non-empty proper suffix of S can also be a prefix of S, if S is a Lyndon word.

2.2 The Lyndon Array & Nearest Smaller Suffix Arrays

Now we introduce the data structures that are in the main focus of our research: The *Lyndon array*, the *next smaller suffix array*, and the *previous smaller suffix array*.

> **2.2.1 Definition (Lyndon Array).** Let S be a non-empty string of length n and let $i \in [1, n]$. Let $\hat{\imath} = \max\{j \mid j \in (i, n] \wedge (S[i..j) \text{ is a Lyndon word})\}$, or $\hat{\imath} = n+1$ for $i = n$. We call $S[i..\hat{\imath})$ the *maximal Lyndon word at position* i. The *Lyndon array* λ is an array of n entries with $\lambda[i] = (\hat{\imath} - i)$, i.e. $\lambda[i]$ contains the length of the maximal Lyndon word starting at position i.

In Figure 2.1c we see the string $S =$ `$northamerica$` and its Lyndon array λ. Each entry of the Lyndon array corresponds to one of the maximal Lyndon words that are displayed between the string and the Lyndon array. For instance, we have $\lambda[7] = 6$ with the corresponding maximal Lyndon word $S[7..12] =$ `americ` of length six.

The Lyndon array is essentially a different representation of the *next smaller suffix array*. Given a suffix S_i of the input string, the next smaller suffix of S_i is the closest suffix right of i that is lexicographically smaller than S_i.

> **2.2.2 Definition (Next Smaller Suffix Array).** Let S be a string of length n and let $i \in [1, n)$. We call S_j *next smaller suffix of* S_i, iff j is the smallest index from $(i, n]$ with $S_i >_{\text{lex}} S_j$. The *next smaller suffix array* (or short *NSS array*) of S is an array nss of size n with $\mathsf{nss}[i] = \min\{j \mid (j \in (i, n] \wedge S_i >_{\text{lex}} S_j) \vee (j = n+1)\}$ for $i \in [1, n]$.

Figure 2.2 (left) visualizes all next smaller suffixes in the text `$northamerica$`. For example, the next smaller suffix of $S_9 =$ `erica$` is $S_{12} =$ `ca$`, because `ca$` is lexicographically smaller than `erica$`, and both $S_{10} =$ `rica$` and $S_{11} =$ `ica$` are lexicographically larger than `erica$`.

2.2.3 Lemma (Franek et al., 2016 [Lemma 15], Louza et al., 2018 [Lemma 1]). *Let S be a string of length n, let λ be its Lyndon array, and let nss be its NSS array. For $i \in [1, n]$ we have $\lambda[i] = \mathsf{nss}[i] - i$.*

Remark. The original statement in [Louza et al., 2018, Lemma 1] is, that $\lambda[n] = 1$ and $\forall i \in [1, n) : \lambda[i] = k - i$ with $k = \min\{j \mid j \in (i, n] \wedge S_i >_{\text{lex}} S_j\}$ hold. Since this exactly matches the definition of the NSS array, we have $k = \mathsf{nss}[i]$ und thus $\lambda[i] = \mathsf{nss}[i] - i$. □

Clearly, given either the Lyndon array or the NSS array, we can simulate access to the other one in constant time. The natural counterpart of the NSS array is the *previous smaller suffix array*, which is defined analogously:

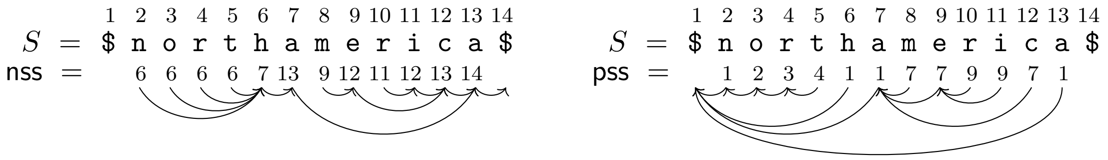

1 2 3 4 5 6 7 8 9 10 11 12 13 14
S = $ n o r t h a m e r i c a $
nss = 6 6 6 6 7 13 9 12 11 12 13 14
1 2 3 4 5 6 7 8 9 10 11 12 13 14
S = $ n o r t h a m e r i c a $
pss = 1 2 3 4 1 1 7 7 9 9 7 1

Figure 2.2: The PSS and NSS arrays of the string $S = \texttt{\$northamerica\$}$. For each index i there is an arch from i to $\mathsf{nss}[i]$ and $\mathsf{pss}[i]$ respectively. We omit outgoing arches and values for the sentinels.

> **2.2.4 Definition (Previous Smaller Suffix Array).** Let S be a string of length n and let $i \in [2, n]$. We call S_j *previous smaller suffix of* S_i, iff j is the largest index from $[1, i)$ with $S_j <_{\text{lex}} S_i$. The *previous smaller suffix array* (or short *PSS array*) of S is an array pss of size n with $\mathsf{pss}[i] = \max\{j \mid (j \in [1, i) \wedge S_j <_{\text{lex}} S_i) \vee (j = 0)\}$ for $i \in [1, n]$.

As seen in Figure 2.2 (right), we can interpret each entry of $\mathsf{pss}[i]$ as a pointer to the previous smaller suffix. For each index there is exactly one pointer, and each pointer's destination is a smaller index. Therefore, there is a unique path from every index i to the artificial index 0. In the figure there is the path $11 \to 9 \to 7 \to 1 \to 0$. The set of nodes on the path starting at index i form the so called *PSS closure* of i:

> **2.2.5 Definition (Previous Smaller Suffix Closure).** Let S be a string of length n, let pss be its PSS array, and let $i \in [0, n]$. The *PSS closure* $\mathsf{pss}^*[i]$ of i is defined as:
>
> $$\mathsf{pss}^*[i] = \begin{cases} \{0\} & \text{, iff } i = 0 \\ \{i\} \cup \mathsf{pss}^*[\mathsf{pss}[i]] & \text{, otherwise} \end{cases}$$

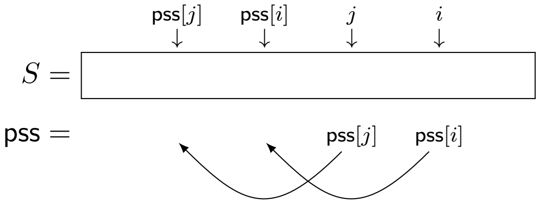
Revisiting the previous example, we have the closure $\mathsf{pss}^*[11] = \{0, 1, 7, 9, 11\}$, since there is the following chain of PSS pointers:

$$0 = \underbrace{\mathsf{pss}[1]}_{=\,0} = \mathsf{pss}[\underbrace{\mathsf{pss}[7]}_{=\,1}] = \mathsf{pss}[\mathsf{pss}[\underbrace{\mathsf{pss}[9]}_{=\,7}]] = \mathsf{pss}[\mathsf{pss}[\mathsf{pss}[\underbrace{\mathsf{pss}[11]}_{=\,9}]]]$$

The PSS closure $\mathsf{pss}^*[i-1]$ conveniently contains the value $\mathsf{pss}[i]$. For example, in Figure 2.2 we have $\mathsf{pss}[12] = 7 \in \mathsf{pss}^*[11]$. Therefore, whenever we want to find the previous smaller suffix of S_i, we only need to consider the indices from $\mathsf{pss}^*[i - 1]$. A simple approach for computing $\mathsf{pss}[i]$ is to try all indices from $\mathsf{pss}^*[i - 1]$ in descending order.

2.2.6 Lemma. *Let S be a string of length n, let* pss *be its PSS array, and let $i \in [2, n]$. Then it holds:* $\mathsf{pss}[i] = \max\{j \mid j \in \mathsf{pss}^*[i - 1] \wedge (S_j <_{\text{lex}} S_i \vee j = 0)\}$.

Proof. We only have to show that $\mathsf{pss}[i] \in \mathsf{pss}^*[i-1]$ holds. Then, the correctness follows directly from Definition 2.2.4. Assume that $\mathsf{pss}[i] \notin \mathsf{pss}^*[i-1]$ holds, and let j be the smallest index from $\mathsf{pss}^*[i-1]$ with $j > \mathsf{pss}[i]$. We have $j < i$ and $\mathsf{pss}[i] \in (\mathsf{pss}[j], j)$:



The fact that we have $\mathsf{pss}[i] \in (\mathsf{pss}[j], j)$ implies that $S_j <_{\text{lex}} S_{\mathsf{pss}[i]}$ holds (otherwise we would have $\mathsf{pss}[j] \geq \mathsf{pss}[i]$). On the other hand, $j \in (\mathsf{pss}[i], i)$ implies that $S_{\mathsf{pss}[i]} <_{\text{lex}} S_j$ holds (otherwise we would have $\mathsf{pss}[i] \geq j$). Because of this contradiction there cannot be $j < i$ with $\mathsf{pss}[i] \in (\mathsf{pss}[j], j)$, and thus we have $\mathsf{pss}[i] \in \mathsf{pss}^*[i-1]$. □

Finally, we show a direct relation between nearest smaller suffixes and Lyndon words.

2.2.7 Lemma. *Let S be a string of length n, let* nss *be its NSS array, and let* pss *be its PSS array. It holds:*

- *Let $i \in [1, n]$, and let $j = \mathsf{nss}[i]$. The string $S[i..j)$ is a Lyndon word.*
- *Let $j \in [2, n]$, and let $i = \mathsf{pss}[j]$. The string $S[i..j)$ is a Lyndon word.*

Proof. In case of $j = \mathsf{nss}[i]$, we know that $S[i..j)$ is exactly the longest Lyndon word starting at position i (Lemma 2.2.3). Therefore, we only have to consider the case $i = \mathsf{pss}[j]$. Following Lemma 2.1.7, the string $S[i..j)$ is a Lyndon word, iff there exists no $k \in (i, j)$ with $S[k..j) < S[i..j)$. Assume that such k exists. By definition of previous smaller suffixes, and because of $i = \mathsf{pss}[j]$, the following properties hold:

$$\text{(a)} \quad S_k >_{\text{lex}} S_i \qquad\qquad \text{(b)} \quad S_{i+(j-k)} >_{\text{lex}} S_j$$

If there is a mismatching character between $S[k..j)$ and $S[i..j)$, then appending S_j to both strings preserves this mismatch. This however implies, that we have $S[k..j) < S[i..j) \iff S[k..j) \cdot S_j < S[i..j) \cdot S_j$, and thus $S_k <_{\text{lex}} S_i$, which contradicts (a). Therefore, there is no mismatch between $S[k..j)$ and $S[i..j)$, and the following property holds:

$$\text{(c)} \quad S[k..j) = S[i..i + (j-k))$$

Finally, we combine the properties (a) and (c) to show a contradiction:

$$\begin{array}{rccc} & S_k & \overset{\text{(a)}}{>_{\text{lex}}} & S_i \\ \iff & S[k..j) \cdot S_j & >_{\text{lex}} & S[i..i+(j-k)) \cdot S_{i+(j-k)} \\ \underset{\text{(c)}}{\iff} & S_j & >_{\text{lex}} & S_{i+(j-k)} \end{array}$$

This contradicts property (b), and it follows that the described k does not exist. Therefore, $S[i..j)$ is a Lyndon word. □

2.3 Ordinal Trees

> **2.3.1 Definition (Ordinal Tree).** Let V be a set of *nodes*, let $\textsc{Parent} : V \to V$ be a function, and let $<$ be a strict total order on V (or on a superset of V). We call $\mathcal{T} = (V, \textsc{Parent}, <)$ *ordinal tree* (or simply *tree*) *of size* $|V|$, iff there is a unique *root* $r \in V$ that is an *ancestor* of all nodes. For $v \in V$ and $i \in \mathbb{N}^+$ the ancestor function is defined recursively as:
>
> $$\textsc{Ancestor}_i(v) = \begin{cases} \textsc{Parent}(v) & \text{, iff } i = 1 \\ \textsc{Parent}(\textsc{Ancestor}_{i-1}(v)) & \text{, otherwise} \end{cases}$$
>
> The root r must satisfy the condition $\forall v \in V : \lim_{x \to \infty} \textsc{Ancestor}_x(v) = r$.

Figure 2.3 shows an ordinal tree with nodes $V = [0, 13]$. Each parent relation is represented by a directed edge from the child to the parent. The root is its own parent, which is necessary in order to satisfy $\forall v \in V : \lim_{x \to \infty} \textsc{Ancestor}_x(v) = r$.

Basic Terminology

Let $u, v \in V$ with $u \neq v$ be nodes of a tree $\mathcal{T} = (V, \textsc{Parent}, <)$. We say that u is a *child* of v, iff $\textsc{Parent}(u) = v$ holds. In our example tree, node 3 is a child of node 1. If v lies on the path from u to the root, i.e. if there is some $i \in \mathbb{N}^+$ with $\textsc{Ancestor}_i(u) = v$, then we say that u is a *descendant* of v. In our example, the descendants of node 7 are exactly the nodes from the interval $[8, 13]$. A node without descendants, for instance node 5, is called *leaf.* From now on we assume that $u < v$ holds. The nodes u and v are called *siblings*, iff they share the same parent, i.e. if $\textsc{Parent}(u) = \textsc{Parent}(v)$ holds. More precisely, u is called *left-side sibling of* v, while v is called *right-side sibling of* u. If additionally there is no other sibling between them, i.e. if $\nexists w \in V \setminus \{u, v\} : \textsc{Parent}(w) = \textsc{Parent}(u) \land u < w < v$ holds, then u is called *left sibling of* v, while v is called *right sibling* of u. Thus, each node

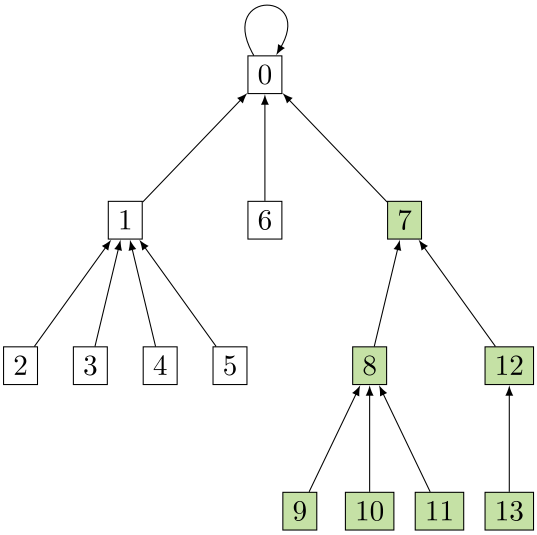


Figure 2.3: An order tree T with nodes $V = [0, 13]$ and root 0. The highlighted nodes belong to the subtree T_7, which is rooted in node 7.

can have multiple left-side siblings (and multiple right-side siblings), but at most one left sibling (and at most one right sibling). A node without left-side siblings is called *leftmost sibling* of its right-side siblings, and *leftmost child* of its parent (analogously defined for *rightmost sibling* and *rightmost child*). Node 3 of our example tree has the right-side siblings 4 and 5, where 4 is the right sibling, and 5 is the rightmost sibling. Node 5 is also the rightmost child of node 1.

2.3.1 Subtrees

Let $\mathcal{T} = (V, \textsc{Parent}, <)$ be a tree with ancestor function $\textsc{Ancestor}$. Each node $v \in V$ is the root of a subtree $\mathcal{T}_v = (V_v, \textsc{Parent}_v, <)$. The set V_v contains v and all of its descendants, i.e. $V_v = \{u \mid \exists i \in \mathbb{N}^+ : \textsc{Ancestor}_i(u) = v\} \cup \{v\}$. Since v is the new root, we have $\textsc{Parent}_v(v) = v$. For all other nodes $u \in V_v$ we have $\textsc{Parent}_v(u) = \textsc{Parent}(u)$.

> **2.3.2 Definition (Subtree Size).** Let $\mathcal{T} = (V, \textsc{Parent}, <)$ be a tree with ancestor function $\textsc{Ancestor}$, and let $v \in V$. We define the subtree size of v as:
>
> $$\textsc{SubtreeSize}(v) = |V_v| = \left|\{u \mid \exists i \in \mathbb{N}^+ : \textsc{Ancestor}_i(u) = v\} \cup \{v\}\right|$$

The colored nodes in Figure 2.3 belong to the subtree $\mathcal{T}_7$ of size $\textsc{SubtreeSize}(7) = 7$.

2.3.2 Tree Traversal

A tree traversal is the process of visiting each node in a tree exactly once. A *preorder-traversal* is a paradigm that clearly specifies the order in which we visit the nodes. We can assign a *preorder-number* to each node (denoted as $\#(v)$ in the following), which identifies the rank of the node in the visiting order, i.e. node v has preorder-number $\#(v) = i$, iff v is the i-th node that we visit. The first node has preorder-number 0, such that the set of preorder-numbers is exactly the interval $[0, |V|)$. Algorithmically, the numbers can be defined as follows: Assign the lowest unused number to the current node. Then, recursively assign numbers to the subtrees that are rooted in children of the current node, where we process the children in ascending (i.e. left-to-right) order. This procedure is outlined in Algorithm 2.1. We start with the root (line 8–9). Whenever we begin processing a node v, we first assign the next available preorder-number to it (line 3). Then, we recursively process the children of v in ascending order (line 5–6).

Algorithm 2.1 Preorder-Traversal

Input: Ordinal tree $\mathcal{T}$ with root r
Output: Preorder-numbers of all nodes of $\mathcal{T}$

```
1: $\textit{next-number} = 0$
2: function $\textit{assign-number}(v)$
3:     $\#(v) \leftarrow \textit{next-number}$
4:     $\textit{next-number} \leftarrow \textit{next-number} + 1$
5:     for $w \in \{u \mid \textsc{Parent}(u) = v\}$ in ascending order do
6:         $\textit{assign-number}(w)$

7: function $\textit{assign-numbers}(\mathcal{T})$
8:     $r \leftarrow$ root of $\mathcal{T}$
9:     $\textit{assign-number}(r)$
```

Another way of characterizing the preorder-traversal is, that every node gets visited after its parent, and also after its left sibling and the descendants of its left sibling (if the left sibling exists).

2.3.3 Observation. Let v be a node of an ordinal tree. Then it holds:

$$\#(v) = \begin{cases} \#(u) + \textsc{SubtreeSize}(u) & \text{, if } u \text{ is the left sibling of } v \\ \#(\textsc{Parent}(v)) + 1 & \text{, if } v \text{ has no left sibling} \end{cases}$$

Chapter 3

Previous Smaller Suffix Trees

In this chapter we introduce the *previous smaller suffix tree*, which is the data structure that we use to simulate the Lyndon array, the NSS array, and the PSS array.

3.1 Tree Structure

Since we have $\mathsf{pss}[i] < i$ for each $i \in [1, n]$, the PSS array inherently forms a tree in which each index i of the input string is represented by a node whose parent is $\mathsf{pss}[i]$. The root of this tree is the artificial index 0, which is the parent of all indices that do not have a previous smaller suffix. Once again, we use the string `$northamerica$` as an example (see Figure 3.1). As mentioned in Chapter 1, the structure of this tree is very similar to the LRM-Tree [Sadakane and Navarro, 2010], which is also known under the name 2d-min-heap [Fischer, 2010].

> **3.1.1 Definition (Previous Smaller Suffix Tree $\mathcal{T}_{\mathsf{pss}}$).** Let S be a string of length n. The *previous smaller suffix tree (PSS tree)* of S is a tree $\mathcal{T}_{\mathsf{pss}} = (V, \textsc{Parent}, <)$ with nodes $V = [0, n]$, where 0 is the root. For $i \in [1, n]$, we define $\textsc{Parent}(i) = \mathsf{pss}[i]$. The children are ordered ascendingly, i.e. if j is a right-side sibling of i, then $i < j$ holds.

Since node i corresponds to index i, from now on we use the terms node and index interchangeably. It is noteworthy, that the path from any node i to the root contains exactly the indices from the PSS closure $\mathsf{pss}^*[i]$. There are two essential properties of the PSS tree, the latter of which has previously been shown for 2d-min-heaps [Fischer, 2010]:

3.1.2 Lemma. *Let $\mathcal{T}_{\mathsf{pss}}$ be the PSS tree of a string S of length n.*

1. *The descendants of any node i form a continuous interval (i, r) with $r > i$.*
2. *The nodes directly correspond to the preorder-numbers of $\mathcal{T}_{\mathsf{pss}}$ such that node i has preorder-number i (if the first preorder-number is 0).*

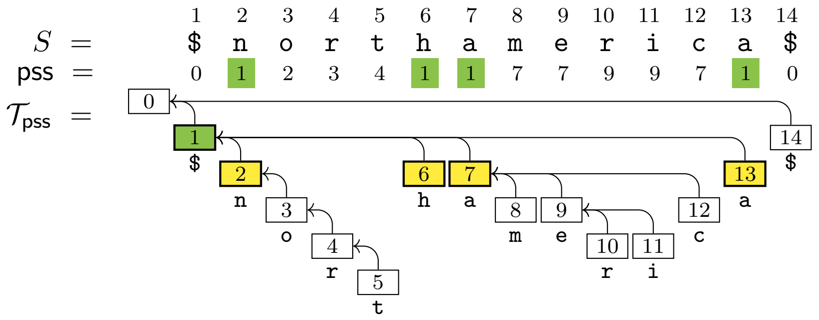

1 2 3 4 5 6 7 8 9 10 11 12 13 14
S = $ n o r t h a m e r i c a $
pss = 0 1 2 3 4 1 1 7 7 9 9 7 1 0
$\mathcal{T}_{\text{pss}}$ =
0
1 $
2 n
3 o
4 r
5 t
6 h
7 a
8 m
9 e
10 r
11 i
12 c
13 a
14 $

Figure 3.1: The PSS tree of the string $S =$ “`$northamerica$`”. The highlighted indices $2, 6, 7$, and 13 are exactly the text positions whose previous smaller suffix is S_1. Therefore they are the children of node 1. The children are sorted in increasing index order.

Proof. We address each property individually.

Regarding 1.: Assume, that r is the smallest node that is not a descendant of i. Since for all descendants $j \in (i, r)$ of i we have $S_i <_{\text{lex}} S_j$, we must have $S_r <_{\text{lex}} S_i$. Otherwise, r would be a descendant of i. Now consider any node $k \in (r, n]$. If $S_i >_{\text{lex}} S_k$ holds, then clearly k is not a descendant of i. If however $S_i <_{\text{lex}} S_k$ holds, then we also have $S_r <_{\text{lex}} S_k$, which implies $\mathsf{pss}[k] \geq r$. In this case, k is not a descendant of i either.

Regarding 2.: Proof by induction. Base case: Node 0 is the root and therefore has preorder-number 0. Inductive step: If the nodes from $[0, i)$ have the correct preorder-numbers, then the node i also has the correct number. Let $p = \textsc{Parent}(i)$. Clearly, we have $p \in [0, i)$. Assume that i is the leftmost child of p, then from the first property of this lemma follows, that $p = i - 1$ holds. As seen in Observation 2.3.3, we have $\#(i) = \#(p) + 1 = i$. This means, that we assign the correct preorder-number to node i. Now assume that i is not the leftmost child, and let l be the left sibling of i. Clearly, we have $l \in [0, i)$. Following the first property of this lemma, the descendants of l are exactly the nodes from the interval (l, i). As seen in Observation 2.3.3, we have $\#(i) = \#(l) + \textsc{SubtreeSize}(l) = l + (i - l) = i$. This means, that we assign the correct preorder-number to node i.

3.1.3 Corollary. *Let $\mathcal{T}_{\text{pss}}$ be the PSS tree of a string S of length n, and let nss be its NSS array. Let $i \in [1, n]$ be a node, then we have $\mathsf{nss}[i] = i + \textsc{SubtreeSize}(i)$.*

Proof. Let (i, r) be the interval containing exactly the descendants of i (which is a continuous interval because of the first property of Lemma 3.1.2). By definition of previous smaller suffixes, we have $S_i <_{\text{lex}} S_j$ for all descendants $j \in (i, r)$ of i. Also, we have $S_i >_{\text{lex}} S_r$ because otherwise r would be a descendant of i. Therefore, r is the smallest index that is larger than i and satisfies $S_i >_{\text{lex}} S_r$ (or such index does not exist and we have $r = n + 1$). It follows $\mathsf{nss}[i] = r$. Since the subtree rooted in i has exactly size $r - i$, we have $\mathsf{nss}[i] = i + \textsc{SubtreeSize}(i)$. □

While the previous and next smaller *suffix* problems intuitively seem to be harder than the previous and next smaller *value* problems, the corollary does not hold for NSVs and the 2d-min-heap (which is essentially a previous smaller *value* tree). In the proof of the corollary we can only assume $S_i >_{\text{lex}} S_r$ because two suffixes can never be equal. Therefore, we have $S_i \not<_{\text{lex}} S_r \implies S_i >_{\text{lex}} S_r$. If we look at values instead of suffixes, then $S[i] \not< S[r] \implies S[i] > S[r]$ does not necessarily hold because we might have $S[i] = S[r]$. It has been shown, that it is still possible to encode all NSVs in the 2d-min-heap, if a binary coloring information is added to some of its nodes [Fischer, 2011].

3.2 Building the NSS Array from the PSS Tree

Corollary 3.1.3 shows how powerful the PSS tree is. The time needed to answer PSS queries is only limited by the time needed to retrieve the parent of a node, while the time needed to answer NSS queries (and thus to access the Lyndon array) depends solely on the time in which we can retrieve subtree sizes.

3.2.1 Corollary. *Let $\mathcal{T}_{\mathsf{pss}}$ be the PSS tree of a string S. We have $\mathsf{pss}[i] = \textsc{Parent}(i)$ and $\mathsf{nss}[i] = i + \textsc{SubtreeSize}(i)$. Therefore, we can answer PSS queries in $t_{\textsc{Parent}}$ time and NSS queries in $t_{\textsc{SubtreeSize}}$ time, where $t_{\textsc{Parent}}$ and $t_{\textsc{SubtreeSize}}$ are the times needed to answer $\textsc{Parent}(\cdot)$ and $\textsc{SubtreeSize}(\cdot)$ respectively.*

Even if we compute $\mathcal{T}_{\mathsf{pss}}$ without support for fast $\textsc{SubtreeSize}(\cdot)$ operations, it allows us to efficiently build the entire NSS array.

3.2.2 Corollary. *Let $\mathcal{T}_{\mathsf{pss}}$ be the PSS tree of a string S. We can compute the NSS array in $\mathcal{O}(n \cdot t_{\textsc{Parent}})$ time using $\mathcal{O}(s_{\textsc{Parent}})$ bits of additional memory, where $t_{\textsc{Parent}}$ and $s_{\textsc{Parent}}$ are the time and space needed to answer $\textsc{Parent}(\cdot)$.*

Proof. We construct the NSS array in three scans: In the first one, we assign $\mathsf{nss}[i] \leftarrow 1$ for all indices. Next, we perform a right-to-left scan over the indices from the interval $[2, n]$, during which we assign $\mathsf{nss}[\textsc{Parent}(i)] \leftarrow \mathsf{nss}[\textsc{Parent}(i)] + \mathsf{nss}[i]$ for each index i. Note that we only use $\mathsf{nss}[i]$ on the right side of the assignment *after* we have processed all nodes $j > i$, and thus also all children of i. Therefore, after the scan we have $\mathsf{nss}[i] = \textsc{SubtreeSize}(i)$. In a final scan we assign $\mathsf{nss}[i] \leftarrow \mathsf{nss}[i] + i$ for all indices. We have to answer $\textsc{Parent}(\cdot)$ exactly $n - 1$ times, resulting in the time bound of $\mathcal{O}(n \cdot t_{\textsc{Parent}})$. Only one $\textsc{Parent}(\cdot)$ query is answered at a time, explaining the additional memory of $\mathcal{O}(s_{\textsc{Parent}})$ bits. □

3.3 Succinct Representation of the PSS Tree

In this section we show how to represent the PSS tree in a way that is both memory and query time aware. Any pointer based approach is problematic, since storing a pointer from each node to its parent already requires $\Theta(n \lg n)$ bits of memory. If we also want fast NSS queries, then a naive solution would require another $\Theta(n \lg n)$ bits to store all subtree sizes.

There already exist smarter solutions that are directly applicable to the PSS tree. One of the most common tree representations in the field of succinct data structures is the *Balanced Parentheses Sequence (BPS)* [Munro and Raman, 2001], which can store an unlabeled ordinal tree of n nodes in $2n$ bits. While there are other tree representations with similar benefits (for example the DFUDS [Benoit et al., 2005] and the LOUDS [Jacobson, 1989; Delpratt et al., 2006]), the BPS is particularly suitable for our purposes: The algorithms shown in Chapters 4 and 5 efficiently construct the BPS of the PSS tree from left to right in an append only manner. First, we explain the structure of the BPS. Then, we show how to use the BPS of the PSS tree to answer both PSS and NSS queries in constant time. Finally, we also explain how to answer *range minimum suffix queries*, and prove that in terms of space complexity the BPS of the PSS tree is an asymptotically optimal way of encoding the Lyndon array.

3.3.1 Storing the Tree as a Balanced Parentheses Sequence

3.3.1 Definition. Let $\mathcal{T}$ be an ordinal tree with nodes V and root $r \in V$. The *Balanced Parentheses Sequence (BPS) of* $\mathcal{T}$ is a string of $2n$ parentheses, where each node v is represented by a pair of matching parentheses "$(\dots)$", and all children of v are encoded between this pair of parentheses. For $v \in V$ we define:

$$enc(v) = \begin{cases} () & \text{, iff } v \text{ is a leaf} \\ (\cdot enc(c_1) \cdot \ \dots \ \cdot enc(c_k) \cdot) & \text{, iff } v \text{ has children } c_1 < c_2 < \dots < c_k \end{cases}$$

The BPS of $\mathcal{T}$ is defined as $\mathcal{B}(\mathcal{T}) = enc(r)$. We write $\mathcal{B}_{\mathsf{pss}}$ as a shorthand for $\mathcal{B}(\mathcal{T}_{\mathsf{pss}})$.

We can more intuitively describe the BPS as the result of a preorder-traversal of the tree. Starting with an empty parentheses string, we append an opening parenthesis whenever we *walk down* an edge, and a closing parenthesis whenever we *walk up* an edge. This traversal nature of the BPS also shows, that the node with preorder-number i corresponds to the $(i + 1)$-th opening parenthesis of the BPS (if we start counting preorder-numbers with 0 and parentheses with 1). Since in turn the preorder-numbers of the PSS tree correspond directly to the node labels (see Lemma 3.1.2), we know that node i corresponds to the

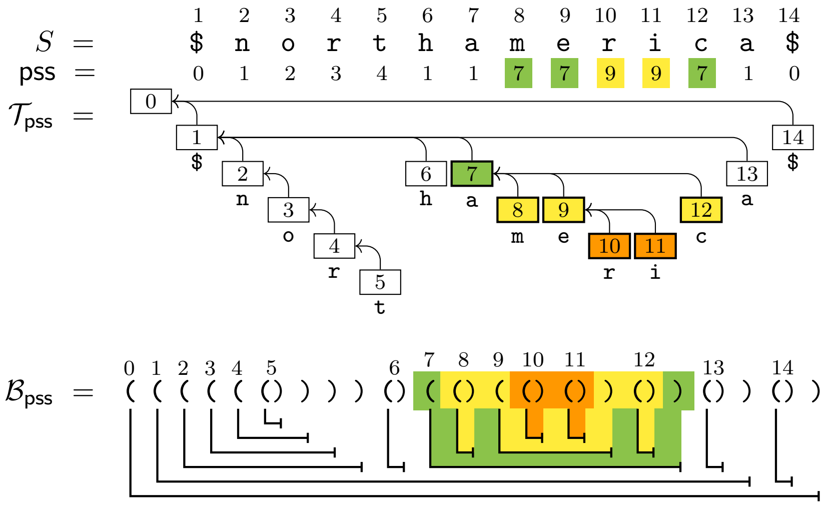


Figure 3.2: The string S = "`$northamerica$`", its PSS array pss, its PSS tree $\mathcal{T}_{\mathsf{pss}}$, and the BPS representation $\mathcal{B}_{\mathsf{pss}}$ of $\mathcal{T}_{\mathsf{pss}}$. Each opening parenthesis of $\mathcal{B}_{\mathsf{pss}}$ is annotated with the corresponding node label. For each opening parenthesis, the respective matching closing parenthesis can be found by following the line under the opening parenthesis. We have highlighted the subtree that is rooted in node 7.

$(i+1)$-th opening parenthesis. Figure 3.1 shows the BPS of the PSS tree of our running example `$northamerica$`.

Since $\mathcal{T}_{\mathsf{pss}}$ has $n+1$ nodes, there are $2n+2$ parentheses in $\mathcal{B}_{\mathsf{pss}}$. In practice, we use a bit vector to store the BPS, such that each opening parenthesis corresponds to a $(1)_2$-bit, and each closing parenthesis to a $(0)_2$-bit. Therefore, the entire tree needs only $2n+2$ bits. It remains to be shown how to efficiently simulate access to the various arrays in constant time.

3.3.2 Answering NSS and PSS Queries

As seen before, we only need fast Parent$(\cdot)$ and SubtreeSize$(\cdot)$ operations in order to efficiently answer PSS and NSS queries. These operations can be reduced to answering the following fundamental queries on the BPS:

- *select*(i): Returns the BPS index o_i of the opening parenthesis that corresponds to the node with preorder-number i, i.e. $\mathcal{B}_{\mathsf{pss}}[\mathit{select}(i)]$ is the $(i+1)$-th opening parenthesis.

- *find-close*(o_i): Returns the BPS index of the closing parenthesis that *matches* the opening parenthesis at position $\mathcal{B}_{\mathsf{pss}}[o_i]$.

- *enclose*(o_i): Returns the BPS index of the opening parenthesis that belongs to the tightest enclosing parentheses pair that contains $\mathcal{B}_{\mathsf{pss}}[o_i]$.

There are many ways of answering these basic queries in constant with only little additional memory. For example, Sadakane and Navarro's auxiliary data structure can be constructed in linear time, needs only $\mathcal{O}(n/\log^c n)$ bits of memory, and answers all of the above queries (and many more) in $\mathcal{O}(c^2)$ time (where we can choose any constant integer $c > 0$) [Sadakane and Navarro, 2010]. Now we show how to use the basic queries as building blocks for $\mathcal{O}(c^2)$ time $\textsc{Parent}(\cdot)$ and $\textsc{SubtreeSize}(\cdot)$ operations. Remember that node i represents textposition i, has preorder-number i and corresponds to the $(i+1)$-th opening parenthesis of the BPS.

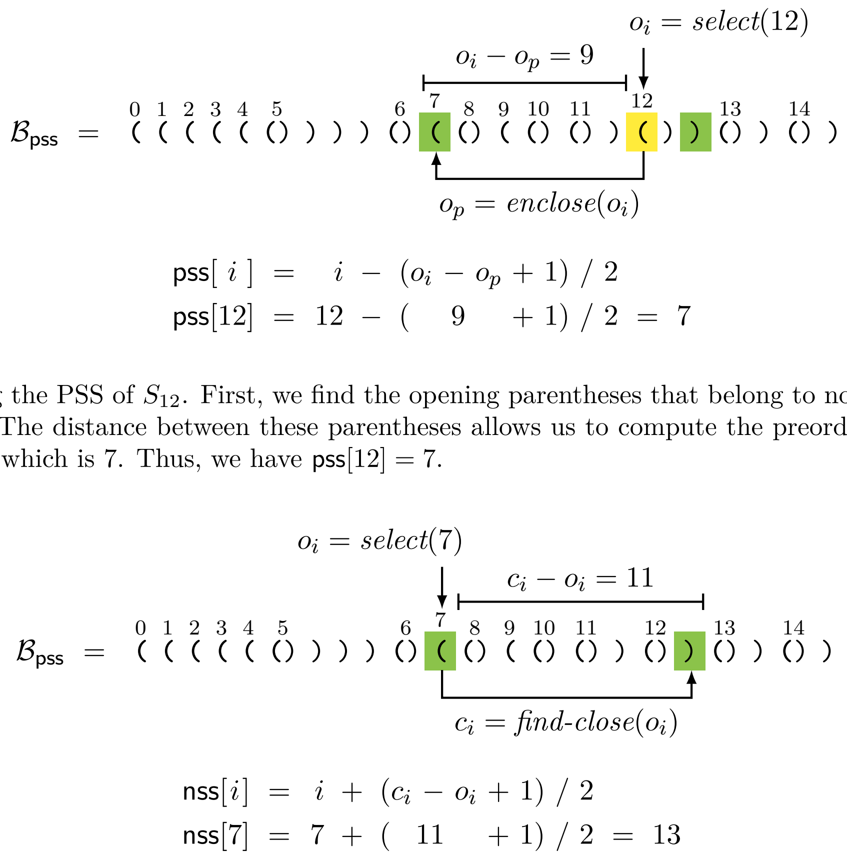
Simulating the PSS Array. For PSS queries we have $\mathsf{pss}[i] = \textsc{Parent}(i)$. First, we find the opening parenthesis of node i at index $o_i = \mathit{select}(i)$ of the BPS. Since node i is encoded between the parentheses pair of its parent, we can retrieve the opening parenthesis of node $\textsc{Parent}(i)$ as $o_p = \mathit{enclose}(o_i)$. The sequence $\mathcal{B}_{\mathsf{pss}}(o_p..o_i)$ contains exactly the encodings of the left-side siblings of i. Since each of these encodings is balanced, exactly half of the parentheses from $\mathcal{B}_{\mathsf{pss}}(o_p..o_i)$ are opening parentheses. Thus, the preorder-number of the parent node is $\mathsf{pss}[i] = \textsc{Parent}(i) = i - (o_i - o_p + 1)/2$. Figure 3.3a shows an example of the query execution.

Simulating the NSS array. For NSS queries we have $\mathsf{nss}[i] = i + \textsc{SubtreeSize}(i)$. Once again, we locate the opening parenthesis at position $o_i = \mathit{select}(i)$. The matching closing parenthesis is located at position $c_i = \mathit{find\text{-}close}(o_i)$. The subtree rooted in i is encoded in the sequence $\mathcal{B}_{\mathsf{pss}}[o_i..c_i]$. Since exactly half of the sequence contains opening parentheses, we have $\textsc{SubtreeSize}(i) = (c_i - o_i + 1)/2$, and thus $\mathsf{nss}[i] = i + (c_i - o_i + 1)/2$. An example is provided in Figure 3.3b.

Simulating the Lyndon array. Recalling Lemma 2.2.3, we have $\lambda[i] = \mathsf{nss}[i] - i$. As seen before, we have $\mathsf{nss}[i] = i + (c_i - o_i + 1)/2$, where c_i is the BPS index of the closing parenthesis of node i, and o_i is the BPS index of the opening parenthesis of node i. Thus, we have $\lambda[i] = (c_i - o_i + 1)/2$.

3.3.2 Lemma. *Let $c \in \mathbb{N}^+$, let S be a string of length n, and let $\mathcal{B}_{\mathsf{pss}}$ be the BPS of its PSS tree. We can augment $\mathcal{B}_{\mathsf{pss}}$ with an auxiliary data structure of size $\mathcal{O}(n/\lg^c n)$ bits that allows us to answer PSS and NSS queries in $\mathcal{O}(c^2)$ time. It can be constructed in $\mathcal{O}(n)$ time using $\mathcal{O}(n)$ bits of memory.*

Proof. Follows directly from the description above as well as [Sadakane and Navarro, 2010, Theorem 1.1]. □



$o_i = select(12)$
$o_i - o_p = 9$
$\mathcal{B}_{\mathsf{pss}}$ = (((((( )))) ( ) (( ) (( ) ( )) ( )) ( )) ( ))
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
$o_p = enclose(o_i)$
$\mathsf{pss}[\,i\,] = i - (o_i - o_p + 1)\ /\ 2$
$\mathsf{pss}[12] = 12 - (\ 9\ + 1)\ /\ 2 = 7$
$o_i = select(7)$
$c_i - o_i = 11$
$\mathcal{B}_{\mathsf{pss}}$ = (((((( )))) ( ) (( ) (( ) ( )) ( )) ( )) ( ))
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
$c_i = \mathit{find\text{-}close}(o_i)$
$\mathsf{nss}[i] = i + (c_i - o_i + 1)\ /\ 2$
$\mathsf{nss}[7] = 7 + (\ 11\ + 1)\ /\ 2 = 13$

(a) Finding the PSS of S_{12}. First, we find the opening parentheses that belong to node 12 and to its parent. The distance between these parentheses allows us to compute the preorder-number of the parent, which is 7. Thus, we have $\mathsf{pss}[12] = 7$.

(b) Finding the NSS of S_7. First, we find the opening and closing parentheses that belong to node 7. The distance between these parentheses allows us to compute the subtree size of node 7, which is 6. Thus, we have $\mathsf{nss}[7] = 7 + 6 = 13$.

Figure 3.3: Simulating access to the PSS array and the NSS array by using the BPS of the PSS tree. We use the same example as in Figure 3.2.

3.3.3 Answering Range Minimum Suffix Queries

A useful feature of the PSS tree is its ability to answer *range minimum suffix queries*:

> **3.3.3 Definition.** Let S be a string of length n, and let $i, j \in [1, n]$ with $i < j$. A *range minimum suffix query (RMSQ)* identifies the lexicographically smallest suffix that begins within the interval $[i, j]$:
>
> $$\mathrm{RMSQ}(i, j) = \min_{<_{\mathrm{lex}}}\{S_k \mid k \in [i, j]\}$$

For example, in Figure 3.2 we have $\mathrm{RMSQ}(4, 10) = 7$, because $S_7 =$ `america$` is the lexicographically smallest suffix that starts in the interval $[4, 10]$. Answering RMSQs can be realized by using the three operations that are explained below. All examples refer to Figure 3.2.

- $\textsc{Depth}(i)$: Returns the *depth* of node i in the PSS Tree, i.e. $\textsc{Depth}(0) = 0$, and $\textsc{Depth}(i) = \min\{j \mid \textsc{Ancestor}_j(i) = 0\}$ for $i \in [1, n]$. For example, we have $\textsc{Depth}(2) = 2$ and $\textsc{Depth}(11) = 4$.

- $\textsc{LevelAnc}(i, d)$: Returns the ancestor of node i that has depth d. Let $d_i = \textsc{Depth}(i)$, then we have $\textsc{LevelAnc}(i, d) = \textsc{Ancestor}_{d-d_i}(i)$. Revisiting our example tree, we have $\textsc{LevelAnc}(12, 1) = 1$ and $\textsc{LevelAnc}(10, 2) = 7$.

- $\text{LCA}(i, j)$: Returns the *lowest common ancestor* of nodes i and j. If i is an ancestor of j, then the result is i. If j is an ancestor of i, then the result is j. Otherwise, the result is $\textsc{LevelAnc}(i, \max\{d \mid \textsc{LevelAnc}(i, d) = \textsc{LevelAnc}(j, d)\})$. For example, we have $\text{LCA}(4, 8) = 1$ and $\text{LCA}(7, 11) = 7$.

3.3.4 Lemma. *Given the PSS tree of a string, we can answer* $\text{RMSQ}(i, j)$ *using only the queries described above. Let* $l = \text{LCA}(i, j)$. *If* $i = l$ *holds, then we have* $\text{RMSQ}(i, j) = i$. *Otherwise, we have* $\text{RMSQ}(i, j) = \textsc{LevelAnc}(j, \textsc{Depth}(l) + 1)$.

Proof. The lemma has been proven for range minimum (value) queries and the 2d-min-heap, and the proof also works for RMSQs and the PSS tree [Fischer, 2010, Lemma 2]. □

Earlier, we considered the example $\text{RMSQ}(4, 10) = 7$ (see Figure 3.2). Using the lemma as a query plan, we obtain the result as follows: We have $\text{LCA}(4, 10) = 1$ with $\textsc{Depth}(1) = 1$. Therefore, we have $\text{RMSQ}(4, 10) = \textsc{LevelAnc}(10, 1 + 1) = 7$. Conveniently, the operations $\textsc{Depth}(\cdot), \textsc{LevelAnc}(\cdot, \cdot)$, and $\text{LCA}(\cdot, \cdot)$ are supported by Sadakane and Navarro's auxiliary data structure.

3.3.5 Lemma. *Let* $c \in \mathbb{N}^+$, *let* S *be a string of length* n *and let* $\mathcal{B}_{\mathsf{pss}}$ *be the BPS of its PSS tree. We can augment* $\mathcal{B}_{\mathsf{pss}}$ *with an auxiliary data structure of size* $\mathcal{O}(n/\lg^c n)$ *bits that allows us to answer RMSQs in* $\mathcal{O}(c^2)$ *time. It can be constructed in* $\mathcal{O}(n)$ *time using* $\mathcal{O}(n)$ *bits of memory.*

Proof. Follows from Lemma 3.3.4 and [Sadakane and Navarro, 2010, Theorem 1.1]. □

3.3.4 Proving Optimal Succinctness

It is commonly known that there are C_n different ordinal trees that have $n+1$ nodes, where $C_n = \binom{2n}{n}/(n + 1)$ is the n-th Catalan number. Therefore, the information-theoretical lower bound for encoding an ordinal tree of $n + 1$ nodes is $\lg C_n$ bits. By using Stirling's approximation we obtain $\lg C_n = 2n - \mathcal{O}(\lg n)$. Thus, using $2n + o(n)$ bits for the BPS and its support data structure results in a tree representation that is asymptotically optimal in terms of space requirements. In simple terms, the BPS including the support data structure requires *a bit more* than $2n$ bits of memory, while the best possible encoding requires *a bit less* than $2n$ bits of memory. For large n, there is no significant difference between the two options ($\lim_{n\to\infty} \frac{2n+o(n)}{2n-\mathcal{O}(\lg n)} = 1$).

In this section, we show that $2n+o(n)$ bits are not only asymptotically optimal for encoding ordinal trees, but also for encoding the PSS tree and the Lyndon array. First, we show that any ordinal tree of $n+1$ nodes is the PSS tree of some string of length n. The proof works by construction: Given an ordinal tree of $n+1$ nodes, we explain how to build a string S of length n such that the ordinal tree is the PSS tree of the string. As always, we use preorder-numbers to identify nodes. For convenience, we define $S[0] = 0$. The resulting string is over the integer alphabet $\Sigma = [1, n]$.

We process the nodes recursively, starting at the root. Let i be the current node, and let $c_1, \dots, c_k$ be exactly the children of i in ascending order. For $j \in [1, k]$ we assign $S[c_j] \leftarrow S[i] + (k - j + 1)$. Then, we process each child recursively.

This construction ensures $S[\textsc{Parent}(i)] < S[i]$ for all nodes $i \in [1, n]$. Therefore, we also have $S_{\textsc{Parent}(i)} <_{\text{lex}} S_i$, and thus $\mathsf{pss}[i] \geq \textsc{Parent}(i)$. The nodes with preorder-numbers from the interval $(\textsc{Parent}(i), i)$ are exactly the left-side siblings of i, as well as their descendants. Let $j \in (\textsc{Parent}(i), i)$ be such a node, then due to the way our construction works, we inherently have $S[j] > S[i]$. Since this implies $S_j >_{\text{lex}} S_i$, it also follows that $\mathsf{pss}[i] \leq \textsc{Parent}(i)$ holds. Therefore, we have $\mathsf{pss}[i] = \textsc{Parent}(i)$, which means that the given tree is the PSS tree of the constructed string. Thus, every ordinal tree of size $n+1$ is the PSS tree of at least one string of length n. It follows:

3.3.6 Lemma. *The information-theoretical lower bound for the number of bits needed to encode a PSS tree of $n+1$ nodes is $\lg C_n = 2n - \mathcal{O}(\lg n)$.*

Finally, we show that there are at least as many Lyndon arrays as there are PSS trees. Consider two different PSS trees of size $n+1$, then there is at least one node that has a different subtree size in these trees (otherwise they would be identical). Since we have $\lambda[i] = \textsc{SubtreeSize}(i)$ (see Lemma 2.2.7 and Corollary 3.1.3), it follows that any two different PSS trees of size $n+1$ simulate access to different Lyndon arrays of size n. Thus, the number of different Lyndon arrays of size n is at least as high as the number of different PSS trees of size $n+1$.

3.3.7 Lemma. *The information-theoretical lower bound for the number of bits needed to encode a Lyndon array of n entries is at least $2n - \mathcal{O}(\lg n)$.*

3.3.8 Corollary. *The BPS of the PSS tree combined with the support data structure from Lemma 3.3.2 is an asymptotically optimal way of encoding the Lyndon array.*

Chapter 4

Constructing the PSS Tree

In the next chapters we introduce a space efficient and linear time algorithm that computes the BPS of the PSS tree without requiring precomputed data structures. We present the final solution in an incremental manner, starting with a very basic $\mathcal{O}(n^2)$ time and $\mathcal{O}(n)$ words memory algorithm. In Chapter 5 we reduce the time bound to $\mathcal{O}(n)$. In Chapter 6 we also reduce the space bound, resulting in a parameterized algorithm with compelling worst-case guarantees:

4.0.1 Theorem. *Let S be a string of length n and let $\delta \in [1, \lfloor\sqrt{n}/(3\lg n)\rfloor]$. The BPS of the PSS tree of S can be computed in $\mathcal{O}(\delta n)$ time using $\lceil 4n/\delta \rceil + \mathcal{O}(\sqrt{n} \cdot \lg n)$ bits of additional memory apart from the space needed for input and output.*

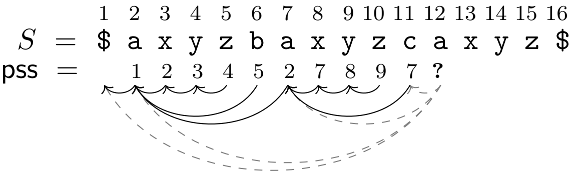
4.1 Computing the NSS & PSS Array

The foundation of our solution is a simple algorithm that calculates NSVs and PSVs. It has been implemented in the first version of the *Succinct Data Structure Library (SDSL)*[1], and pseudocode can be found for example in [Goto and Bannai, 2013, Algorithm 4]. If we take this algorithm and replace all element comparisons with suffix comparisons (i.e. replace $S[i] > S[j]$ with $S_i >_{\text{lex}} S_j$), then we already have an algorithm that computes the NSS and PSS array.

4.1.1 Pointer Jumping Technique

The general idea is simple: We calculate all entries of the PSS array from left to right. Lets assume we have already computed pss up to index $i-1$ and now we want to find $\mathsf{pss}[i]$. The most naive approach is to simply iterate over the indices $j \in [1, i)$ in decreasing order

[1] https://github.com/simongog/sdsl, see file `include/sdsl/algorithms.hpp`


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
S = $ a x y z b a x y z c a x y z $
pss = 1 2 3 4 5 2 7 8 9 7 ?

$$\mathsf{pss}^*[11] : \begin{cases} 11 & S_{11} \not<_{\text{lex}} S_{12} \\ 7 & S_7 \not<_{\text{lex}} S_{12} \\ 2 & S_2 \not<_{\text{lex}} S_{12} \\ 1 & S_1 <_{\text{lex}} S_{12} \end{cases}$$

Figure 4.1: Pointer Jumping during the calculation of the PSS array. We have already computed $\mathsf{pss}[1, 11]$. As displayed on the left side, every entry $\mathsf{pss}[i] = j$ can be interpreted as a pointer from index i to index j. When computing $\mathsf{pss}[12]$, we simply try the values from $\mathsf{pss}^*[11]$ in decreasing order, until we find a suffix that is lexicographically smaller than S_{12}, i.e. we start at index 11 and then follow the outgoing chain of pointers until we find a suitable value. For example, after we discover that $S_{11} \not<_{\text{lex}} S_{12}$ holds, we proceed with index $\mathsf{pss}[11] = 7$, i.e. we use the pointer to *jump* over the indices $10, 9$, and 8.

and set $\mathsf{pss}[i] \leftarrow j$ as soon as $S_j <_{\text{lex}} S_i$ holds. However, as we have seen in Lemma 2.2.6, the value $\mathsf{pss}[i]$ is contained in the PSS closure $\mathsf{pss}^*[i-1]$. Therefore, it is sufficient to iterate over the indices $j \in \mathsf{pss}^*[i-1]$ in decreasing order instead. This method saves a substantial amount of suffix comparisons and is often named *pointer jumping*, because trying the elements from $\mathsf{pss}^*[i-1]$ in decreasing order can be seen as following the chain of pss pointers that starts at index $i-1$. An example can be seen in Figure 4.1.

Even though we are saving suffix comparisons, following the chain of pss pointers is expensive in practice: We have to repeatedly look up many different entries of the PSS array that are potentially far apart from each other. Thus, there is a lot of random access on the array, which in return causes expensive cache misses. Therefore, we use a dedicated stack to store only the actually needed entries of the PSS array. At the time we want to compute $\mathsf{pss}[i]$, the stack H contains exactly the elements $1 = h_1, \dots, h_k = i - 1$ with $h_{j-1} = \mathsf{pss}[h_j]$ for $j \in [2, k]$, i.e. $\mathsf{pss}^*[i-1] = \{0, h_1, \dots, h_k\}$. In the following section, we show that the stack is easy to maintain while using the pointer jumping technique.

4.1.2 Computing the NSS & PSS Array Simultaneously

In Algorithm 4.1 (left) we see the pseudocode of $\mathsf{xss\text{-}array}$, which realizes the pointer jumping while utilizing the previously described stack H. So far we have only been talking about computing the PSS array. As a matter of fact, the algorithm computes both the PSS array *and* the NSS array at the same time. Before going into detail, it ought to be mentioned that this algorithm is designed to work on guarded strings only. In the beginning of the execution we push index 1 onto H (line 2). Since $S[1]$ is the left sentinel, we have $S_1 <_{\text{lex}} S_i$ for $i \in (1, n)$. Therefore, the index 1 always stays on the stack, and we can assume that the stack never becomes empty. Because of the sentinels we already know that $\mathsf{pss}[1] = \mathsf{pss}[n] = 0$ and $\mathsf{nss}[n] = n + 1$ hold (lines 3–4). Although we also know

Algorithm 4.1 xss-array (left) and xss-bps (right)

```
Input: Guarded string $S$ of length $n$
Output: PSS and NSS array of $S$
 1: $H \leftarrow$ empty stack
 2: $H.push(1)$
 3: $\mathsf{nss}[n] \leftarrow n+1$
 4: $\mathsf{pss}[1] \leftarrow 0$; $\mathsf{pss}[n] \leftarrow 0$;
 5: for $i = 2$ to $n-1$ do
 6:     while $S_{H.top()} >_{\text{lex}} S_i$ do
 7:         $\mathsf{nss}[H.top()] \leftarrow i$
 8:         $H.pop()$
 9:     $\mathsf{pss}[i] \leftarrow H.top()$
10:     $H.push(i)$
11: while $H$ is not empty do
12:     $\mathsf{nss}[H.top()] \leftarrow n$
13:     $H.pop()$
14:
15: return $(\mathsf{pss}, \mathsf{nss})$
```

```
Input: Guarded string $S$ of length $n$
Output: PSS tree of $S$ as BPS
 1: $H \leftarrow$ empty stack
 2: $H.push(1)$
 3: $\mathcal{B}_{\mathsf{pss}} \leftarrow$ "(("                          ▷ open 0 and 1
 4:
 5: for $i = 2$ to $n-1$ do
 6:     while $S_{H.top()} >_{\text{lex}} S_i$ do
 7:         $\mathcal{B}_{\mathsf{pss}} \leftarrow \mathcal{B}_{\mathsf{pss}} \cdot$ ")"       ▷ close $H.top()$
 8:         $H.pop()$
 9:     $\mathcal{B}_{\mathsf{pss}} \leftarrow \mathcal{B}_{\mathsf{pss}} \cdot$ "("               ▷ open $i$
10:     $H.push(i)$
11: while $H$ is not empty do
12:     $\mathcal{B}_{\mathsf{pss}} \leftarrow \mathcal{B}_{\mathsf{pss}} \cdot$ ")"           ▷ close $H.top()$
13:     $H.pop()$
14: $\mathcal{B}_{\mathsf{pss}} \leftarrow \mathcal{B}_{\mathsf{pss}} \cdot$ "())"              ▷ leaf $n$; close 0
15: return $\mathcal{B}_{\mathsf{pss}}$
```

that $\mathsf{nss}[0] = n$ holds, we do not manually assign this value at the beginning, since it will be assigned naturally at a later point of the algorithm execution.

Now we look at the main loop (line 5), which considers each index $i \in (1, n)$ between the sentinels in ascending order. As an invariant for this loop, we require the stack to work as described earlier: At the beginning of iteration i, the stack contains exactly the elements $1 = h_1, \ldots, h_k = i - 1$ with $h_{j-1} = \mathsf{pss}[h_j]$ for $j \in [2, k]$. Before the first iteration with $i = 2$ this invariant is clearly satisfied, since H contains only the index 1. Now we compare $S_{H.top()}$ with S_i and keep popping the topmost element on the stack until we have $S_{H.top()} <_{\text{lex}} S_i$ (lines 6–8). Each pop represents a pointer jump as described earlier, i.e. each pop means that we succeed to the next smaller index from $\mathsf{pss}^*[i-1]$. The invariant ensures two things: First, whenever we pop $H.top()$, there has not yet been an index $j \in (H.top(), i)$ with $S_{H.top()} >_{\text{lex}} S_j$. Therefore, we know that $\mathsf{nss}[H.top()] = i$ holds (line 7). Second, after reaching $S_{H.top()} <_{\text{lex}} S_i$, we also know that $\mathsf{pss}[i] = H.top()$ holds (line 9). By pushing i onto the stack, we maintain the invariant for the next iteration (line 10).

After processing index $n - 1$, we have assigned $\mathsf{pss}[i]$ for $i \in [1, n]$. Also, we have assigned $\mathsf{nss}[i]$ for all $i \in [1, n]$ that are currently not on the stack. Let $1 = h_1, \ldots, h_k = n - 1$ be the remaining indices on the stack. For any index h_j with $j \in [1, k]$ we clearly have not found an index $y > h_j$ with $S_{h_j} >_{\text{lex}} S_y$. Therefore, we can simply assign $\mathsf{nss}[h_j] = n$ for all stack elements (lines 11–13). Note that this also includes index 1, explaining why we did not need to assign $\mathsf{nss}[1]$ manually in the beginning.

4.1.1 Lemma. *The algorithm* xss-array *computes the NSS and PSS array of a guarded string of length n in $\mathcal{O}(n^2)$ time using $\mathcal{O}(n)$ words of memory apart from input and output.*

Proof. The correctness of the algorithm follows directly from the description and maintaining the invariant of the loop. Since the stack contains at most $n-1$ elements, we need $\mathcal{O}(n)$ words of additional memory. Each suffix comparison in line 6 is followed by either a push or a pop. Considering that each index gets pushed and popped at most once, we can follow that there are at most $2n$ suffix comparisons. Each comparison takes $\mathcal{O}(n)$ time, which is why all comparisons combined take at most $\mathcal{O}(n^2)$ time. This dominates the execution time of the algorithm. □

An interesting property of the algorithm is, that each suffix comparison directly corresponds to a previous or next smaller suffix relation. After comparing two suffixes S_j and S_i (line 6), we always assign either $\mathsf{nss}[j] = i$ (line 7) or $\mathsf{pss}[i] = j$ (line 9).

4.1.2 Lemma. *If we compare two suffixes S_j and S_i with $j < i$ during the execution of* xss-array*, then we have $\mathsf{nss}[j] = i$ or $\mathsf{pss}[i] = j$. This also holds for the algorithms* xss-bps*,* xss-bps-lcp*, and* xss-real*, which we present in the following sections.*

4.1.3 Corollary. *If we compare two suffixes S_j and S_i with $j < i$ during the execution of* xss-array*, then $S[j..i)$ is a Lyndon word. This also holds for the algorithms* xss-bps*,* xss-bps-lcp*, and* xss-real*, which we present in the following sections.*

Proof. Follows directly from Lemma 4.1.2 and Lemma 2.2.7. □

4.2 Computing the BPS of the PSS Tree

As we can see in Algorithm 4.1 (right), not many modifications to xss-array are needed to construct the BPS $\mathcal{B}_{\mathsf{pss}}$ of the PSS tree instead of the PSS and NSS array. The structure of the new algorithm xss-bps remains essentially unchanged.

We build $\mathcal{B}_{\mathsf{pss}}$ from left to right. In Section 3.3.1 we have seen that one way of obtaining the BPS of a tree is to perform a depth-first traversal and write an opening parenthesis, whenever we *walk down* an edge, and a closing parenthesis, whenever we *walk up* an edge. Intuitively, the algorithm can be seen as a simulation of a depth-first traversal of $\mathcal{T}_{\mathsf{pss}}$. At the beginning of iteration i of the outer loop, we are currently at node $H.top() = (i-1)$. Let $1 = h_1, \ldots, h_k = (i-1)$ be the elements of H, then clearly the stack contains exactly the path in $\mathcal{T}_{\mathsf{pss}}$ that starts at node $(i-1)$ and extends all the way up to the left sentinel node 1 (in Section 4.1.2 we have shown that $\forall j \in [2, k] : h_{j-1} = \mathsf{pss}[h_j]$ holds at the beginning of the iteration). The next node that we have to visit during the traversal is i. We know that i is a child of h_j for some $j \in [1, k]$. For all nodes on the path from h_k up to h_j, i.e. the nodes h_l with $l \in (j, k]$, we have $S_{h_l} >_{\text{lex}} S_i$. Therefore, we have one iteration of the inner loop for each such h_l, and thus write a total of $|(j, k]|$ closing parentheses (line 7). This exactly corresponds to walking up the path from h_k to h_j during the depth-first traversal. After the last iteration of the inner loop, the topmost element on H is h_j. Writing the opening parenthesis in line 7 directly corresponds to walking down the edge from h_j to i. As before, we push i onto H in order to satisfy the invariant of the outer loop (line 10).

After the last iteration of the outer loop, we are at node $H.top() = (n - 1)$. We write one closing parenthesis for each node on the stack, which corresponds to walking up the path from $(n - 1)$ to the root 0. Lastly, we append one parenthesis pair for the sentinel index n (which is a leaf), as well as the closing parenthesis of the root (line 14).

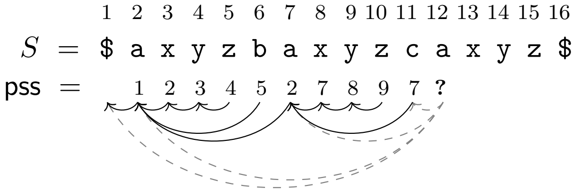
4.2.1 Corollary. *The algorithm* xss-bps *computes the BPS of the PSS tree of a guarded string of length n in $\mathcal{O}(n^2)$ time using $\mathcal{O}(n)$ words of memory apart from input and output.*

Proof. The correctness follows from the description. The time and memory bounds are identical with the ones of xss-array. □

4.3 Introducing a Separate LCP Stack

One of the most expensive operations in xss-bps is the suffix comparison in line 6. Comparing two suffixes S_i and S_j can be reduced to calculating the length of the prefix that is shared by both suffixes: Let $\textsc{Lcp}_S(i, j)$ be the length of the longest common prefix (see Definition 2.1.4), then we have $S_i <_{\text{lex}} S_j$, iff $S[i + \textsc{Lcp}_S(i, j)] < S[j + \textsc{Lcp}_S(i, j)]$ holds. Note that the statement is only true for guarded strings, where no suffix can be the prefix of another suffix. Conveniently, the right sentinel ensures that when determining the LCP we do not need to check if we reached the end of the string.

Even though not explicitly stated in xss-bps, everytime we compare two suffixes in line 6, we have to calculate $\textsc{Lcp}_S(H.top(), i)$, which naturally takes $\textsc{Lcp}_S(H.top(), i) + 1$ individual character comparisons (assuming that we do not use a precomputed support data for fast LCE queries). Processing a single index i can take $\Theta(n^2)$ time because we might have to compare S_i with $\Theta(n)$ other suffixes, and on average each suffix comparisons might require $\Theta(n)$ character comparisons. We will now reduce the worst-case processing time per index from $\mathcal{O}(n^2)$ to $\Theta(|\gamma| + c)$, where γ is the longest LCP that we discover during the iteration, and c is the number of stack elements that we pop during the iteration. First, we use a simple example to demonstrate why unnecessary character comparisons happen. Assume that our input string is $S =$ `$axyzbaxyzcaxyz$`, and the next index we want to process is $i = 12$ with $S_{12} = \texttt{axyz\$}$, i.e. we have already calculated $\mathsf{pss}[j]$ for $j \in [1, 11]$. Below we see the state of all relevant data structures at that point in time.


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
S = $ a x y z b a x y z c a x y z $
pss = 1 2 3 4 5 2 7 8 9 7 ?

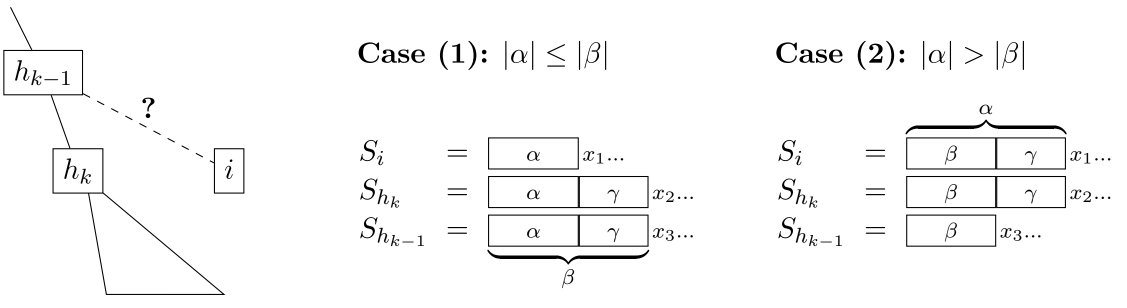
$$H : \begin{cases} 11 & S_{11} = \texttt{c}\ldots \\ 7 & S_7 = \texttt{axyzc}\ldots \\ 2 & S_2 = \texttt{axyzb}\ldots \\ 1 & S_1 = \texttt{\$}\ldots \end{cases}$$

It is not hard to see, that S_1 is the previous smaller suffix of S_{12}. On the stack H we have the indices 11, 7, 2, and 1. Therefore, xss-bps will compare S_{12} with S_{11}, S_7, S_2, and S_1, which takes 1, 5, 5, and 1 individual character comparisons respectively. A step-by-step walk through the algorithm reveals which comparisons are unnecessary. First, we compare S_{11} with S_{12}. Since the first character is already a mismatch ($S[11] = \texttt{c} > \texttt{a} = S[12]$), we have $S_{11} >_{\text{lex}} S_{12}$. Therefore, we pop 11 and the new topmost element is 7. Again, we compare S_7 and S_{12}, for which we find a shared prefix of length $\textsc{Lcp}_S(7, 12) = 4$ with a mismatch $S[7+4] = \texttt{c} > \texttt{\$} = S[12+4]$. We pop 7, and 2 becomes the topmost element on the stack. Now, since 7 was lying right on top of 2, we must have compared S_2 and S_7 in the past, for which we calculated $\textsc{Lcp}_S(2, 7) = 4$. Clearly, when we have to compare S_{12} with S_2, we have a shared prefix of length $\textsc{Lcp}_S(2, 12) \geq \min(\textsc{Lcp}_S(2, 7), \textsc{Lcp}_S(7, 12)) = 4$. Had we actually stored $\textsc{Lcp}_S(2, 7)$ instead of immediately discarding it, we could have skipped the first four character comparisons when comparing S_2 and S_{12}.

4.3.1 Skipping Previously Computed Prefixes

As we have seen, memorizing the length of the LCP between adjacent stack elements allows us to speed up future suffix comparisons. Lets look at this more formally. Assume that we are currently in iteration i of the outer loop of xss-bps, and just popped index h_k because $S_{h_k} >_{\text{lex}} S_i$ holds. Let h_{k-1} be the new topmost element on the stack. In terms of the PSS tree, we know that i is not a descendant of h_k. Next, we want to find out if i is a child of h_{k-1}, or more precisely, if i is the right sibling of h_k (see left side of the figure below).

The next suffix comparison that we have to evaluate is $S_{h_{k-1}} >_{\text{lex}} S_i$. Let α be the longest common prefix between S_{h_k} and S_i, and let β be the longest common prefix between $S_{h_{k-1}}$ and S_{h_k}. Assume that at this point we have knowledge of $|\alpha|$ and $|\beta|$. There are two cases to consider:


h_{k-1}
?
h_k
i
Case (1): $|\alpha| \leq |\beta|$
$S_i = \alpha\ x_1...$
$S_{h_k} = \alpha\ \gamma\ x_2...$
$S_{h_{k-1}} = \alpha\ \gamma\ x_3...$
β
Case (2): $|\alpha| > |\beta|$
α
$S_i = \beta\ \gamma\ x_1...$
$S_{h_k} = \beta\ \gamma\ x_2...$
$S_{h_{k-1}} = \beta\ x_3...$

Case (1): The string α is a prefix of β, i.e. $\beta = \alpha\gamma$ for some possibly empty string γ. Then $S_{h_{k-1}}$ and S_i share at least the prefix α (exactly prefix α, if γ is non-empty).

Case (2): The string β is a proper prefix of α, i.e. $\alpha = \beta\gamma$ for some non-empty string γ. Then $S_{h_{k-1}}$ and S_i share exactly the prefix β. Since S_{h_k} was lying on top of $S_{h_{k-1}}$, we have $S_{h_{k-1}} <_{\text{lex}} S_{h_k}$ with $S[h_{k-1} + |\beta|] < S[h_k + |\beta|] = S[i + |\beta|]$. Therefore, we also have $S_{h_{k-1}} <_{\text{lex}} S_i$.

Now assume that we want to calculate $\textsc{Lcp}_S(h_{k-1}, i)$ in order to evaluate $S_{h_{k-1}} >_{\text{lex}} S_i$. Regardless of which case actually applies, this value is at least $\min(|\beta|, |\alpha|)$. Therefore, we can simply skip the first $\min(|\beta|, |\alpha|)$ character comparisons when calculating $\textsc{Lcp}_S(h_{k-1}, i)$. If Case (1) applies, then we saved $|\alpha| = \textsc{Lcp}_S(h_k, i)$ character comparisons. If Case (2) applies, then we saved $|\beta| = \textsc{Lcp}_S(h_{k-1}, h_k)$ character comparisons, and we will immediately find a mismatch on the next character with $S[h_{k-1} + |\beta|] < S[i + |\beta|]$. Since that also implies that we break out of the inner loop of xss-bps, no further suffixes will be compared during iteration i of the outer loop. Therefore, apart from the last calculated LCP (which caused Case (2) to apply), all other LCPs calculated in iteration i of the outer loop must have been (not necessarily strictly) monotonically increasing in length. Also, since in Case (1) we always skip the full length of the previously calculated LCP, the total cost for iteration i of the outer loop becomes $\Theta(|\gamma| + c)$, where γ is the longest LCP discovered and c is the number of popped elements during the iteration. It remains to be shown how to provide the values of $|\alpha|$ and $|\beta|$ whenever they are needed. For $|\alpha|$ this is trivial: It is exactly the last LCP value that we have calculated. For $|\beta|$ we use a dedicated stack, which we describe in the next section.

4.3.2 Applying the New Knowledge to the Algorithm

In Algorithm 4.3 we see xss-bps-lcp, which is a modified version of xss-bps that uses the technique from the previous section. To store the length of computed prefixes we use a dedicated stack L (line 2). Let $h_1, \ldots, h_k$ be the elements on the index stack H at any time during the algorithm execution. Then $l_1, \ldots, l_{k-1}$ with $l_j = \textsc{Lcp}_S(h_j, h_{j+1})$ are the

Algorithm 4.2 Naive Computation of Longest Common Prefixes

Input: A guarded string S, indices i and j, optional ℓ with $S[i..i+\ell) = S[j..j+\ell)$
Output: The length of the longest common prefix between S_i and S_j

```
function $\textsc{Lcp}_S(i, j, \ell)$
    while $S[i+\ell] = S[j+\ell]$ do                          ▷ skips the first $\ell$ characters
        $\ell \leftarrow \ell + 1$
    return $\ell$

function $\textsc{Lcp}_S(i, j)$
    return $\textsc{Lcp}_S(i, j, 0)$
```

Algorithm 4.3 xss-bps-lcp

Input: A guarded string S of length n
Output: The BPS $\mathcal{B}_{\mathsf{pss}}$ of the PSS tree of S

```
$H \leftarrow$ empty stack                   ▷ contains $h_1, \dots, h_k$, such that $h_j = \mathsf{pss}[h_{j+1}]$
$L \leftarrow$ empty stack                ▷ contains $l_1, \dots, l_{k-1}$, such that $l_j = \textsc{Lcp}_S(h_j, h_{j+1})$
$H.push(1)$
$\mathcal{B}_{\mathsf{pss}} \leftarrow$ "(("
for $i = 2$ to $n - 1$ do
    $|\alpha| \leftarrow \textsc{Lcp}_S(H.top(), i)$                 ▷ first comparison cannot skip characters
    while $S[H.top() + |\alpha|] > S[i + |\alpha|]$ do
        $\mathcal{B}_{\mathsf{pss}} \leftarrow \mathcal{B}_{\mathsf{pss}} \cdot$ ")"
        $|\beta| \leftarrow L.top()$
        $H.pop()$
        $L.pop()$
        $|\alpha| \leftarrow \textsc{Lcp}_S(H.top(), i, \min(|\beta|, |\alpha|))$        ▷ skip the already known prefix
    $\mathcal{B}_{\mathsf{pss}} \leftarrow \mathcal{B}_{\mathsf{pss}} \cdot$ "("
    $H.push(i)$
    $L.push(|\alpha|)$
while $H$ is not empty do
    $\mathcal{B}_{\mathsf{pss}} \leftarrow \mathcal{B}_{\mathsf{pss}} \cdot$ ")"
    $H.pop()$
$\mathcal{B}_{\mathsf{pss}} \leftarrow \mathcal{B}_{\mathsf{pss}} \cdot$ "())"
return $\mathcal{B}_{\mathsf{pss}}$
```

elements of L. We make sure that this property of L is satisfied at all times: Whenever we push an element i onto $H.top()$, we also push $\textsc{Lcp}_S(H.top(), i)$ onto the LCP stack (lines 14–15), and whenever we pop an element of H we also pop an element of L (lines 10–11). A fundamental difference between xss-bps and xss-bps-lcp is, that we become more explicit about the suffix comparison. Instead of writing $S_{H.top()} >_{\text{lex}} S_i$ in line 7, we split the comparison into the calculation of the length of the LCP α between $S_{H.top()}$ and S_i, and the comparison of $S[H.top() + |\alpha|]$ and $S[i + |\alpha|]$ (lines 6, 7, and 12). When processing i, we only need to calculate the first LCP value without skipping characters (line 6). For each following comparison, we use the known LCP length $|\beta|$ of the two topmost index-stack elements as shown in the previous section (line 12). We deploy Algorithm 4.2 to calculate the LCP values. Everything else remains unchanged from xss-bps. At this point we want to emphasize, that using the LCP stack only reduces the worst case processing time *per iteration of the outer loop* to $\Theta(|\gamma| + c)$. The tightest possible bound for xss-bps-lcp is still $\mathcal{O}(n^2)$ (for example for the text "\$a^{n-2}\$").

Considering that maintaining an additional stack means both memory and execution time overhead, xss-bps-lcp only outperforms xss-bps, if the input string offers high LCP values. However, deploying the separate LCP stack is an important step towards a linear time solution, as it will also be used by our final algorithm.

4.3.3 Analyzing the Cost

Before we explain how to achieve linear time, we first need to gain a better understanding of the time that we need per iteration. The bound of $\Theta(|\gamma| + c)$ consists of three components:

1. We pop exactly c elements during the iteration, which takes $\Theta(c)$ time (assuming constant time pop operations).

2. We compute one LCP value for each popped element, and one additional LCP value that causes the condition in line 7 to no longer hold. For each LCP value, we first find between zero and $|\gamma|$ matching characters, and then one mismatch. For now, we only consider the $c + 1$ mismatches, which take $\Theta(c)$ time.

3. Finally, we also consider the matching character comparisons. As seen before, the LCP values per iteration are non-decreasing (except for possibly the last one). Whenever Case (1) applies (see Section 4.3.1), we skip the full length of the previously computed LCP. If Case (2) applies, then we immediately find a mismatch and break out of the loop, i.e. no matching character comparison occurs. Thus, the total number of matching character comparisons in the entire iteration is exactly $|\gamma|$, which means that they take $\Theta(|\gamma|)$ time.

Keeping in mind that we want to achieve linear time, the first two components are not problematic. Since c is the number of popped elements in the iteration, and we naturally pop each element at most once, it follows that the total cost of the first two components for *all* iterations is $\mathcal{O}(n)$. The critical factor is third component, which we will take care of in the next chapter.

Chapter 5

Achieving Linear Time

We have seen in Section 4.3.3, that the critical time component of each iteration of xss-bps-lcp is $\Theta(|\gamma|)$, where γ is the longest LCP that we discovered during the iteration. Without this cost, we would already achieve linear time. However, because we found an LCP of length $|\gamma|$, we also gained some important information about the input string. In this chapter we show how to exploit this information, such that whenever we find an LCP of length $|\gamma|$, we can fast-forward through the next $z = \Omega(|\gamma|)$ iterations of our algorithm. We achieve this by using two techniques, where after each iteration of the outer loop of xss-bps-lcp we can always use exactly one of the two:

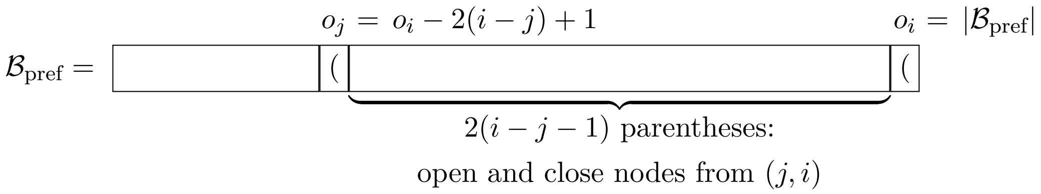
Run Extension: We found a *run* in the input string. This allows us to fast-forward through $z = \Theta(|\gamma|)$ iterations.

Amortized Look-Ahead: We did not find a run in the input string. However, we can still fast-forward through $z = \Omega(|\gamma|)$ iterations (where sometimes this might even be $z = \omega(|\gamma|)$).

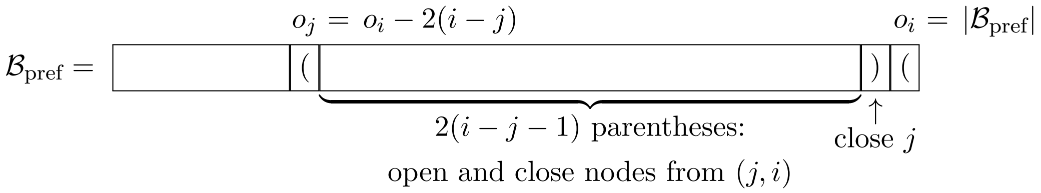
Regardless of which technique we actually use, the additional time spent on fast-forwarding through the iterations is $\Theta(z)$. Now assume that we just finished iteration i of the outer loop of xss-bps-lcp, during which we found the longest LCP of length $|\gamma|$. Also, assume that we used one of the techniques to fast-forward through the next z iterations, i.e. we can continue with iteration $i+z$. Then the total time that we spent on iterations i to $i+z-1$ is $\Theta(|\gamma|) + \Theta(z)$ (the critical cost of iteration i, plus the cost of either the run extension or the amortized look-ahead; ignoring the cost $\mathcal{O}(c)$ of iteration i, see Section 4.3.3). Since we have $|\gamma| = \mathcal{O}(z)$ and thus $\Theta(|\gamma|) + \Theta(z) = \Theta(z)$, it follows that the average time spent on each of the $z+1$ iterations is constant. Thus, if we use one of the two techniques after each iteration, then we achieve linear time. In the next two sections we use the following notation:

- S_j is the suffix that we compared S_i with while discovering the LCP γ, i.e. at some point during iteration i, the topmost element on H was j and we have $S[j..j+|\gamma|) = S[i..i+|\gamma|) = \gamma$ and $S[j+|\gamma|] \neq S[i+|\gamma|]$ (we have $S[j+|\gamma|] < S[i+|\gamma|]$, if the last suffix comparison of the iteration was covered by Case (1) of Section 4.3.1, and $S[j+|\gamma|] > S[i+|\gamma|]$, if it was covered by Case (2)). If multiple indices satisfy this criterion, we define j to be the leftmost (i.e. smallest) matching index.

- o_j is the BPS index of the opening parenthesis that belongs to node j. Let $\mathcal{B}_{\text{pref}}$ be the prefix of the BPS that we have computed after iteration i, then the last parenthesis that we have written is the opening parenthesis that belongs to node i. Since during the iteration we compared S_j and S_i, we know that we have already written both the opening and the closing parenthesis for each node from the interval (j, i). If $S_j <_{\text{lex}} S_i$ holds, then we have not written the closing parenthesis of S_j yet, and the opening parenthesis of j is located at index $o_j = |\mathcal{B}_{\text{pref}}| - 2(i-j) + 1$:


$o_j = o_i - 2(i-j) + 1$
$o_i = |\mathcal{B}_{\text{pref}}|$
$\mathcal{B}_{\text{pref}} =$
(
(
$2(i-j-1)$ parentheses:
open and close nodes from (j, i)

Otherwise, we have already written the closing parenthesis of j, and therefore we have $o_j = |\mathcal{B}_{\text{pref}}| - 2(i-j)$:


$o_j = o_i - 2(i-j)$
$o_i = |\mathcal{B}_{\text{pref}}|$
$\mathcal{B}_{\text{pref}} =$
(
)
(
$2(i-j-1)$ parentheses:
open and close nodes from (j, i)
close j

Note that there cannot be another parenthesis between the closing parenthesis of j and the opening parenthesis of i. Assume that there was another node k whose closing parenthesis should be between the closing parenthesis of j and the opening parenthesis of i, then we have $k < j$ with $S_k >_{\text{lex}} S_i$. Since the LCPs discovered in one iteration of the algorithm are non-decreasing (except for the last one; see Section 4.3), the shared prefix between S_k and S_i has length $|\gamma|$. However, we defined j to be the leftmost index with $S[j..j+|\gamma|) = S[i..i+|\gamma|) = \gamma$ and $S[j+|\gamma|] \neq S[i+|\gamma|]$. Therefore, the described k does not exist.

Given $|\gamma|$ and j, we can evaluate $S_j <_{\text{lex}} S_i$ in constant time, because we have $S_j <_{\text{lex}} S_i \Leftrightarrow S[j+|\gamma|] < S[i+|\gamma|]$. Therefore, computing o_j also takes constant time.

Since the knowledge of $|\gamma|$ and j is essential for both the run extension and the amortized look-ahead, we have to augment xss-bps-lcp with additional bookkeeping. This is shown in Algorithm 5.1, which is the final version of our algorithm. It is called xss-real (**R**un **E**xtension and **A**mortized **L**ook-ahead). We have highlighted the parts of the algorithm that differ from xss-bps-lcp. In each iteration of the outer loop, we store the values of $|\gamma|$ and j after the first LCP computation (lines 7–8). Then, whenever we discover an LCP of longer or equal length, we update the values accordingly (lines 14–16). At the end of each iteration, we use either the run extension or the amortized look-ahead to skip the next $\Omega(|\gamma|)$ iterations (lines 19–23), which we will explain in depth in the following sections.

Algorithm 5.1 xss-real

```
Input: A guarded string $S$ of length $n$
Output: The BPS $\mathcal{B}_{\mathsf{pss}}$ of the PSS tree of $S$
 1: $H \leftarrow$ empty stack                    ▷ contains $h_1, \ldots, h_k$, such that $h_j = \mathsf{pss}[h_{j+1}]$
 2: $L \leftarrow$ empty stack                    ▷ contains $l_1, \ldots, l_{k-1}$, such that $l_j = \textsc{Lcp}_S(h_j, h_{j+1})$
 3: $H.push(1)$
 4: $\mathcal{B}_{\mathsf{pss}} \leftarrow$ "(("
 5: for $i = 2$ to $n - 1$ do
 6:     $|\alpha| \leftarrow \textsc{Lcp}_S(H.top(), i)$            ▷ first comparison cannot skip characters
 7:     $|\gamma| \leftarrow |\alpha|$                      ▷ use $|\gamma|$ and $j$ to keep track of the longest LCP
 8:     $j \leftarrow H.top()$
 9:     while $S[H.top() + |\alpha|] > S[i + |\alpha|]$ do
10:         $\mathcal{B}_{\mathsf{pss}} \leftarrow \mathcal{B}_{\mathsf{pss}} \cdot$ ")"
11:         $|\beta| \leftarrow L.top()$
12:         $H.pop()$; $L.pop()$
13:         $|\alpha| \leftarrow \textsc{Lcp}_S(H.top(), i, \min(|\beta|, |\alpha|))$   ▷ skip the already known prefix length
14:         if $|\alpha| \geq |\gamma|$ then          ▷ if a longer or equally long LCP was discovered:
15:             $|\gamma| \leftarrow |\alpha|$                         ▷ update $|\gamma|$ accordingly
16:             $j \leftarrow H.top()$                          ▷ update $j$ accordingly
17:     $\mathcal{B}_{\mathsf{pss}} \leftarrow \mathcal{B}_{\mathsf{pss}} \cdot$ "("
18:     $H.push(i)$; $L.push(|\alpha|)$
19:     if $|\gamma| \geq 2(i - j)$ then                                  ▷ skip iterations
20:         $i_{\text{next}} \leftarrow RunExtension(S, H, L, \mathcal{B}_{\mathsf{pss}}, |\gamma|, j, i)$
21:     else
22:         $i_{\text{next}} \leftarrow AmortizedLookahead(S, H, L, \mathcal{B}_{\mathsf{pss}}, |\gamma|, j, i)$
23:     continue with iteration $i = i_{\text{next}}$
24: while $H$ is not empty do
25:     $\mathcal{B}_{\mathsf{pss}} \leftarrow \mathcal{B}_{\mathsf{pss}} \cdot$ ")"
26:     $H.pop()$
27: $\mathcal{B}_{\mathsf{pss}} \leftarrow \mathcal{B}_{\mathsf{pss}} \cdot$ "())"
28: return $\mathcal{B}_{\mathsf{pss}}$
```

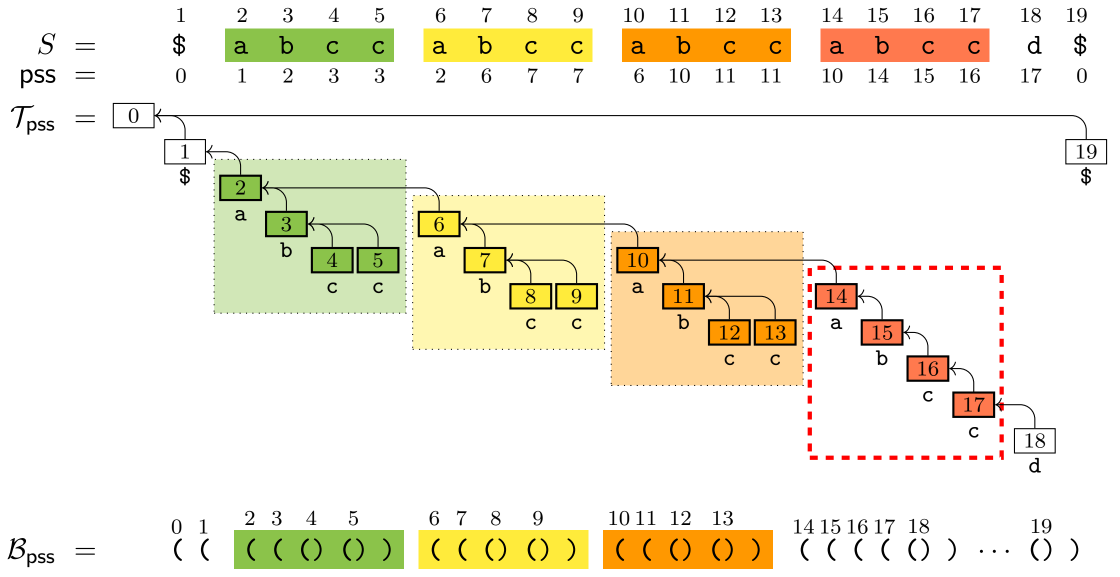


Figure 5.1: The PSS tree of the string $S =$ `$abccabccabccabccd$` and its BPS. The Lyndon word `abcc` repeats itself four times at the beginning of S_2. Each repetition, except for the last one, induces the same structure within the PSS tree. This structure is represented by the parentheses sequence `((()())`, which therefore repeats itself three times in the BPS.

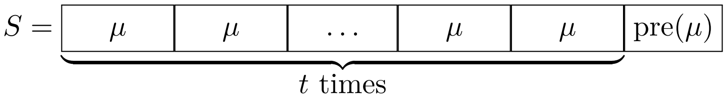
5.1 Run Extension

As the name suggests, the run extension exploits properties of *runs*, or more precisely runs of Lyndon words. The general idea is, that a repeating Lyndon word in the input string implies a repeating structure within the PSS tree, and thus also a run in its BPS. Therefore, instead of expensively recomputing the same part of the BPS for every repetition, we can simply create copies of the respective substring of the BPS. After that, we only have to ensure that the stacks H and L are updated to the correct state. Figure 5.1 shows how the run of the Lyndon word `abcc` in the string `$abccabccabccabccd$` causes a run in the BPS of the PSS tree. The run extension is applicable, iff $|\gamma| \geq 2(i-j)$ holds (see Algorithm 5.1, line 19), and skips at least $|\gamma|/3 = \Omega(|\gamma|)$ iterations.

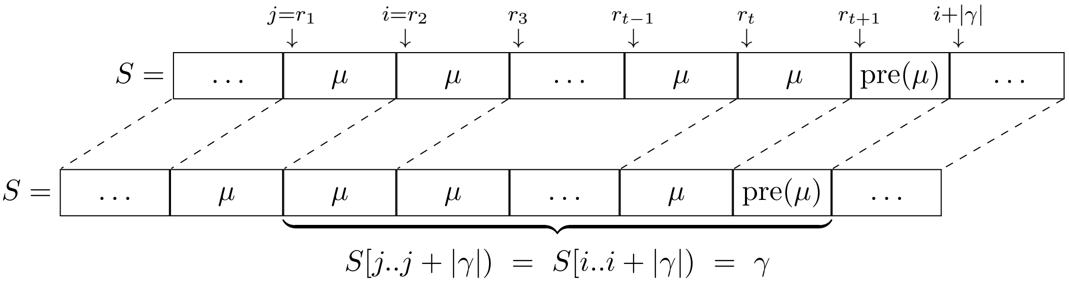
5.1.1 Properties of Lyndon Runs

Before explaining the run extension, we formally define Lyndon runs. Also, we will show some general properties of Lyndon runs within the input string.

5.1.1 Definition (Lyndon Run). We say that a string S is a *Lyndon run* of t *repetitions*, if it has the form $S = \mu^t \cdot \mathrm{pre}(\mu)$ with $t \geq 2$, where μ is a Lyndon word and $\mathrm{pre}(\mu)$ is a proper (possibly empty) prefix of μ. The length $|\mu|$ of the repeating Lyndon word is called *period* of the run.



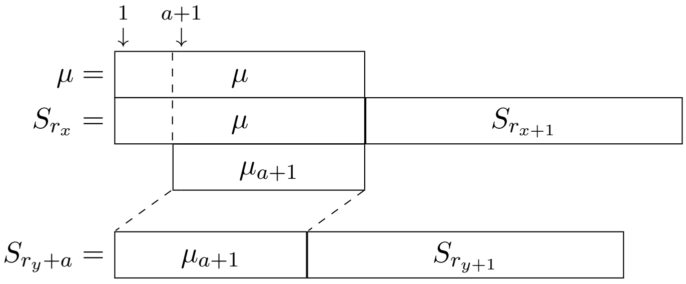
For example, the string $S = \mathtt{xyz\,xyz\,xyz\,xyz\,xy}$ is a Lyndon run of $t = 4$ repetitions with period $|\mu| = |\mathtt{xyz}| = 3$. We already claimed, that the run extension is applicable iff $|\gamma| = \textsc{Lcp}_S(j, i) \geq 2(i - j)$ holds. Now we assume that this condition is satisfied, and define $\mu = S[j..i)$. Note that from Corollary 4.1.3 follows, that μ is a Lyndon word. As visualized below, the substring $S[j..i + |\gamma|)$ is a Lyndon run:



The Lyndon word μ fits exactly $\lfloor |\gamma|/|\mu| \rfloor$ times into the shared prefix $S[i..i + |\gamma|) = \gamma$. Additionally, we have to count the first occurrence $S[j..i)$ as one repetition. Therefore, the number of repetitions is $t = \lfloor |\gamma|/|\mu| \rfloor + 1$. From $|\gamma| \geq 2|\mu|$ follows, that $t \geq 3$ holds. The starting positions of the repetitions are $r_1 = j$, $r_2 = i$, and $r_x = j + (x - 1) \cdot |\mu|$ for $x \in [3, t]$. The first index after the last repetition is $r_{t+1} = j + t \cdot |\mu|$. Now we show that each repetition of μ, except for the last one, induces the same structure in the PSS tree.

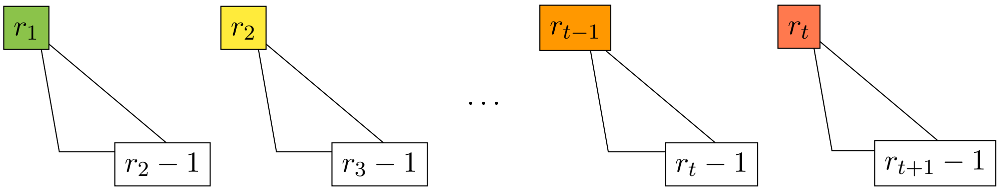
5.1.2 Lemma. *Let $x, y \in [1, t]$ and $a \in [1, |\mu|)$. We have $S_{r_x} <_{\mathrm{lex}} S_{r_y + a}$.*

Proof. Since r_x and r_y are starting positions of repetitions of μ, we know that μ is a prefix of S_{r_x} and S_{r_y}. Also, μ_{a+1} is a prefix of S_{r_y+a}:



Each proper non-empty suffix of μ is lexicographically larger than μ, because μ is a Lyndon word (Lemma 2.1.7). Therefore, we have $\mu <_{\text{lex}} \mu_{a+1}$, and thus $S_{r_x} <_{\text{lex}} S_{r_y+a}$. □

The lemma characterizes a structural property of the PSS tree: For an arbitrary repetition $S[r_x..r_{x+1})$ of μ, the nodes from the interval (r_x, r_{x+1}) are descendants of r_x. Therefore, the interval $[r_x, r_{x+1})$ is represented by a subtree that is rooted in r_x, and has $r_{x+1} - 1$ as its rightmost leaf.

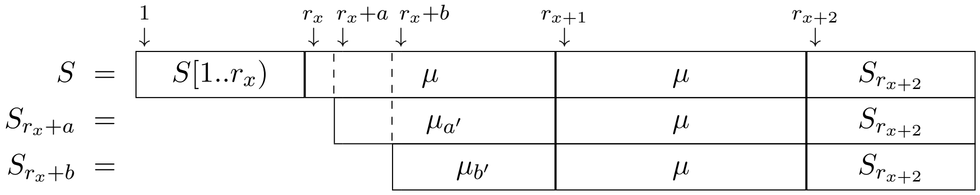


It remains to be shown that all of these structures, except for the last one, are isomorphic. As we have seen many times throughout the previous chapters, the structure of the PSS tree essentially depends on the outcome of suffix comparisons. To show that the structures are isomorphic, we prove that the suffix comparisons on which they depend have the same outcome for all repetitions of the Lyndon word.

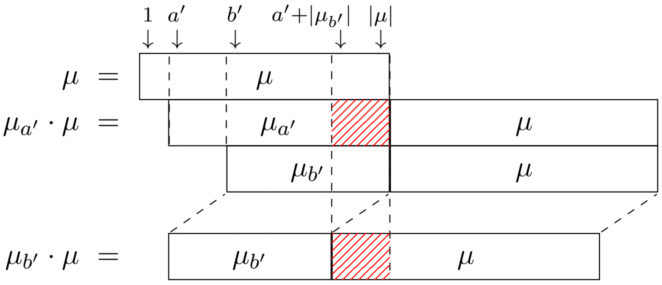
5.1.3 Lemma. *Let $x, y \in [1, t)$ and $a, b \in [0, |\mu|)$ with $a < b$. It holds:*

$$S_{r_x+a} <_{\text{lex}} S_{r_x+b} \iff S_{r_y+a} <_{\text{lex}} S_{r_y+b}$$

Proof. Let $a' = a + 1$ and $b' = b + 1$. There are at least two repetitions of μ at the beginning of S_{r_x}. Therefore, $\mu_{a'} \cdot \mu$ and $\mu_{b'} \cdot \mu$ are prefixes of S_{r_x+a} and S_{r_x+b} respectively:



This allows us to show, that the strings $\mu_{a'} \cdot \mu$ and $\mu_{b'} \cdot \mu$ have a guaranteed mismatch.



Consider the two hatched areas in the drawing above. The upper area highlights the suffix $\mu_{a'+|\mu_{b'}|}$ of μ. Let $c = |\mu| - (a' + |\mu_{b'}|) + 1$ be the length of this suffix. 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 (otherwise we contradict Lemma 2.1.7). It follows, that the two highlighted areas are not equal, i.e. we have $\mu_{a'+|\mu_{b'}|} \neq \mu[1..c]$. This guarantees that $\mu_{a'} \cdot \mu$ and $\mu_{b'} \cdot \mu$ have a mismatch within the first $|\mu_{a'}|$ characters. Since the mismatch occurs, appending an arbitrary string to $\mu_{a'} \cdot \mu$ and $\mu_{b'} \cdot \mu$ does not influence the outcome of a lexicographical comparison. Therefore we have:

$$\begin{aligned} \mu_{a'} \cdot \mu <_{\text{lex}} \mu_{b'} \cdot \mu &\iff \mu_{a'} \cdot \mu \cdot S_{r_{x+2}} <_{\text{lex}} \mu_{b'} \cdot \mu \cdot S_{r_{x+2}} \\ &\iff S_{r_x+a} <_{\text{lex}} S_{r_x+b} \end{aligned}$$

Clearly, the proof above also holds if we replace x with y. It follows:

$$S_{r_x+a} <_{\text{lex}} S_{r_x+b} \iff \mu_{a'} \cdot \mu <_{\text{lex}} \mu_{b'} \cdot \mu \iff S_{r_y+a} <_{\text{lex}} S_{r_y+b} \qquad \square$$

Finally, we can show the actual isomorphism:

5.1.4 Lemma. *Let $x, y \in [1, t)$ and $b \in [1, |\mu|)$. It holds:*

$$\exists a \in [0, b) : (\mathsf{pss}[r_x + b] = r_x + a \ \wedge \ \mathsf{pss}[r_y + b] = r_y + a)$$

Proof. From Lemma 5.1.2 follows, that all nodes from the interval (r_x, r_{x+1}) are descendants of r_x in the PSS tree. Particularly, this means that there is some $a \in [0, b)$ with $\mathsf{pss}[r_x + b] = r_x + a$. By definition of previous smaller suffixes, we have $S_{r_x+a} <_{\text{lex}} S_{r_x+b}$, and all suffixes that begin within $S(r_x + a..r_x + b)$ are lexicographically larger than S_{r_x+b}. These suffix comparisons are covered by Lemma 5.1.3:

$$\begin{aligned} & \forall c \in (a, b) : S_{r_x+c} >_{\text{lex}} S_{r_x+b} \wedge S_{r_x+a} <_{\text{lex}} S_{r_x+b} \\ \underset{\textit{Lemma } 5.1.3}{\iff} & \forall c \in (a, b) : S_{r_y+c} >_{\text{lex}} S_{r_y+b} \wedge S_{r_y+a} <_{\text{lex}} S_{r_y+b} \end{aligned}$$

By definition of previous smaller suffixes, this is equivalent to $\mathsf{pss}[r_y + b] = r_y + a$. □

Thus, all repetitions except for the last one induce identical structures in the PSS tree. This is the fundamental insight that allows us to skip iteration by copying the repeating part of the PSS tree (or more precisely its BPS). However, we still do not know how the isomorphic structures are connected within the tree. There are only two possible scenarios, which depend on the so called *direction* of the Lyndon run.

5.1.5 Definition. We call $S[j..i + |\gamma|)$ *increasing Lyndon run*, iff $S_{r_1} <_{\text{lex}} S_{r_2}$ holds, and *decreasing Lyndon run*, otherwise.

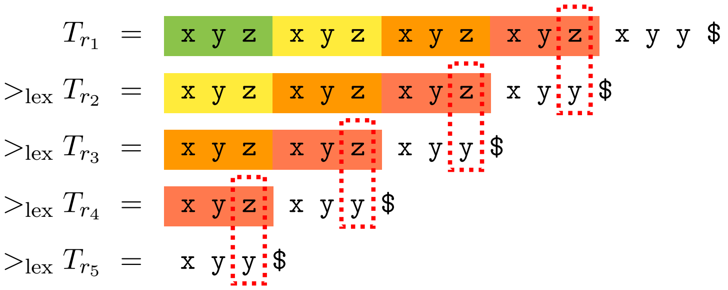
The string in Figure 5.1 contains an increasing run, because $S_{r_1} =$ `abccabccabccabccd$` is lexicographically smaller than $S_{r_2} =$ `abccabccabccd$`. On the other hand, the string $T =$ `$xyzxyzxyzxyzxyy$` contains a decreasing run, because $T_{r_1} =$ `xyzxyzxyzxyzxyy$` is lexicographically larger than $T_{r_2} =$ `xyzxyzxyzxyy$`. The names *increasing* and *decreasing* are motivated by the fact, that the repetitions of the run are monotonically increasing or decreasing in terms of the lexicographical order.

5.1.6 Observation. We have $S_{r_1} <_{\text{lex}} S_{r_2} <_{\text{lex}} S_{r_3} <_{\text{lex}} \ldots <_{\text{lex}} S_{r_{t+1}}$ for increasing runs. This holds because S_{r_1} and S_{r_2} share the prefix μ.

$$S_{r_1} <_{\text{lex}} S_{r_2} \iff \mu \cdot S_{r_2} <_{\text{lex}} \mu \cdot S_{r_3} \iff S_{r_2} <_{\text{lex}} S_{r_3}$$

By repeatedly applying the implication above, we get the chain $S_{r_1} <_{\text{lex}} \cdots <_{\text{lex}} S_{r_{t+1}}$. For decreasing runs we have $S_{r_1} >_{\text{lex}} \ldots >_{\text{lex}} S_{r_{t+1}}$ instead.

Looking at the previously used example string $T =$ `$xyzxyzxyzxyzxyy$` and its decreasing run, we have $T_{r_1} >_{\text{lex}} T_{r_2} >_{\text{lex}} T_{r_3} >_{\text{lex}} T_{r_4} >_{\text{lex}} T_{r_5}$. As visualized below, the same mismatch is responsible for all inequalities of the chain:


T_{r_1} = x y z x y z x y z x y z x y y $
$>_{\text{lex}} T_{r_2}$ = x y z x y z x y z x y y $
$>_{\text{lex}} T_{r_3}$ = x y z x y z x y y $
$>_{\text{lex}} T_{r_4}$ = x y z x y y $
$>_{\text{lex}} T_{r_5}$ = x y y $

Using the new definitions and insights, we can finally show how to skip iterations by using the run extension. The procedure for increasing runs and decreasing runs differs slightly. In both cases, we will skip $r_t - r_2$ iterations, i.e. the next regular iteration that we have to perform after the run extension is iteration $i_{\text{next}} = r_t + 1$. Note that we can determine if the run is increasing or decreasing in constant time: Since γ is exactly the LCP between S_{r_1} and S_{r_2}, we have $S_{r_1} <_{\text{lex}} S_{r_2} \Leftrightarrow S[r_1 + |\gamma|] < S[r_2 + |\gamma|]$.

5.1.2 Extending Increasing Runs

Take any $x \in [2, t]$, then following Lemma 5.1.2 we have $S_{r_{x-1}+a} >_{\text{lex}} S_{r_x}$ for $a \in [1, |\mu|)$. Since in increasing runs we also have $S_{r_{x-1}} <_{\text{lex}} S_{r_x}$, we know that $\mathsf{pss}[r_x] = r_{x-1}$ holds.

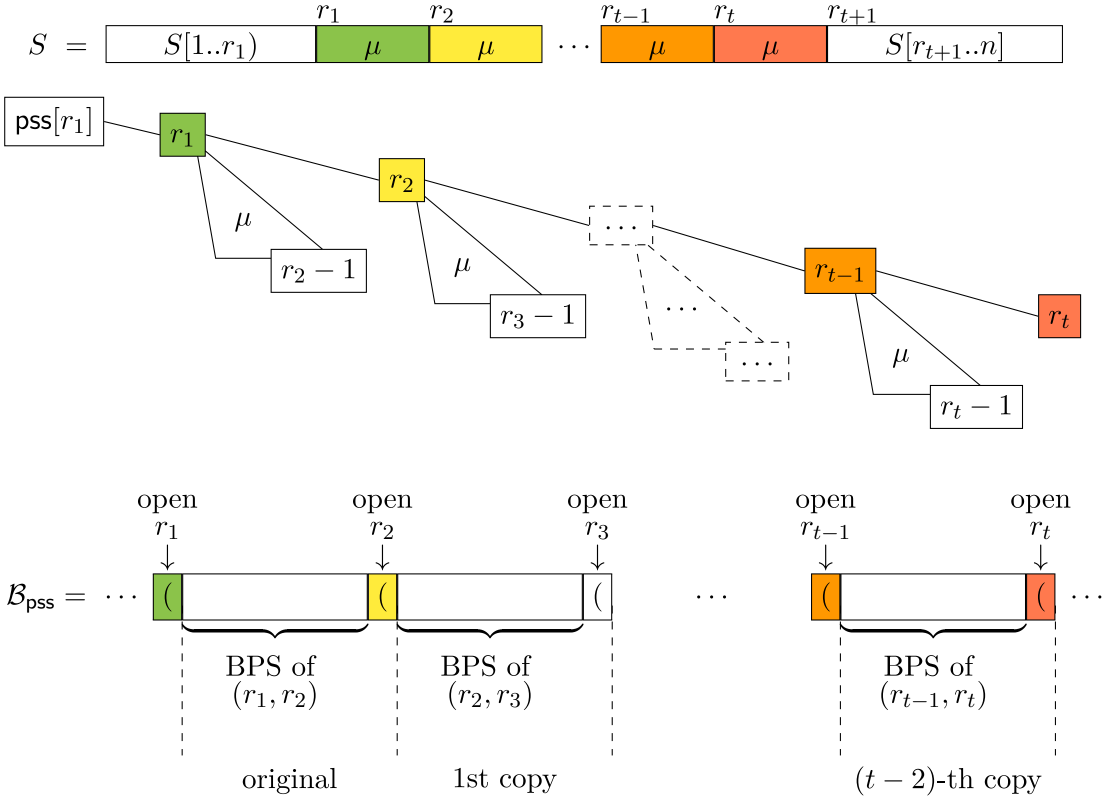


Figure 5.2: The increasing run of the Lyndon word μ induces repeated structures in the PSS tree. The t repetitions of μ cause $t-1$ repetitions of a subsequence in the BPS. As shown in Section 5.1, the BPS of interval (r_{x-1}, r_x) is identical for all $x \in [2, t]$. Since the run is increasing, we have a path $(r_1, r_2, \ldots, r_t)$ in the PSS tree.

This results in a chain of previous smaller suffix relations between the indices $r_1, \ldots, r_t$. Therefore, as seen in in Figure 5.2, there is also a chain of edges that connects the structures that are induced by the repetitions of μ. In the BPS of the PSS tree, the opening parenthesis of node r_x is always preceded by a balanced sequence of $2(|\mu| - 1)$ parentheses that represent the interval (r_{x-1}, r_x). This sequence is balanced because the suffixes $S_{r_{x-1}}$ and S_{r_x} are lexicographically smaller than all suffixes that start within the interval (r_{x-1}, r_x) (see Lemma 5.1.2). Therefore, all nodes from this interval are encoded between the opening parentheses of r_{x-1} and r_x. At the time we finish iteration $i = r_2$ during the execution of xss-real, the last parenthesis that we have written is the opening parenthesis of r_2. We simply take the last $2|\mu| - 1$ parentheses that we have written and append them $t - 2$ times to the known prefix of the BPS, which is also visualized in Figure 5.2.

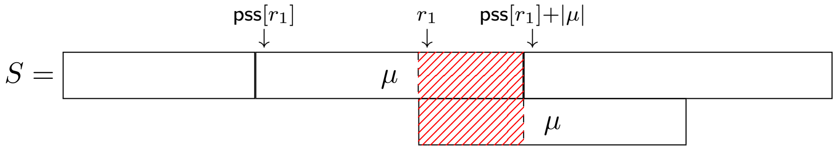
After this, the last parenthesis that we have written is the opening parenthesis of node r_t. Since after the run extension we want to continue with iteration $r_t + 1$ of our algorithm, we still have to update the stacks H and L to their correct state. Currently, the topmost element on H is r_2. Therefore, we have to push the indices $r_3, \ldots, r_t$ onto H, and the corresponding LCP values $\textsc{Lcp}_S(r_2, r_3), \ldots, \textsc{Lcp}_S(r_{t-1}, r_t)$ onto L. We compute the LCP values one at a time, and in left-to-right order, i.e. $\textsc{Lcp}_S(r_2, r_3)$ first and $\textsc{Lcp}_S(r_{t-1}, r_t)$ last. Looking at three adjacent repetitions beginning at indices r_{x-1}, r_x, and r_{x+1}, we clearly have $\textsc{Lcp}_S(r_x, r_{x+1}) = \textsc{Lcp}_S(r_{x-1}, r_x) - |\mu|$. Therefore, we can compute each LCP value in constant time.

5.1.3 Extending Decreasing Runs

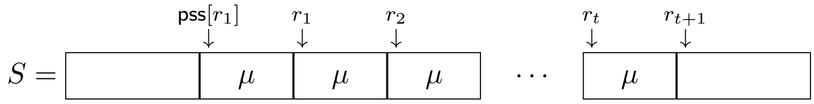
Take any $x \in [2, t]$, then following Lemma 5.1.2 we have $S_{r_{x-1}+a} >_{\text{lex}} S_{r_x}$ for all $a \in [1, |\mu|)$. Since in decreasing runs we also have $S_{r_{x-1}} >_{\text{lex}} S_{r_x}$, we know that $\mathsf{pss}[r_x] < r_{x-1}$ holds. Now we will show that in fact we have $\mathsf{pss}[r_x] = \mathsf{pss}[r_1]$.

5.1.7 Lemma. *(In this lemma we are only considering increasing runs.) Let $x \in [2, t]$. The suffixes S_{r_1} and S_{r_x} have the same PSS, and also share the same LCP with this PSS, i.e. $\mathsf{pss}[r_x] = \mathsf{pss}[r_1]$ and $\textsc{Lcp}_S(\mathsf{pss}[r_x], r_x) = \textsc{Lcp}_S(\mathsf{pss}[r_1], r_1)$ hold. The length of the LCP is bound by $\textsc{Lcp}_S(\mathsf{pss}[r_1], r_1) < |\mu|$.*

Proof. Assume that $S_{\mathsf{pss}[r_1]}$ begins with the prefix μ. If we additionally assume that $\mathsf{pss}[r_1] + |\mu| > r_1$ holds, we get the following picture:



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 contradicts Lemma 2.1.7. Thus, we have shown that $\mathsf{pss}[r_1] + |\mu| \not> r_1$ holds. Also, we cannot have $\mathsf{pss}[r_1] + |\mu| = r_1$, because then $\mathsf{pss}[r_1]$ is the starting position of another repetition of μ.



Since the run is decreasing, this would imply $S_{\mathsf{pss}[r_1]} >_{\text{lex}} S_{r_1}$, which contradicts the definition of previous smaller suffixes. It follows, that $\mathsf{pss}[r_1] + |\mu| < r_1$ holds. Consequently, $S_{\mathsf{pss}[r_1]+|\mu|}$ begins somewhere between $\mathsf{pss}[r_1]$ and r_1, and we have $S_{\mathsf{pss}[r_1]+|\mu|} >_{\text{lex}} S_{r_1}$. However, this leads to a contradiction:

$$\begin{aligned} S_{\mathsf{pss}[r_1]} <_{\text{lex}} S_{r_1} &\iff \mu \cdot S_{\mathsf{pss}[r_1]+|\mu|} <_{\text{lex}} \mu \cdot S_{r_2} \\ &\iff \quad S_{\mathsf{pss}[r_1]+|\mu|} <_{\text{lex}} S_{r_2} \\ &\underset{S_{r_1} >_{\text{lex}} S_{r_2}}{\Longrightarrow} \quad S_{\mathsf{pss}[r_1]+|\mu|} <_{\text{lex}} S_{r_1} \end{aligned}$$

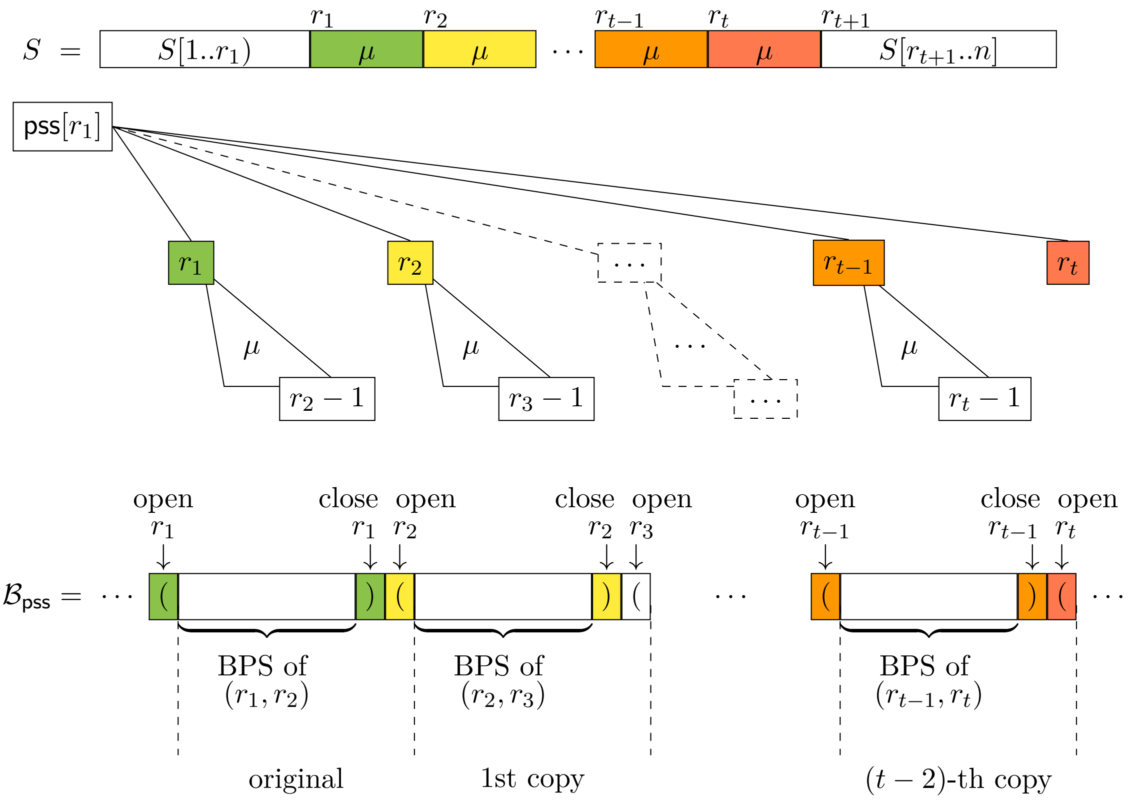


Figure 5.3: The decreasing run of the Lyndon word μ induces repeated structures in the PSS tree. The t repetitions of μ cause $t-1$ repetitions of a subsequence in the BPS. As shown in Section 5.1, the BPS of interval (r_{x-1}, r_x) is identical for all $x \in [2, t]$. Since the run is decreasing, the nodes $r_1, r_2, \ldots, r_t$ are children of $\mathsf{pss}[r_1]$ in the PSS tree, such that r_x is the right sibling of r_{x-1}.

It follows that μ is not a prefix of $S_{\mathsf{pss}[r_1]}$. Thus, we have $\textsc{Lcp}_S(\mathsf{pss}[r_1], r_1) < |\mu|$. Since all other suffixes S_{r_x} with $x \in [2, t]$ begin with the prefix μ as well, we have $\textsc{Lcp}_S(\mathsf{pss}[r_1], r_x) = \textsc{Lcp}_S(\mathsf{pss}[r_1], r_1)$. This also means, that the suffixes $S_{\mathsf{pss}[r_1]}$ and S_{r_1} compare in the same way as $S_{\mathsf{pss}[r_1]}$ and S_{r_x}. We have $S_{\mathsf{pss}[r_1]} <_{\text{lex}} S_{r_x}$, and therefore $\mathsf{pss}[r_x] = \mathsf{pss}[r_1]$. □

As a direct consequence of this lemma, all nodes r_x with $x \in [2, t]$ are children of $\mathsf{pss}[r_1]$, as visualized in Figure 5.3. If we look at the BPS of the PSS tree, the opening parenthesis of node r_x is preceded by the closing parenthesis of node r_{x-1}, which in turn is preceded by a balanced sequence of $2(|\mu| - 1)$ parentheses that represent the interval (r_{x-1}, r_x). We simply take the last $2|\mu|$ parentheses that we have written and append them $t-2$ times to the known prefix of the BPS. Bringing the stack H into the correct state is easy: We simply replace the topmost element on H (which is $r_2 = i$) with r_t. Currently, the topmost element on L is $\textsc{Lcp}_S(\mathsf{pss}[r_2], r_2)$. We have already shown that $\textsc{Lcp}_S(\mathsf{pss}[r_2], r_2) = \textsc{Lcp}_S(\mathsf{pss}[r_t], r_t)$ holds. Therefore, the stack L does not need to be altered. Thus, we have skipped the next $r_t - r_2$ iterations.

5.1.4 Skipping $\Omega(|\gamma|)$ Iterations

We have shown that for both increasing and decreasing runs we can skip $r_t - r_2$ iterations. Remember that γ contains exactly $(t-1)$ full repetitions of μ, as well as an additional proper prefix of μ. Thus, we have $|\gamma| < t \cdot |\mu|$. It follows, that the number of skipped iterations is bound by $\Omega(|\gamma|)$:

$$r_t - r_2 \;=\; (t-2) \cdot |\mu| \;>\; \frac{(t-2) \cdot |\mu|}{t \cdot |\mu|} \cdot |\gamma| \;\geq\; |\gamma|/3 \;=\; \Omega(|\gamma|)$$

The time needed for performing the run extension is $\mathcal{O}(|\gamma|)$, which is dominated by the time needed for copying either $(2|\mu|-1) \cdot (t-2) < 2|\gamma|$ parentheses for increasing runs, or $2|\mu| \cdot (t-2) < 2|\gamma|$ parentheses for decreasing runs respectively. Therefore, we satisfy the requirements that we stated at the beginning of Chapter 5, i.e. each skipped iteration takes constant time on average.

It is noteworthy, that during the execution of xss-real we actually detect *all* Lyndon runs of at least three repetitions:

5.1.8 Lemma. *Let $r_1, \ldots, r_t$ be the starting positions of a Lyndon run that has $t \geq 3$ repetitions. If iteration r_2 of xss-real is a regular iteration, i.e. if we do not skip iteration r_2, then we will detect the Lyndon run in this iteration. The longest LCP that we discover in iteration r_2 is $\textsc{Lcp}_S(r_1, r_2)$, and we only find this LCP between S_{r_1} and S_{r_2}. Therefore, the run extension will be used after iteration r_2, allowing us to skip $\Omega(\textsc{Lcp}_S(r_1, r_2))$ iterations.*

Proof. First, assume that the run is increasing. In iteration r_2 we only compare S_{r_2} against suffixes that begin within $[\mathsf{pss}[r_2], r_2) = [r_1, r_2)$. Clearly, the longest LCP between S_{r_2} and any S_a with $a \in [r_1, r_2)$ is exactly the one between S_{r_2} and S_{r_1}. For decreasing runs, we have to consider all suffixes starting in $[r_1, r_2)$, and additionally the suffix $S_{\mathsf{pss}[r_2]}$. However, we have already shown that $\textsc{Lcp}_S(\mathsf{pss}[r_2], r_2)$ is less than the length of one repetition (Lemma 5.1.7). Therefore, regardless of the direction of the run, the longest LCP discovered in iteration r_2 is $\textsc{Lcp}_S(r_1, r_2)$. □

5.1.5 Presence of Lyndon Run Indices on *H*

We conclude the section by showing some additional properties of Lyndon runs that are particularly useful for our algorithmic setting. We will use these properties to prove the time bound of the amortized look-ahead, as well as the the space bound of our succinct representation of the stack L.

Lyndon runs not only induce isomorphic structures in the PSS tree, but also cause specific patterns on the index stack H. The presence of some indices of a Lyndon run on H can imply the absence of others.

5.1.9 Lemma. *Let $r_1, \ldots, r_t$ be the starting positions of a Lyndon run that has $t \geq 3$ repetitions and period $|\mu| = (r_2 - r_1)$. Let $x \in [1, t)$ and $a \in [1, |\mu|)$. If at any point in time during the algorithm execution the stack H contains the index $r_x + a$, then the topmost element on H is smaller than r_{x+1}.*

Proof. From Lemma 5.1.2 follows, that $S_{r_x+a} >_{\text{lex}} S_{r_{x+1}}$ holds. Therefore, once we reach iteration r_{x+1} of our algorithm, we pop $r_x + a$. Thus, if H contains $r_x + a$, we have not yet reached iteration r_{x+1}, and we have only pushed elements that are smaller than r_{x+1}. □

5.1.10 Lemma. *Let $r_1, \ldots, r_t$ be the starting positions of an increasing Lyndon run that has $t \geq 3$ repetitions. Let $x \in [1, t)$. If at any point in time during the algorithm execution the topmost element on the stack H is larger than r_x, but H does not contain r_x, then it also does not contain any index from the interval $(r_x, r_t]$.*

Proof. Let $|\mu| = (r_2 - r_1)$. Since the topmost element is larger than r_x, we must have popped r_x already. Therefore, we have processed a suffix S_a with $a > r_x$ that is lexicographically smaller than S_{r_x}. Since in increasing runs all suffixes starting within $(r_x, r_t + |\mu|]$ are lexicographically larger than S_{r_x} (Lemma 5.1.2 and Observation 5.1.6), we have $a > r_t + |\mu|$. It follows, that we popped all indices from the interval $(r_x, r_t + |\mu|]$ during iteration a of our algorithm. □

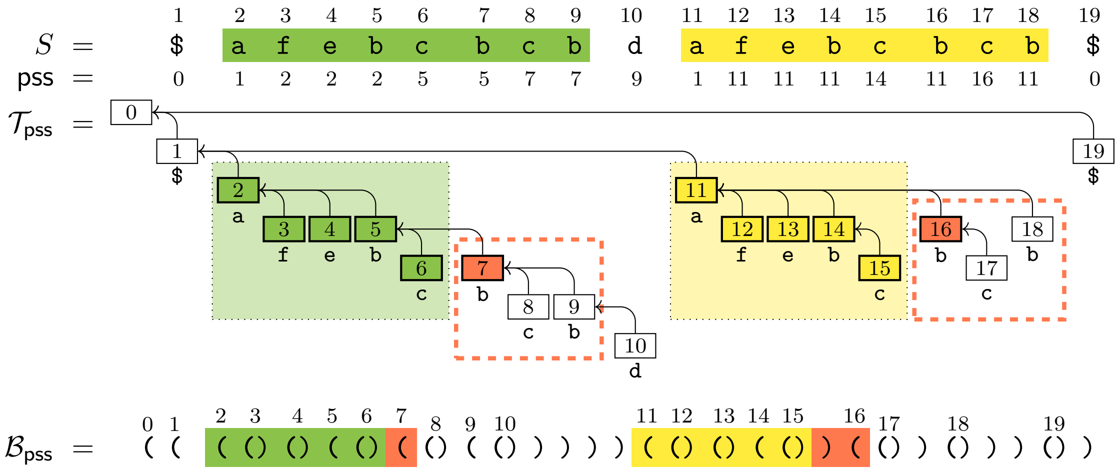


Figure 5.4: The PSS tree of the string $S =$ `$abccabccabccabccd$` and its BPS. There are two occurrences of the substring `afebcbcb`. The structures in the PSS tree that are induced by these occurrences are similar. Even though the substring has length eight, the completely identical areas in the PSS tree only contain five nodes.

5.2 Amortized Look-Ahead

It remains to be shown how to skip $\Omega(|\gamma|)$ iterations, if the run extension is not applicable, i.e. if we have $|\gamma| < 2(i-j)$. Once again, the general idea is to copy a part of the already computed prefix of the BPS. A long LCP between j and i might cause a large repeating structure within the PSS tree. Computing the LCP can be interpreted as expensively *looking ahead* in the input string S. Since we amortize this cost by skipping iterations, the new technique is called *amortized look-ahead.* While being conceptually simple, the approach is complicated in technical detail. Unfortunately, the two occurrences of $\gamma = S[i..i+|\gamma|) = S[j..j+|\gamma|)$ do not necessarily induce completely identical structures in the PSS tree. Figure 5.4 shows the PSS tree of the string $S =$ `$afebcbcbdafebcbcb$`. Even though we have the shared prefix $S[2..9] = S[11..18] =$ `afebcbcb` of length eight between S_2 and S_{11}, the respective identical parts of the PSS tree contain only five nodes. The difficulty of the amortized look-ahead is to find the part of the BPS that can be copied. Before going into detail, we will briefly outline the three main steps of the algorithmic procedure:

- First, we compute a so called *anchor* $\chi \in [1, |\gamma|]$, which indicates that we can skip exactly $\chi - 1$ iterations by using the amortized look-ahead. Similar to the way that each repetition of a run induces the same structure in the PSS tree, the anchor ensures that $S[j..j+\chi)$ and $S[i..i+\chi)$ induce the same structure in the PSS tree. If we find an anchor that is relatively small, i.e. $\chi < \lfloor|\gamma|/4\rfloor$, then we can skip $\Omega(|\gamma|)$ additional iterations by using the run extension. Otherwise, we have $\chi \geq \lfloor|\gamma|/4\rfloor$, which means that we can skip $\Omega(|\gamma|)$ iterations with the amortized look-ahead. Therefore, we always skip $\Omega(|\gamma|)$ iterations.

- We copy a part of the already computed prefix of the BPS and append it to its end. Since $S[j..j+\chi)$ and $S[i..i+\chi)$ induce the same structures in the PSS tree, we have $\mathcal{B}_{\mathsf{pss}}(o_j..o_{j+\chi-1}] = \mathcal{B}_{\mathsf{pss}}(o_i..o_{i+\chi-1}]$, where o_x denotes the BPS index of the opening parenthesis of node x.
- Lastly, we update the stacks H and L such that they reflect the changes that we made to the BPS.

We make sure, that we spend at most $\mathcal{O}(|\gamma|)$ additional time on the amortized look-ahead. This way, we guarantee that the average time per skipped iteration is constant.

5.2.1 Finding an Anchor

We already stated, that we want $S[j..j+\chi)$ and $S[i..i+\chi)$ to induce identical structures in the PSS tree. As we have seen multiple times throughout this thesis, the edges in the PSS tree depend on suffix comparisons. If we can ensure, that suffix comparisons within $S[j..j+\chi)$ have the same result as suffix comparisons within $S[i..i+\chi)$, then we also have identical structures in the PSS tree. Ideally, we want to select the largest possible χ for which there are no two offsets $a, b \in [0, \chi)$ with $S_{j+a} <_{\text{lex}} S_{j+b}$ and $S_{i+a} >_{\text{lex}} S_{i+b}$. However, we have to keep in mind that we want to spent at most $\mathcal{O}(|\gamma|)$ time. Therefore, it is not feasible to simply compute the results of all possible suffix comparisons. Thus, we need a smarter mechanism to determine the anchor.

Assume that we are looking at any two *problematic* offsets $a, b \in [0, |\gamma|)$ for which we have $S_{j+a} <_{\text{lex}} S_{j+b}$ and $S_{i+a} >_{\text{lex}} S_{i+b}$. Since we have $S[j..j+|\gamma|) = S[i..i+|\gamma|) = \gamma$, we know that the shared prefix between S_{j+a} and S_{j+b} (and also the one between S_{i+a} and S_{i+b}) must extend at least all the way to the end of $S[j..j+|\gamma|)$ (or the end of $S[i..i+|\gamma|)$, respectively). Otherwise, the first mismatch between S_{j+a} and S_{j+b} would occur in $S[j..j+|\gamma|)$, and therefore the same mismatch would occur between S_{i+a} and S_{i+b} in $S[i..i+|\gamma|)$. If a and b are small, then also the distance between $j+a$ and $j+b$ becomes small compared to the length of the shared prefix between S_{j+a} and S_{j+b}. In fact, if both a and b are from the interval $[0, \lfloor|\gamma|/4\rfloor)$, then they can only be problematic, if γ ends in a run of length at least $\lceil 3|\gamma|/4\rceil$ and period $|a-b|$. We will show how to detect runs of this kind, which then allows us to choose an anchor for which there are no problematic offsets.

Let $\ell = \lfloor|\gamma|/4\rfloor$. We split γ into the prefix $\gamma[1..\ell]$ and the suffix $\gamma_{\ell+1}$. Then, we determine whether there is a Lyndon run that begins in $\gamma[1..\ell]$ and extends to the very end of γ. Formally, we introduce extended Lyndon runs:

5.2.1 Definition (Extended Lyndon Run). Let S be a string. We say that S is an *extended Lyndon run* with period $|\mu|$, if there is a Lyndon word μ and an integer $t \geq 2$, such that S has the form $S = \mathrm{suf}(\mu) \cdot \mu^t \cdot \mathrm{pre}(\mu)$, where $\mathrm{suf}(\mu)$ and $\mathrm{pre}(\mu)$ are a proper suffix and a proper prefix of μ.

$$S = \boxed{\mathrm{suf}(\mu)}\underbrace{\boxed{\mu}\boxed{\mu}\boxed{\cdots}\boxed{\mu}\boxed{\mu}}_{t\text{ times}}\boxed{\mathrm{pre}(\mu)}$$

Consider the string γ = "az xyz xyz xyz xyz xyz xyz xyz xy" of length 25 as an example. We have $\ell = \lfloor 25/4 \rfloor = 6$, resulting in the following substrings $\gamma[1..\ell]$ and $\gamma_{\ell+1}$:

$$\gamma = \underbrace{\boxed{\texttt{az xyz x}}}_{\gamma[1..\ell]}\underbrace{\boxed{\texttt{yz xyz xyz xyz xyz xyz xy}}}_{\gamma_{\ell+1}}$$

In this case, $\gamma_{\ell+1}$ is an extended Lyndon run with $\mu = \texttt{xyz}$, $\mathrm{suf}(\mu) = \texttt{yz}$, $\mathrm{pre}(\mu) = \texttt{xy}$ and $t = 5$. Determining whether or not $\gamma_{\ell+1}$ is an extended Lyndon run takes $\mathcal{O}(|\gamma_{\ell+1}|)$ time and $\mathcal{O}(1)$ words of additional memory, using a modified version of Duval's algorithm [Duval, 1983, Algorithm 2.1], which we describe in Appendix A. Given an extended Lyndon run, the algorithm does not only detect that it is an extended Lyndon run, but also outputs the period $|\mu|$ of the run, as well as the length $|\mathrm{suf}(\mu)|$, which identifies the starting position of the first full repetition of μ.

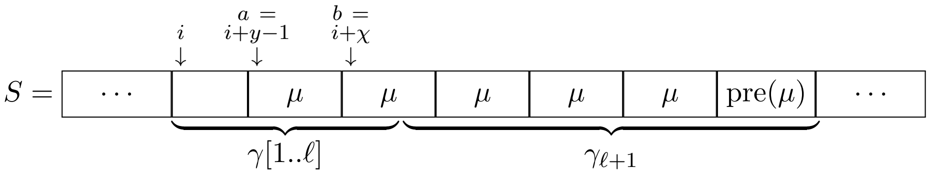
If $\gamma_{\ell+1}$ is not an extended Lyndon run, then we simply choose the anchor $\chi = \ell = \lfloor |\gamma|/4 \rfloor$. Otherwise, we try to extend the run to the left: We are now not only considering the suffix $\gamma_{\ell+1}$, but the entire string γ. We want to find the leftmost index y of γ that is the starting position of a repetition of μ. Given $|\mu|$ and $|\mathrm{suf}(\mu)|$, this can be done naively by scanning $\gamma[1..\ell]$ from right to left, which takes $\mathcal{O}(\ell)$ time. Finally, we define the anchor $\chi = \min(y + |\mu| - 1, \ell)$. For our example, we have $y = 3$ and $\chi = y + |\mu| - 1 = 5$:

$$\gamma = \underbrace{\boxed{\texttt{az }\overset{\overset{y}{\downarrow}}{\texttt{x}}\texttt{y}\overset{\overset{\chi}{\downarrow}}{\texttt{z}}\texttt{ x}}}_{\gamma[1..\ell]}\underbrace{\boxed{\texttt{yz xyz xyz xyz xyz xyz xy}}}_{\gamma_{\ell+1}}$$

For completeness, we also give an example with $\ell < y + |\mu| - 1$:

$$\gamma = \underbrace{\boxed{\texttt{abc ab}\overset{\overset{\chi}{\downarrow}}{\texttt{c}}}}_{\gamma[1..\ell]}\underbrace{\boxed{\texttt{yz }\overset{\overset{y}{\downarrow}}{\texttt{x}}\texttt{y}\overset{\overset{y+|\mu|-1}{\downarrow}}{\texttt{z}}\texttt{ xyz xyz xyz xyz xy}}}_{\gamma_{\ell+1}}$$

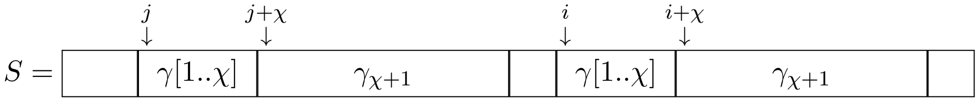
We will show later, that we can skip exactly $\chi - 1$ iterations by using the amortized look-ahead. After that, we can continue with iteration $i_{\text{next}} = i + \chi$ of xss-real. Now we show, that for small anchors $\chi < \ell$ we can skip additional $\Omega(|\gamma|)$ iterations by using the run extension. From $\chi < \ell$ follows, that we actually found an extended Lyndon run. Therefore, in the input string S we have:



Let $a = i + y - 1$ and $b = i + \chi$. Looking at the two suffixes S_a and S_b, we know that their LCP extends at least all the way to the end of γ, which means that we have $\textsc{Lcp}_S(a, b) > |\gamma_{\ell+1}| = \Omega(|\gamma|)$. Also, we clearly have $\textsc{Lcp}_S(a, b) > 2|\mu|$, because $\gamma_{\ell+1}$ contains at least two repetitions of μ. Therefore, $S[a..b + \textsc{Lcp}_S(a, b))$ is a Lyndon run as defined in Definition 5.1.1. Following Lemma 5.1.8, we will detect this run in iteration $b = i + \chi$ of xss-real. Thus, the run extension will be used immediately after the amortized look-ahead and we skip $\Omega(|\gamma|)$ additional iterations.

5.2.2 Properties of the Anchor

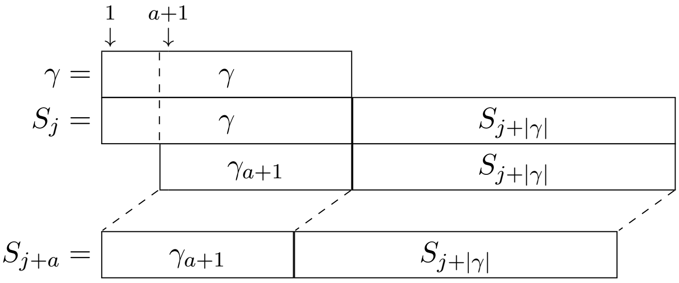
So far we only gave a vague intuition of how the anchor ensures identical structures in the PSS tree. In this section, we explain its functionality in more detail. By splitting γ into $\gamma[1..\chi]$ and $\gamma_{\chi+1}$ we get the following picture:



Now we show that the two occurrences of $\gamma[1..\chi]$ actually induce identical structures in the PSS tree. We proceed in a similar fashion as in Section 5.1.1. The lemmas and corollaries of this chapter are specific for our current algorithmic situation, which means that i, j, χ, and γ always refer to the setting displayed above.

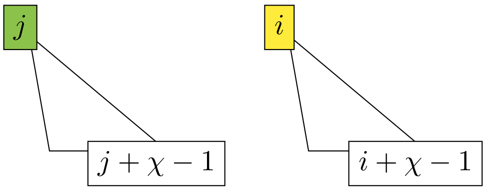
5.2.2 Lemma. *Let $a \in [1, |\gamma|)$. We have $S_j <_{\text{lex}} S_{j+a}$, and therefore* pss$[j + a] \geq j$*. The lemma also holds for i instead of j.*

Proof. Since we compared S_j and S_i during the execution of xss-real, it follows from Corollary 4.1.3 that γ is a Lyndon word.



Each proper non-empty suffix of γ is lexicographically larger than γ (Lemma 2.1.7). We have $\gamma <_{\text{lex}} \gamma_{a+1}$, and thus $S_j <_{\text{lex}} S_{j+a}$. The proof also works if we replace j with i. □

The lemma characterizes a structural property of the PSS tree: The nodes from the interval $(j, j+\chi)$ are descendants of node j. Therefore, the interval $[j, j+\chi)$ is represented by a subtree that is rooted in j, and has $j+\chi-1$ as its rightmost leaf. Analogously, the interval $[i, i+\chi)$ is represented by a subtree that is rooted in i, and has $i+\chi-1$ as its rightmost leaf.

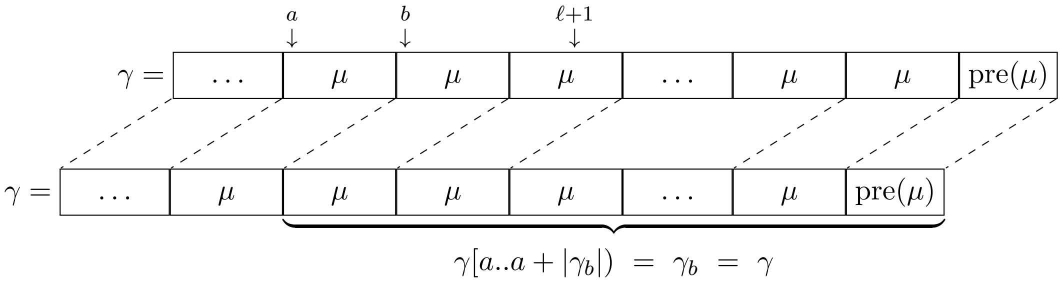


Next, we show that these two structures are actually isomorphic. We benefit from the properties of the anchor, which we exploit in the following auxiliary lemma:

5.2.3 Lemma. *Let $a, b \in [1, \chi]$ with $a < b$. At least one of the following holds:*

1. *The string $\gamma[a..b)$ is not a Lyndon word.*
2. *The string γ_b is not a prefix of γ_a.*

Proof. Assume that none of the two properties holds, i.e. $\gamma[a..b)$ is a Lyndon word and γ_b is a prefix of γ_a. Let $\mu = \gamma[a..b)$. By definition of χ we have $b \leq \chi \leq \ell = \lfloor |\gamma|/4 \rfloor$.

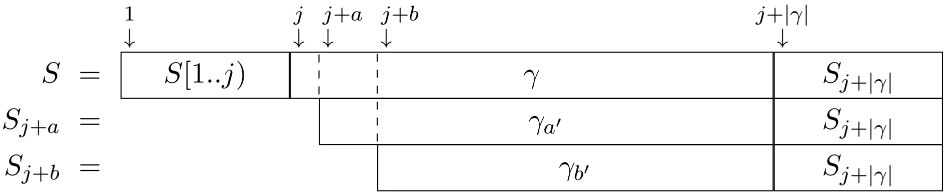


As visualized above, there is a run of the Lyndon word μ that starts at index a and extends all the way to the end of γ. The period $|\mu|$ of this run is bound by $|\mu| = b - a < \chi \leq \ell$. Since we have $|\gamma_{\ell+1}| \geq 3\ell$, there are at least two full repetitions of μ that lie within $\gamma_{\ell+1}$. This however means that $\gamma_{\ell+1}$ is an extended Lyndon run. While determining χ we would have extended the run as far as possible to the left, resulting in an anchor $\chi < b$ (see Section 5.2.1). From this contradiction follows, that at least one of the properties holds.□

5.2.4 Lemma. *Let $a, b \in [0, \chi)$ with $a < b$. At least one of the following holds:*

1. *The string $\gamma[a+1..b+1) = S[j+a..j+b) = S[i+a..i+b)$ is not a Lyndon word.*
2. *We have $S_{j+a} <_{\text{lex}} S_{j+b} \iff S_{i+a} <_{\text{lex}} S_{i+b}$.*

Proof. Let $a' = a + 1$ and $b' = b + 1$. The string γ is a prefix of S_j. Therefore, the strings $\gamma_{a'}$ and $\gamma_{b'}$ are prefixes of S_{j+a} and S_{j+b} respectively:



Now assume that $\mu = S[j+a..j+b) = \gamma[a'..b')$ is a Lyndon word. Then from Lemma 5.2.3 follows, that $\gamma_{b'}$ is not a prefix of $\gamma_{a'}$. Since this means that there is a definite mismatch between $\gamma_{a'}$ and $\gamma_{b'}$, we can append an arbitrary string to $\gamma_{a'}$ and $\gamma_{b'}$ without influencing the outcome of a lexicographical comparison. It follows:

$$\begin{aligned}\gamma_{a'} <_{\text{lex}} \gamma_{b'} &\iff \gamma_{a'} \cdot S_{j+|\gamma|} <_{\text{lex}} \gamma_{b'} \cdot S_{j+|\gamma|} \\ &\iff S_{j+a} <_{\text{lex}} S_{j+b}\end{aligned}$$

Clearly, the proof above also holds if we replace j with i. It follows:

$$S_{j+a} <_{\text{lex}} S_{j+b} \iff \gamma_{a'} <_{\text{lex}} \gamma_{b'} \iff S_{i+a} <_{\text{lex}} S_{i+b}$$ □

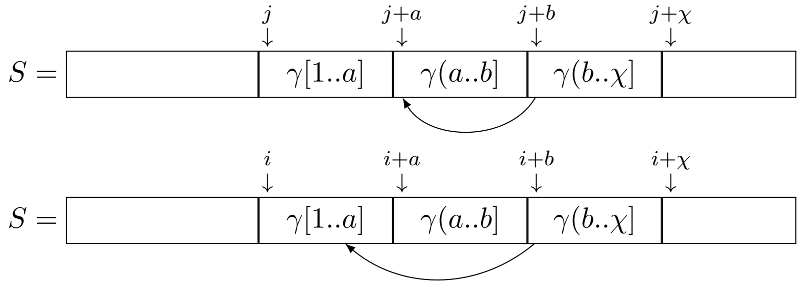
Finally, we can prove the actual isomorphism.

5.2.5 Lemma. *Let $b \in [1, \chi)$. It holds:*

$$\exists a \in [0, b) : (\mathsf{pss}[j+b] = j+a \ \wedge\ \mathsf{pss}[i+b] = i+a)$$

Proof. From Lemma 5.2.2 follows, that all nodes from the interval $(j, j+|\gamma|)$ are descendants of j in the PSS tree. Particularly, this means that there is some $a \in [0, b)$ with

$\mathsf{pss}[j+b] = j+a$. Assume that we have $(\mathsf{pss}[i+b] < i+a)$:



From Lemma 2.2.7 follows that $S[j+a..j+b)$ is a Lyndon word. By definition of previous smaller suffixes we have $S_{j+a} <_{\text{lex}} S_{j+b}$. Therefore, by applying Lemma 5.2.4 we get $S_{i+a} <_{\text{lex}} S_{i+b}$, which contradicts the assumption that $\mathsf{pss}[i+b] < i+a$ holds. Thus, we cannot have $\mathsf{pss}[i+b] < i+a$. Now assume $\mathsf{pss}[i+b] > i+a$ and let $i+a' = \mathsf{pss}[i+b]$. From Lemma 2.2.7 follows that $S[i+a'..i+b)$ is a Lyndon word. By definition of previous smaller suffixes we have $S_{i+a'} <_{\text{lex}} S_{i+b}$. Again, we can apply Lemma 5.2.4 and get $S_{j+a'} <_{\text{lex}} S_{j+b}$, which contradicts $\mathsf{pss}[j+b] = j+a < j+a'$. Thus, we also cannot have $\mathsf{pss}[i+b] > i+a$, and the only possible option is $\mathsf{pss}[i+b] = i+a$. □

Since the structures induced by the occurrences of $\gamma[1..\chi]$ are isomorphic, they are represented by the same parentheses sequence in the BPS of the PSS tree. We use the notation o_x for the BPS index of the opening parenthesis of node x. As a direct consequence of Lemma 5.2.5 we have:

5.2.6 Corollary. *It holds:* $\mathcal{B}_{\mathsf{pss}}[o_j..o_{j+\chi-1}] = \mathcal{B}_{\mathsf{pss}}[o_i..o_{i+\chi-1}]$.

5.2.3 Skipping Iterations

Finally, we can show how to skip iterations by using the amortized look-ahead. We have already shown how to find the BPS index o_j of the opening parenthesis of node j in constant time (see first section of Chapter 5). The last parenthesis that we have written is the opening parenthesis of node i at BPS index o_i. We can simply extend the known prefix of the BPS by appending a copy of $\mathcal{B}_{\mathsf{pss}}(o_j..o_{j+\chi-1}]$, as visualized in Figure 5.5. The correctness of this extension is guaranteed by Corollary 5.2.6. Appending the copy is trivial: We start at index o_j+1, and keep copying parentheses until we have copied exactly $\chi-1$ opening parentheses. As shown in Lemma 5.2.2, we have $S_j <_{\text{lex}} S_{j+a}$ for $a \in (j, j+\chi)$. Therefore, the BPS interval $\mathcal{B}_{\mathsf{pss}}[o_j..o_{j+\chi-1}]$ does not contain the closing parenthesis of node j. It follows that it contains at most $2(\chi-1)$ parentheses: Exactly χ opening parentheses for all nodes from the interval $[j, j+\chi)$, as well as at most $\chi-2$ closing parentheses for the nodes from the interval $(j, j+\chi-1)$. Thus, appending the

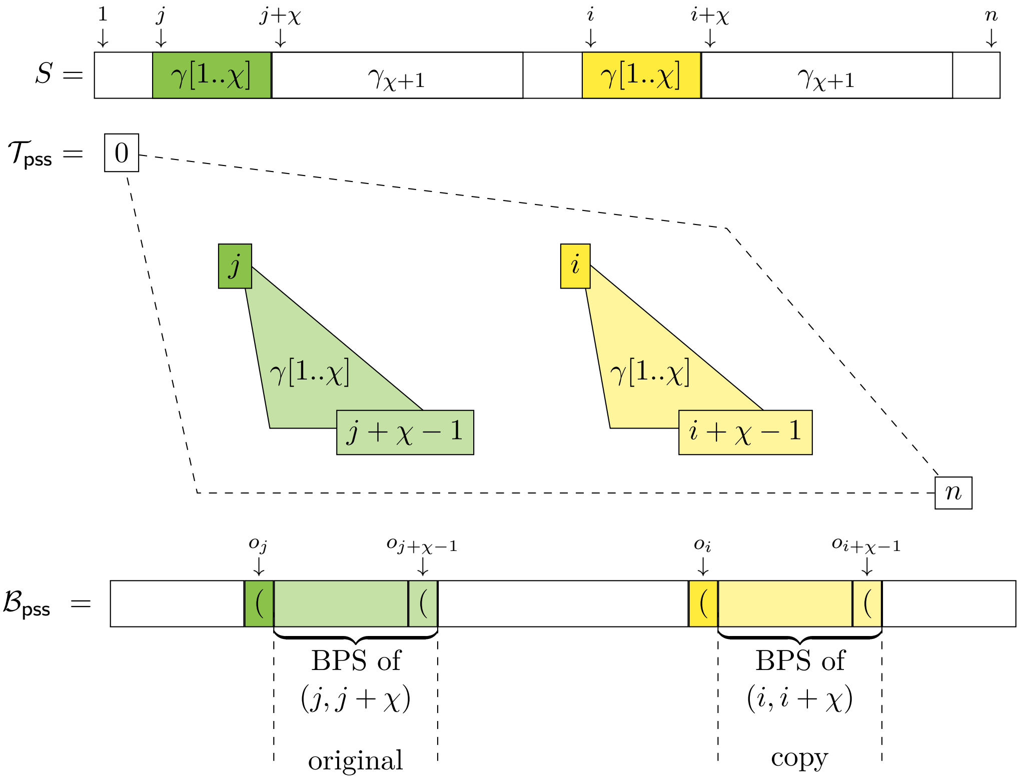
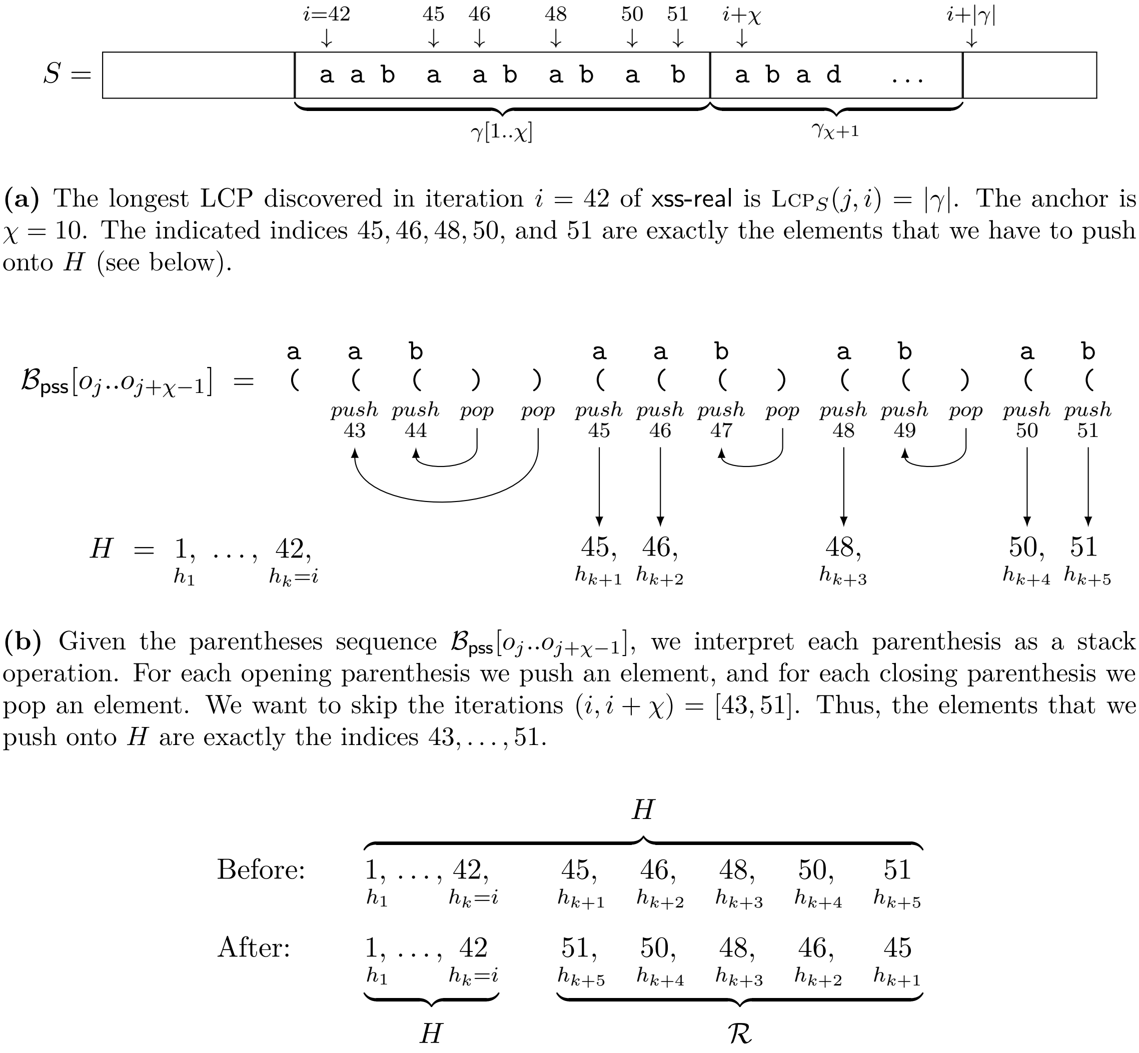


Figure 5.5: The two occurrences of $\gamma[1..\chi]$ induce isomorphic structures in the PSS tree. Therefore, there are also two substrings of the BPS that are identical.

copy of $\mathcal{B}_{\mathsf{pss}}(o_j..o_{j+\chi-1}]$ takes $\mathcal{O}(\chi)$ time. After appending the copy, the last parenthesis that we have written is the opening parenthesis of node $i+\chi-1$. In order to continue the execution of xss-real with iteration $i+\chi$, we still have to update the stacks H and L accordingly.

Updating the Index Stack

In practice, updating H is not a separate step. Instead, we simultaneously append the copy of $\mathcal{B}_{\mathsf{pss}}(o_j..o_{j+\chi-1}]$ and bring H into the correct state. As we have seen in the first version xss-bps of our algorithm (Algorithm 4.1, right), the push operations exactly correspond with the opening parentheses, while the pop operations exactly correspond with the closing parentheses. Therefore, while appending $\mathcal{B}_{\mathsf{pss}}(o_j..o_{j+\chi-1}]$, we can interpret each parenthesis as a stack instruction: Every time we write an opening parenthesis, we push an element onto H, and every time we write a closing parenthesis, we pop an element. The opening parentheses that we write belong exactly to the nodes from the interval $(i, i+\chi)$. Therefore, we always know which index to push next. Let $1 = h_1, \ldots, h_k = i$ be the elements on H before appending the copy of $\mathcal{B}_{\mathsf{pss}}(o_j..o_{j+\chi-1}]$, and let $h_{k+1}, \ldots, h_{k+m} = i+\chi-1$


i=42
45
46
48
50
51
$i+\chi$
$i+|\gamma|$
$S =$
a a b a a b a b a b
a b a d ...
$\gamma[1..\chi]$
$\gamma_{\chi+1}$
$\mathcal{B}_{\mathsf{pss}}[o_j..o_{j+\chi-1}]$ =
a a b a a b a b a b
((( )) ((( ) (( ) ((
push push pop pop push push push pop push push pop push push
43 44 45 46 47 48 49 50 51
H = 1, ..., 42, 45, 46, 48, 50, 51
h_1 h_k=i h_{k+1} h_{k+2} h_{k+3} h_{k+4} h_{k+5}
H
Before: 1, ..., 42, 45, 46, 48, 50, 51
h_1 h_k=i h_{k+1} h_{k+2} h_{k+3} h_{k+4} h_{k+5}
After: 1, ..., 42 51, 50, 48, 46, 45
h_1 h_k=i h_{k+5} h_{k+4} h_{k+3} h_{k+2} h_{k+1}
H
$\mathcal{R}$

(a) The longest LCP discovered in iteration $i = 42$ of xss-real is $\textsc{Lcp}_S(j, i) = |\gamma|$. The anchor is $\chi = 10$. The indicated indices $45, 46, 48, 50$, and 51 are exactly the elements that we have to push onto H (see below).

(b) Given the parentheses sequence $\mathcal{B}_{\mathsf{pss}}[o_j..o_{j+\chi-1}]$, we interpret each parenthesis as a stack operation. For each opening parenthesis we push an element, and for each closing parenthesis we pop an element. We want to skip the iterations $(i, i+\chi) = [43, 51]$. Thus, the elements that we push onto H are exactly the indices $43, \ldots, 51$.

(c) On the stack H, the newly computed indices $h_{k+1}, \ldots, h_{k+m}$ can only be accessed in descending order, i.e. h_{k+m} first and h_{k+1} last. Therefore, we move them from H onto a separate stack $\mathcal{R}$, which reverses their order.

Figure 5.6: Updating the stack H during the amortized look-ahead.

be the additional elements on H afterwards. In Figures 5.6a and 5.6b we provide a short example that demonstrates how to determine $h_{k+1}, \ldots, h_{k+m}$ while appending the copy of $\mathcal{B}_{\mathsf{pss}}(o_j..o_{j+\chi-1}]$.

Note that the additional effort of performing the stack operations does not affect the time bound for appending the copy of $\mathcal{B}_{\mathsf{pss}}[o_j..o_{j+\chi-1}]$. We only perform one stack operation per parenthesis. Therefore, assuming that we have constant time push and pop operations on H, we still need $\mathcal{O}(\chi)$ time to perform the extension.

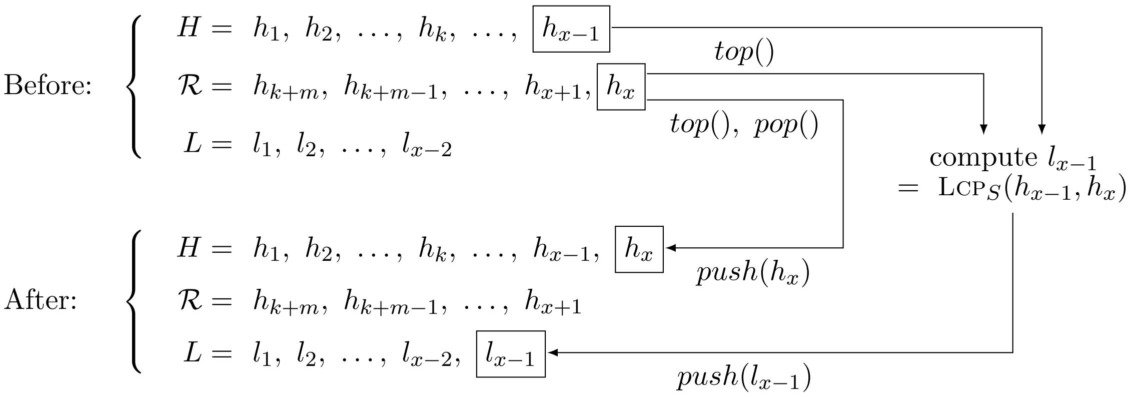

Before:
$H = h_1, h_2, \ldots, h_k, \ldots, \boxed{h_{x-1}}$
$\mathcal{R} = h_{k+m}, h_{k+m-1}, \ldots, h_{x+1}, \boxed{h_x}$
$L = l_1, l_2, \ldots, l_{x-2}$
$top()$
$top(), pop()$
compute l_{x-1}
$= \textsc{Lcp}_S(h_{x-1}, h_x)$
After:
$H = h_1, h_2, \ldots, h_k, \ldots, h_{x-1}, \boxed{h_x}$
$push(h_x)$
$\mathcal{R} = h_{k+m}, h_{k+m-1}, \ldots, h_{x+1}$
$L = l_1, l_2, \ldots, l_{x-2}, \boxed{l_{x-1}}$
$push(l_{x-1})$

Figure 5.7: Computing an LCP value while updating the stacks.

Updating the LCP Stack

It remains to be shown how to update L in $\mathcal{O}(|\gamma|)$ time, i.e. how to efficiently compute the LCP values $l_k, \ldots, l_{k+m-1}$ with $l_x = \textsc{Lcp}_S(h_x, h_{x+1})$ for $x \in [k, k+m)$. We have to push these values onto L in the correct order, i.e. l_k first and l_{k+m-1} last. Therefore, we have to be able to access the indices $h_k, \ldots, h_{k+m}$ in ascending order. This can be achieved by using a separate stack $\mathcal{R}$: We pop the elements $h_{k+1}, \ldots, h_{k+m}$ from the top of H (h_{k+m} first and h_{k+1} last) and push them onto $\mathcal{R}$. This reverses their order, i.e. h_{k+m} lies at the bottom of $\mathcal{R}$, while h_{k+1} is the topmost element. Again, we provide an example in Figure 5.6c.

Now we process the indices $h_{k+1}, \ldots, h_{k+m}$ one at a time and in ascending order. Let h_x be the next index to process. First, we pop h_x from the top of $\mathcal{R}$. Then, we naively compute the LCP length $l_{x-1} = \textsc{Lcp}_S(h_{x-1}, h_x)$, and push h_x and l_{x-1} onto H and L respectively. Figure 5.7 visualizes how the stacks change when processing one index. The described procedure already determines all missing values of L in the right order. However, due to the naive LCP computation, processing index h_x takes $\mathcal{O}(l_{x-1})$ time. This is unfeasible because we want to spend at most $\mathcal{O}(|\gamma|)$ time to process *all* indices. Therefore, after processing h_x we may have to perform one additional step in order to make up for the cost of computing the LCP: If we have $l_{x-1} \geq (h_x - h_{x-1})$, then both suffixes $S_{h_{x-1}}$ and S_{h_x} begin with the prefix $\mu = S[h_{x-1}, h_x)$. In this case we look at the next index h_{x+1}, which is currently the topmost element on $\mathcal{R}$. If the indices h_{x-1}, h_x, and h_{x+1} are equidistant, i.e. if we have $(h_{x+1} - h_x) = (h_x - h_{x-1})$, then it follows that $S_{h_{x-1}} = \mu \cdot S_{h_x}$ and $S_{h_x} = \mu \cdot S_{h_{x+1}}$ hold. Therefore, we have $l_x = \textsc{Lcp}_S(h_x, h_{x+1}) = \textsc{Lcp}_S(h_{x-1}, h_x) - |\mu| = l_{x-1} - |\mu|$. Since both l_{x-1} and $|\mu|$ are known, we can compute l_x in constant time. Thus, we pop h_{x+1} from the top of $\mathcal{R}$, and push h_{x+1} and l_x onto H and L respectively. This step can be repeated until the newly computed LCP value becomes smaller than $(h_x - h_{x-1})$, or until the next index from $\mathcal{R}$ is not equidistant to the two previous indices anymore. For example, when computing $l_{k+2} = \textsc{Lcp}_S(h_{k+2}, h_{k+3}) = \textsc{Lcp}_S(46, 48) = 7$ in the setting of Figure 5.6, the next index $h_{k+3} = 50$ is equidistant to the two previous ones, i.e. we have $(50 - 48) = (48 - 46) = 2$. Therefore, we can compute $l_{k+3} = l_{k+2} - 2 = 7 - 2 = 5$ in constant time. The following index on $\mathcal{R}$ is 51, which is not equidistant to 50 and 48. Thus, we compute the LCP between S_{50} and S_{51} naively.

Once the stack $\mathcal{R}$ is empty, we have computed all missing LCP values $l_k, \ldots, l_{k+m-1}$ and pushed them onto L. Also, we have pushed all indices $h_{k+1}, \ldots, h_{k+m}$ back onto H. Thus, both the BPS and the stacks are ready for iteration $i + \chi$ of $\mathsf{xss\text{-}real}$. It remains to be shown that computing the LCP values in the described manner takes $\mathcal{O}(|\gamma|)$ time.

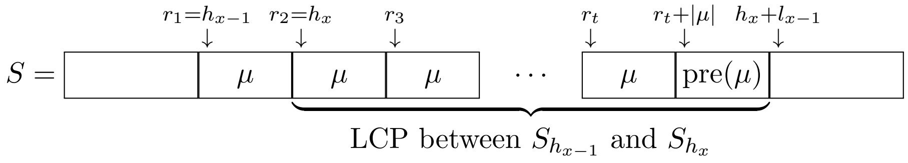
Proving the Time Bound

First, we only consider the LCP values with $l_{x-1} < 2(h_x - h_{x-1})$. We have to perform $l_{x-1} + 1$ individual character comparisons in order to naively compute l_{x-1}. Thus, the total number of character comparisons for all l_{x-1} with $l_{x-1} < 2(h_x - h_{x-1})$ is bound by:

$$\sum_{x=k+1}^{k+m} (l_{x-1} + 1) \;\leq\; \sum_{x=k+1}^{k+m} 2(h_x - h_{x-1}) \;=\; 2(h_{k+m} - h_k)$$

From $h_k = i$ and $h_{k+m} = i + \chi - 1$ follows that the total number of comparisons is at most 2χ. Therefore, we compute all LCP values with $l_{x-1} < 2(h_x - h_{x-1})$ in $\mathcal{O}(\chi)$ time.

Now we consider only the remaining elements l_{x-1} with $l_{x-1} \geq 2(h_x - h_{x-1})$. As described in Section 5.1.1, the string $S[h_{x-1}..h_x + l_{x-1})$ is an increasing Lyndon run of $t \geq 3$ repetitions with period $|\mu| = (h_x - h_{x-1})$ (increasing because of $\mathsf{pss}[h_x] = h_{x-1}$). Let $r_1 = h_{x-1}$, $r_2 = h_x$, and $r_x = r_{x-1} + |\mu|$ for $x \in [3, t]$ be the starting positions of the repetitions.



Each computed LCP value is covered by one of the following three cases.

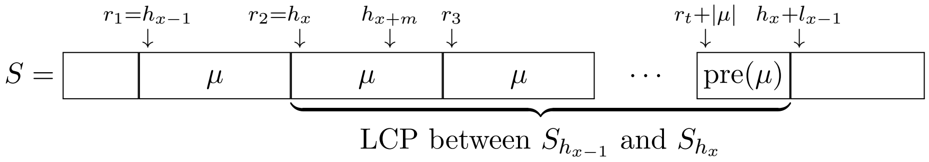
Case (1): The next index h_{x+1} lies after r_3, i.e. $h_{x+1} > r_3$. From Lemma 5.1.10 follows that $h_{x+1} > r_t + |\mu| = h_{x-1} + (t + 1) \cdot |\mu|$ holds. Since we also have $l_{x-1} < t \cdot |\mu|$, the LCP is bound by $l_{x-1} < (h_{x+1} - h_{x-1})$. Thus, the total number of character comparisons needed for this case is $\mathcal{O}(\chi)$:

$$\sum_{x=k+1}^{k+m-1} (l_{x-1} + 1) \leq \sum_{x=k+1}^{k+m-1} (h_{x+1} - h_{x-1}) < 2(h_{k+m} - h_k) = \mathcal{O}(\chi)$$

Case (2): The next index h_{x+1} is exactly r_3. Since h_{x-1}, h_x, and h_{x+1} are equidistant, we compute the next LCP length $l_x = l_{x-1} - |\mu|$ in constant time. The number of character comparisons needed to compute l_{x-1} is exactly $l_{x-1} + 1$. We interpret $l_x + 1$ of these comparisons as the cost of computing l_x. Since l_x is covered by one of the three cases as well (or we have $l_x < 2(h_{x+1} - h_x)$), we only have to account for the remaining $l_{x-1} - l_x = |\mu| = (h_x - h_{x-1})$ character comparisons. Again, we obtain the upper bound $\mathcal{O}(\chi)$ for all LCP values of this case:

$$\sum_{x=k+1}^{k+m} (l_{x-1} - l_x) = \sum_{x=k+1}^{k+m} (h_x - h_{x-1}) = (h_{k+m} - h_k) = \mathcal{O}(\chi)$$

Case (3): The only remaining scenario is $h_{x+1} < r_3$. Lemma 5.1.9 implies that the topmost element on H is smaller than r_3. It follows, that all remaining larger indices $h_{x+2}, \ldots, h_{x+m}$ are from the interval $(h_{x+1}, r_3) = (r_2, r_3)$.



Assume that we are currently looking at the leftmost l_{x-1} that is covered by this case. Then the next LCP value l_{y-1} that is covered by this case (if it exists) is the length of the LCP between h_{y-1} and h_y, where both of these indices are from the interval $[h_x, r_3)$. We have already seen that two indices in the same repetition of a Lyndon run share a prefix of length less than $|\mu|$ (for example in Lemma 5.1.3, $\mu_a \cdot \mu$ and $\mu_b \cdot \mu$ have a guaranteed mismatch within the first $|\mu_a|$ characters). Thus, we also have $l_{y-1} < |\mu|$. Since at the same time we have $l_{x-1} \geq 2|\mu|$, we know that $l_{y-1} < l_{x-1}/2$ holds. If we iterate over all LCP values that are covered by this case, then every value is less than half as large as the previous one. It follows, that the sum of all LCP values of this case is smaller than $2 \cdot l_{x-1}$, if l_{x-1} is the leftmost and thus largest of the covered values. Lastly, we also know that the largest LCP that we compute during the amortized look-ahead is bound by $|\gamma|$ because otherwise we contradict Lemma 5.2.3. Therefore, the sum of all LCP values covered by this case is bound by $2|\gamma|$, and computing the values takes $\mathcal{O}(|\gamma|)$ time.

5.3 Algorithmic Summary

We have seen how to directly compute the BPS of the PSS tree in linear time. The new algorithm can be summarized as follows:

- A commonly used stack algorithm for the computation of NSVs and PSVs is the base of our algorithm. We use a loop to process all indices from the interval $(1, n)$ in ascending order, i.e. the loop runs for $n - 2$ iterations. Instead of performing element comparisons, which are used in the NSV/PSV setting, we perform naive suffix comparisons (xss-array). Also, instead of saving the PSS and NSS array, we directly build the BPS of the PSS tree in an append-only manner (xss-bps).
- At all times, the stack H keeps track of the indices that have already been processed, but whose next smaller suffix has not been found yet.
- We use an additional stack L to memorize the LCP values of adjacent indices on H. By using the stored information we can skip character comparisons while performing suffix comparisons (xss-bps-lcp).
- If an iteration causes a critical time overhead of $\mathcal{O}(|\gamma|)$, then we can fast-forward the next $\Omega(|\gamma|)$ iterations by using either the run extension or the amortized look-ahead (xss-real). Since the time needed for this process is linear in the number of iterations that we skip, on average each iteration takes constant time.
  - If we detect a Lyndon run in the input string, then each repetition of the run (except for the last one) induces the same structure in the PSS tree.
  - Otherwise, we can still identify two isomorphic structures in the PSS tree. If these structures are small, then we can detect an additional Lyndon run.
  - The run extension and the amortized look-ahead exploit the isomorphic structures by copying a part of the already computed BPS prefix and appending it to its end. The stacks H and L get updated to reflect these changes.

The correctness of xss-real follows directly from the description. Effectively, the algorithm performs the same operations as xss-bps-lcp, but takes some shortcuts when fast-forwarding through iterations. The worst-case time bound is linear because on average each iteration takes constant time.

5.3.1 Lemma. *Let S be a guarded string of length n. The algorithm xss-real computes the BPS of the PSS tree of S in $\mathcal{O}(n)$ time using $\mathcal{O}(n)$ words of memory apart from input and output.*

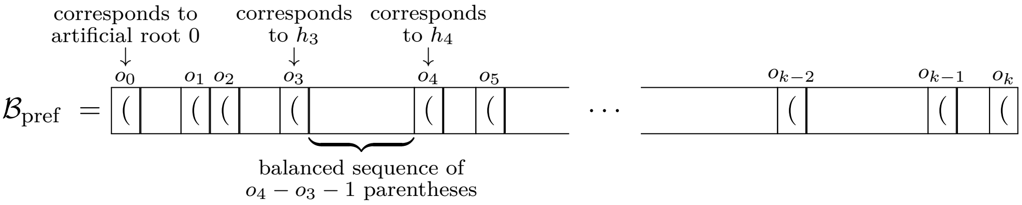
Chapter 6

Decreasing the Memory Bound

In order to obtain a more space aware solution, we do not need to change the algorithm. Instead, we simply show how to maintain succinct representations of the stacks H and L. We can show, that $\mathcal{O}(\sqrt{n} \cdot \lg n)$ bits are enough to simulate H without affecting the worst-case time bound of our algorithm. Also, we give a more general alternative representation that can be used for any stack of strictly increasing integers. While it uses $n + \mathcal{O}(\lg n \cdot \lg \lg n)$ bits of memory, it has the advantage of high cache efficiency. By using this representation for the stack $\mathcal{R}$ during the amortized look-ahead, we can fit all elements of $\mathcal{R}$ into the free space at the back of the BPS, such that $\mathcal{R}$ effectively needs no additional memory. For L we introduce a parameterized representation that uses $4n/\delta + \mathcal{O}(\lg n \cdot \lg \lg n)$ bits of memory and allows our algorithm to run in $\mathcal{O}(\delta n)$ time.

6.1 Maintaining H in $\mathcal{O}(\sqrt{n} \cdot \lg n)$ Bits

Consider the stack $H = h_1, \ldots, h_k$ at any point in time during the execution of our algorithm. At this point we have written a prefix $\mathcal{B}_{\text{pref}}$ of the BPS, where for some of the opening parentheses we have already written the matching closing parentheses, and all other opening parentheses are still *unmatched*. Since for every push operation on the stack we write an opening parenthesis and for every pop operation we write a closing parenthesis, the elements on the stack directly correspond to the unmatched opening parentheses of the known BPS prefix. More precisely, iff h_i is an element on H, then the $(h_i + 1)$-th opening parenthesis of the BPS does not have a matching closing parenthesis yet (remember that node h_i has preorder-number $h_i + 1$, see Lemma 3.1.2). For $i \in [1, k]$ let o_i be the index of the $(h_i + 1)$-th opening parenthesis, and let $o_0 = 1$ be the index of the first opening parenthesis, which belongs to the artificial root node 0. Below we see the known BPS prefix, where each empty block between two indices o_i and o_{i+1} represents a balanced parentheses string of length $(o_{i+1} - o_i - 1) = 2(h_{i+1} - h_i - 1)$:


corresponds to artificial root 0
corresponds to h_3
corresponds to h_4
o_0
o_1
o_2
o_3
o_4
o_5
o_{k-2}
o_{k-1}
o_k
$\mathcal{B}_{\text{pref}}$ =
(
(
(
(
(
(
...
(
(
(
balanced sequence of $o_4 - o_3 - 1$ parentheses

Now assume that we know the pair (o_{i+1}, h_{i+1}), then we can find the pair (o_i, h_i) using only the known prefix of the BPS. We scan the BPS from right to left, starting at index $o_{i+1} - 1$. As soon as we have read more opening than closing parentheses, we have found the next unmatched opening parenthesis at index o_i. Then we have:

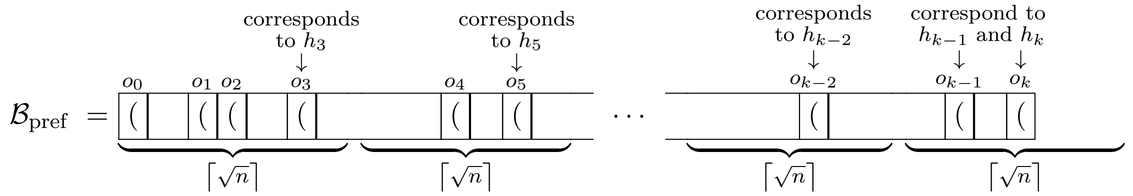
$$h_i = h_{i+1} - \underbrace{(o_{i+1} - o_i - 1)/2}_{\substack{\text{opening parenthesis} \\ \text{between } o_i \text{ and } o_{i+1}}} - 1$$

Theoretically, we can already simulate top access to H, if we always keep the rightmost pair (o_k, h_k) in memory. Whenever we write a closing parenthesis, we then need to restore the next pair (o_{k-1}, h_{k-1}) using the described scanning technique. This however would increase the time bound of our algorithm, since we might need to scan large parts of the BPS multiple times. These repeated scans can be prevented by using a small support data structure that was introduced by Davoodi et al. for the construction of Cartesian trees [Davoodi et al., 2017, Chapter 3]. Conceptually, we divide the known prefix of the BPS in blocks of length $\lceil\sqrt{n}\rceil$, where the first block starts at BPS index 1. For each block that is entirely contained in the known prefix of the BPS and that contains at least one unmatched opening parenthesis, we store the pair (o, h) on a stack $\mathcal{P}_{\text{all}}$, where:

- o is the BPS index of the rightmost so far unmatched opening parenthesis of the block. This parenthesis has to be unmatched *in the entire known prefix of the BPS*, i.e. an opening parenthesis can be matched even though its closing parenthesis does not lie within the same block.

- h is the text index that corresponds to the rightmost so far unmatched opening parenthesis of the block (i.e. the $(h+1)$-th opening parenthesis of the BPS is located at index o).

There are at most $\mathcal{O}(\sqrt{n})$ blocks and each pair (o, h) uses $\mathcal{O}(\lg n)$ bits. Thus, all pairs together need $\mathcal{O}(\sqrt{n} \cdot \lg n)$ bits of memory. Since in general the length of $\mathcal{B}_{\text{pref}}$ is not necessarily a multiple of $\lceil\sqrt{n}\rceil$, we have a partial block at the end. For this block we maintain a separate stack $\mathcal{P}_{\text{last}}$ that stores both the BPS index and the corresponding text index of *all* unmatched opening parentheses, which takes another $\mathcal{O}(\sqrt{n} \cdot \lg n)$ bits of memory. Below we see a visualization of the data structures at hand:



$$\mathcal{P}_{\text{all}} \;=\; (o_3, h_3), \quad (o_5, h_5), \quad \ldots, \quad (o_{k-2}, h_{k-2})$$

$$\mathcal{P}_{\text{last}} \;=\; (o_{k-1}, h_{k-1}),\ (o_k, h_k)$$

Now we describe how to perform push, pop and top operations on H using only the known prefix of the BPS as well as the stacks $\mathcal{P}_{\text{all}}$ and $\mathcal{P}_{\text{last}}$.

***top*()**: If $\mathcal{P}_{\text{last}}$ is empty, return the h-component of the topmost element on $\mathcal{P}_{\text{all}}$. Otherwise return the h-component of the topmost element on $\mathcal{P}_{\text{last}}$. Clearly this takes constant time.

***push*(*e*)**: Let $h_{k+1} = e$ and let $o_{k+1} = |\mathcal{B}_{\text{pref}}|$. Since we are pushing an element, we have just appended an opening parenthesis to the BPS. Therefore the h_{k+1}-th opening parenthesis is located at index o_{k+1} of the BPS. Let o_{top} be the o-component of the topmost element on $\mathcal{P}_{\text{last}}$. If $\lfloor o_{\text{top}}/\lceil\sqrt{n}\rceil\rfloor = \lfloor o_{k+1}/\lceil\sqrt{n}\rceil\rfloor$ holds (or if $\mathcal{P}_{\text{last}}$ is empty), then the rightmost block is still unfinished and we push (o_{k+1}, h_{k+1}) onto $\mathcal{P}_{\text{last}}$. Otherwise, we have finished the rightmost block. In this case we push the topmost element from $\mathcal{P}_{\text{last}}$ (which is (o_k, h_k)) onto $\mathcal{P}_{\text{all}}$, and then make (o_{k+1}, h_{k+1}) the only element on $\mathcal{P}_{\text{last}}$. This also takes constant time.

***pop*()**: If $\mathcal{P}_{\text{last}}$ is not empty, then we simply pop the topmost element on $\mathcal{P}_{\text{last}}$. Otherwise, let (o_k, h_k) be the topmost element on $\mathcal{P}_{\text{all}}$. We can restore (o_{k-1}, h_{k-1}) by using the scanning technique as described earlier. If we reach the beginning of the block without finding an unmatched parenthesis, then we pop (o_k, h_k) off the top of $\mathcal{P}_{\text{all}}$. Otherwise, we update the topmost element on $\mathcal{P}_{\text{all}}$ to be (o_{k-1}, h_{k-1}). Apart from the actual scanning, the pop operation takes constant time. Since the scanning time is linear in the length of the scanned area, and since we only scan each block once, the total scanning time during the execution of our algorithm is limited by $\mathcal{O}(n)$.

Note that the operations only work as described, if we only push elements immediately after writing the respective opening parentheses. However, even when performing the run extension or the amortized look-ahead of xss-real, we can always write the BPS and update the stack H simultaneously. Therefore, the operations work as described for all of our algorithms. We have shown:

6.1.1 Lemma. *We can simulate the stack H using only the already computed prefix of the BPS as well as a support data structure of size $\mathcal{O}(\sqrt{n} \cdot \lg n)$ bits without affecting the worst-case time bound of our algorithm.*

6.2 Maintaining H in $n + \mathcal{O}(\lg n \cdot \lg\lg n)$ Bits

While using the support data structure from the previous section results in a small memory footprint, it may perform poorly in terms of cache efficiency. To simulate H we have to access essentially arbitrary positions of the BPS during the entire execution of the algorithm. In this section, we propose an alternative representation of H that is very cache efficient. As we will see, the reason for this efficiency is, that elements that are close to each other on the logical stack are also stored close to each other in terms of physical memory layout. However, it comes at the cost of a higher memory bound of up to $n + \mathcal{O}(\lg n \cdot \lg\lg n)$ bits. A similar stack has been introduced by Fischer in the context of range minimum queries [Fischer, 2010, Section 4.2]. Our new approach has multiple advantages. Most importantly, it is straightforward to implement, allows dynamic memory allocation based on the actual number of elements on the stack, and the $\mathcal{O}(\lg n \cdot \lg\lg n)$ bit term can be dropped on modern CPUs. The core of the new representation of H is a stack that stores small values in unary representation, and large values in binary representation (we will clarify the exact meaning of small and large later). Since the stack achieves its memory bound thanks to the unary representation, we call it *unary stack*.

6.2.1 Counting Trailing Zeros

An important prerequisite of the unary stack is the ability to count trailing zeros in constant time. Let w be the size of a computer word in bits. Given a bit string x of length $b = \mathcal{O}(w)$ (i.e. $b \in [0, 2^b)$), we want to find the length $\mathtt{TZ}(x)$ of the longest zero-only suffix of x. For example, $\mathtt{TZ}((11001000)_2) = 3$, because the last three bits are zero. Note that on modern Intel and AMD processors these queries can easily be answered using the `TZCNT` instruction from the Bit Manipulation Instruction Set 1 (BMI1)[1], which counts the trailing zeros of an entire computer word in constant time. In this section, we show two small index data structures that can answer $\mathtt{TZ}(x)$ in constant time, even if the `TZCNT` instruction is not available. In any case, we preprocess x by extracting the rightmost $(1)_2$-bit as follows: If $x = 0$, then we have $\mathtt{TZ}(x) = b$. Otherwise x has the form $(\alpha 1 0^{\mathtt{TZ}(x)})_2$ for some bit string α. Let $-x$ be the two's complement of x, i.e. $-x = \bar{x} + (1)_2$. We define

[1] https://software.intel.com/file/36945

the extraction function $f(x)$: $[1, 2^b) \to \{2^z \mid z \in [0, b)\}$ with $f(x) = x\&{-x}$. It is easy to show that we have $f(x) = 2^{\mathtt{TZ}(x)}$:

$$\begin{aligned} f(x) &= x \quad \& \quad -x \\ &= (\alpha 10^{\mathtt{TZ}(x)})_2 \;\&\; ((\bar{\alpha} 01^{\mathtt{TZ}(x)})_2 + (1)_2) \\ &= (\alpha 10^{\mathtt{TZ}(x)})_2 \;\&\; (\bar{\alpha} 10^{\mathtt{TZ}(x)})_2 \\ &= (10^{\mathtt{TZ}(x)})_2 \\ &= 2^{\mathtt{TZ}(x)} \end{aligned}$$

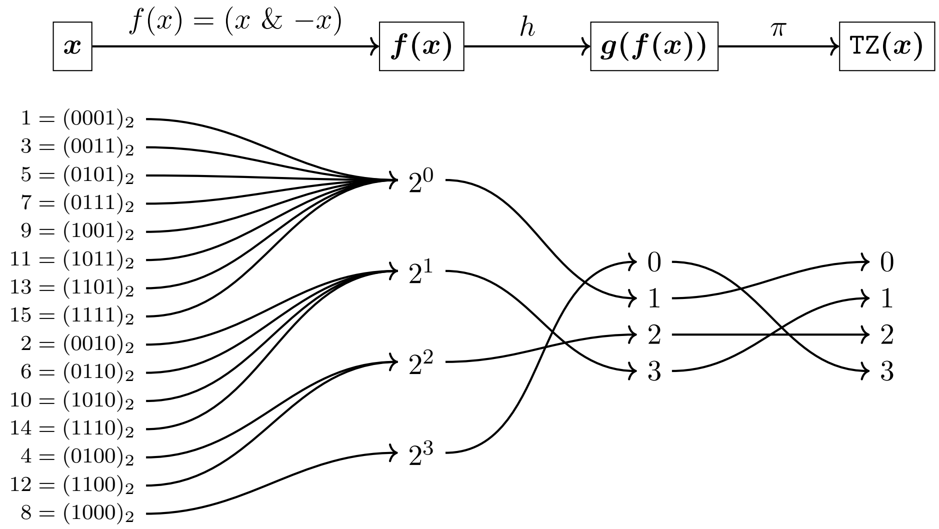
Computing $f(x)$ clearly takes constant time. If we use this simple preprocessing, then our index data structure only has to be able to answer trailing zero queries for elements from the set $\{2^z \mid z \in [0, b)\}$ (instead of $[0, 2^b)$).

Using De Bruijn Sequences

A common technique uses minimal perfect hashing to map $\{2^z \mid z \in [0, b)\}$ onto the compact interval $[0, b)$. Since 2^z is likely not going to be mapped onto z, we have to permute the results back into the correct order, which can be realized by using a lookup table. For computer words of size 64 bits (i.e. $b = 64$), this approach can be found in [Knuth, 2011, p. 142]. We show a generalization to bit strings of arbitrary length b, which is derived from [Leiserson et al., 1970]. Without loss of generality we assume $(\lg b) \in \mathbb{N}$.

Let $x \in [1, 2^b)$ be the bit string for which we want to count the number of trailing zeros, and let $f(x) = 2^{\mathtt{TZ}(x)} \in \{2^z \mid z \in [0, b)\}$ be the rightmost $(1)_2$-bit of x, which we have already extracted as described earlier. We compute $\mathtt{TZ}(x)$ by applying two functions h and π with $\pi(h(f(x))) = \mathtt{TZ}(x)$.

- The bijective function h: $\{2^z \mid z \in [0, b)\} \to [0, b)$ is a minimal perfect hash function that maps the range of f onto the compact interval $[0, b)$. Finding a suitable function h requires a one time precomputation that takes $\mathcal{O}(b)$ time and $\mathcal{O}(b \lg b)$ bits of memory. After this precomputation, the function can be stored in b bits and evaluated in constant time.
- The bijective function π: $[0, b) \to [0, b)$ brings the range of h back into the correct order, i.e. $\pi(h(2^z)) = z$. Since effectively this function is a permutation of $[0, b)$, it can be stored in a lookup table of size $b \lg b$ bits. Filling the table takes $\mathcal{O}(b)$ time and no additional memory.



De Bruijn sequence for h:

$$h(f(x)) = ((a \cdot f(x)) \bmod 16) \gg 2$$
$$a = (a_1a_2a_3a_4)_2 = (0110)_2 = 6$$

$f(x)$	$h(f(x))$
$2^0 = (0001)_2$	$(a_1a_2)_2 = (01)_2 = 1$
$2^1 = (0010)_2$	$(a_2a_3)_2 = (11)_2 = 3$
$2^2 = (0100)_2$	$(a_3a_4)_2 = (10)_2 = 2$
$2^3 = (1000)_2$	$(a_40)_2 = (00)_2 = 0$

Lookup table for π:

$$\pi(h(f(x))) = Z_\pi[h(f(x))] = \texttt{TZ}(x)$$

$h(f(x))$	$Z_\pi[h(f(x))]$
$0 = h(2^3)$	3
$1 = h(2^0)$	0
$2 = h(2^2)$	2
$3 = h(2^1)$	1

Figure 6.1: Interaction of the functions f, h and π in the de Bruijn method for bit strings of length four. The utilized de Bruijn sequence is $a = (0110)_2$. Each entry of the lookup table Z_π can be stored using two bits.

First, we show how to find a suitable function h. Let $a \in [0, 2^b)$, i.e. a is a bit string $(a_1a_2 \dots a_b)_2$ of length b. Consider the following function:

$$h(2^z) = ((a \cdot 2^z) \ \texttt{mod}\ 2^b) \gg (b - \lg b)$$

Clearly, this function can be computed in constant time, and we only need to store a, which takes b bits. Effectively, the function extracts a consecutive interval of bits from a. If we define $a_i = 0$ for $i > b$, then the result of the function can be rewritten as follows:

$$\begin{aligned} h(2^z) &= ((a \cdot 2^z) \ \texttt{mod}\ 2^b) \gg (b - \lg b) \\ &= ((a \ll z) \ \texttt{mod}\ 2^b) \gg (b - \lg b) \\ &= (a_{z+1}a_{z+2} \dots a_{z+\lg b})_2 \end{aligned}$$

We have to select a in a way such that h becomes bijective, i.e. $(a_{z_1+1} \dots a_{z_1+\lg b})_2 \neq (a_{z_2+1} \dots a_{z_2+\lg b})_2$ for all $z_1, z_2 \in [0, b)$ with $z_1 \neq z_2$. A *binary de Bruijn sequence of order* $(\lg b)$ is a cyclic bit string of length b that contains each possible bit string of length $(\lg b)$ exactly once [de Bruijn, 1946]. For example, $(00010111)_2$ is a de Bruijn sequence of order three, because it contains all bit strings of length three exactly once. In this case, $(110)_2$ and $(100)_2$ are contained because we consider $(00010111)_2$ in a cyclic manner. For our hash function h we select a such that it is a binary de Bruijn sequence of order $(\lg b)$ and begins with the prefix $(0^{\lg b - 1})_2$. It has been proven over a century ago that many of such de Bruijn sequences exist for each b [Flye Sainte-Marie, 1894]. Finding a binary de Bruijn sequence for a fixed b takes $\mathcal{O}(b)$ time and $\mathcal{O}(b \lg b)$ bits of memory (for example by using Hierholzer's algorithm [Hierholzer and Wiener, 1873] to find a Eulerian circuit in the $(\lg b - 1)$-dimensional binary de Bruijn graph in which every node has in- and out-degree two).

The permutation $\pi(h(2^z)) = z$ is naturally defined by h. We simply create a lookup table Z_π that has b entries $Z_\pi[0], \dots, Z_\pi[b-1]$ with $Z_\pi[i] = \pi(i)$. The table can be computed in $\mathcal{O}(b)$ time by taking each $z \in [0, b)$ and setting $Z_\pi[h(2^z)] = z$. The final formula for computing $\texttt{TZ}(x)$ with $x \neq 0$ is:

$$\texttt{TZ}(x) = Z_\pi[\underbrace{((a \cdot \overbrace{(x \mathbin{\&} -x)}^{f(x)}) \bmod 2^b) \gg (b - \lg b)}_{h(f(x))}]$$

Figure 6.1 visualizes how the functions f, h and π interact. Answering $\texttt{TZ}(x)$ takes constant time, because only a constant amount of basic word RAM operations is required. Therefore we have shown:

6.2.1 Lemma. *Let $b \in [1, w]$. There is a data structure of size $\mathcal{O}(b \lg b)$ bits that can answer $\texttt{TZ}(x)$ in constant time for any $x \in [0, 2^b)$. It can be constructed in $\mathcal{O}(b)$ time using $\mathcal{O}(b \lg b)$ bits of memory.*

In practice, the most expensive operation during the computation of $\texttt{TZ}(x) = h(g(f(x)))$ is the integer multiplication $a \cdot 2^z$ in the function g. Apart from that, only primitive CPU instructions like bit-shifts and additions are used. The lookup in Z_π is also very fast because even for $b = 64$ the entire table uses only 64 bytes, which fits into a single cache line on most Intel and AMD processors.

Using Binary Search

An alternative to the de Bruijn method of the previous section is a simple binary search. Once again we assume $(\lg b \in \mathbb{N})$, and we manually check for the special case $x = 0$. For

$x > 0$ the number of trailing zeros can be described recursively as $\texttt{TZ}(x) = \mathit{search}_b(x)$ with:

$$\mathit{search}_b(x) = \begin{cases} \mathit{search}_{b/2}(x \gg b/2) + b/2 & \text{, iff } (x \texttt{ mod } 2^{b/2} = 0) \ \wedge \ (b > 1) \\ \mathit{search}_{b/2}(x) & \text{, iff } (x \texttt{ mod } 2^{b/2} > 0) \ \wedge \ (b > 1) \\ 0 & \text{, otherwise (i.e. } b = 1) \end{cases}$$

If the lower (less significant) $b/2$ bits of x do not contain a $(1)_2$-bit (i.e. if $x \texttt{ mod } 2^{b/2} = 0$), then the number of trailing zeros is $b/2$ plus the number of trailing zeros in the bit string $x \gg b/2$, which is of length $b/2$ (see first case of the definition above). If however there is a $(1)_2$-bit in the lower half of x (i.e. if $x \texttt{ mod } 2^{b/2} > 0$), then the number of trailing zeros of x is the same as the number of trailing zeros of the lower half of x (see second case of the definition above). For bit strings of length one, we have no trailing zeros because we already manually checked for $x = 0$ in the beginning (see third case of the definition above). Since in every step we cut the number of bits in half, the total number of recursive steps is $(\lg b)$. Therefore, $\mathit{search}_b(x)$ is already a suitable algorithm for answering trailing zero queries in $\mathcal{O}(\lg b)$ time.

Only a small change to the definition of $\mathit{search}_b(x)$ is necessary to achieve constant query time instead. We allow a larger base case for the recursive call to search, i.e. once we have narrowed down the position of the rightmost $(1)_2$-bit to an interval of length c, we use a lookup table Z_c to count the trailing zeros of those c bits in constant time. Once again we assume $(\lg c) \in \mathbb{N}$ without loss of generality.

$$\mathit{search}_b(x) = \begin{cases} \mathit{search}_{b/2}(x \gg b/2) + b/2 & \text{, iff } (x \texttt{ mod } 2^{b/2} = 0) \ \wedge \ (b > c) \\ \mathit{search}_{b/2}(x) & \text{, iff } (x \texttt{ mod } 2^{b/2} > 0) \ \wedge \ (b > c) \\ Z_c[f(x)] & \text{, otherwise (i.e. } b \leq c) \end{cases}$$

The lookup table Z_c has $1 + 2^{c-1}$ entries of size $\lg c$ bits, such that $Z_c[f(x)] = \texttt{TZ}(x)$ (note that at this point x has been shifted far enough to the right, that the rightmost $(1)_2$-bit is located in the lowest c bits). Since we use the function f from the previous section to extract the rightmost $(1)_2$-bit, we only need to assign the lookup table entries $Z_c[2^z] = z$ for $z \in [0, c)$, which takes $\mathcal{O}(c)$ time. We need $(\lg(b/c))$ recursive steps of *search* to reach the base case.

6.2.2 Lemma. *Let $b \in [1, w]$ and $c \in [1, b]$. There is a data structure of size $\mathcal{O}(2^c \cdot \lg c)$ bits that can answer $\texttt{TZ}(x)$ in $\mathcal{O}(1 + \lg(b/c))$ time for any $x \in [0, 2^b)$. It can be constructed in $\mathcal{O}(c)$ time using no additional memory.*

For instance, we can select $c = b/2$, which allows constant time trailing zero queries, using a lookup table of size $\mathcal{O}(\sqrt{2^b} \cdot \lg b)$ bits. Figure 6.2 shows an example for $b = 32$ and

$$\begin{aligned}
&\texttt{TZ}((\,0110\;1010\;\;1110\;\;0000\;\;0000\;0000\;0000\;0000\,)_2)\\
&= \mathit{search}_{32}((\,0110\;1010\;\;1110\;\;0000\;\;\underbrace{0000\;0000\;0000\;0000}_{=\,0}\,)_2)\\
&= \mathit{search}_{16}((\,0110\;1010\;\;\underbrace{1110\;\;0000}_{>\,0}\,)_2) \quad + \; 16\\
&= \mathit{search}_{8}((\,1110\;\;\underbrace{0000}_{=\,0}\,)_2) \quad + \; 16\\
&= \mathit{search}_{4}((\,1110\,)_2) \; + \; 4 \; + \; 16\\
&= Z_4[f((\,1110\,)_2)] \; + \; 4 \; + \; 16\\
&= Z_4[(\,0010\,)_2] \; + \; 4 \; + \; 16\\
&= 1 \; + \; 4 \; + \; 16 \; = \; 21
\end{aligned}$$

i	$Z_4[i]$
0	–
$1 = (0001)_2$	0
$2 = (0010)_2$	1
3	–
$4 = (0100)_2$	2
5	–
6	–
7	–
$8 = (1000)_2$	3

Figure 6.2: Counting trailing zeros in constant time using binary search and lookup. We take a bit string of length $b = 32$ bits as an example and use a lookup table for bit strings of length $c = 4$ bits. Therefore, we need $\lg(32/4) = \lg 8 = 3$ recursive steps to reach the base case.

$c = 4$. In Section 7.2 we evaluate the practical query time and memory usage of the two strategies from Lemmas 6.2.1 and 6.2.2. We also try different values for c, and compare the query time of both approaches against the `TZCNT` instruction.

6.2.2 Succinct Unary Stack

With the trailing zero queries of the previous section, we can realize the unary stack. Note that this stack is not used as the final representation of the index stack H of our algorithms. Instead, we first introduce the unary stack, and then use it as an underlying data structure for the actual representation of H.

6.2.3 Lemma. *There is a stack with constant time push, pop, and top operations that stores up to n elements from the interval $[1, n]$ using $n + \mathcal{O}(\lg n \cdot \lg\lg n)$ bits of memory, if the sum of the elements on the stack never exceeds n. It can be initialized in $\mathcal{O}(\lg n)$ time.*

Proof. We use a bit vector $\mathcal{V}$ of length n bits to store the stack elements, which we classify into *small* and *large* elements. The front of the bit vector contains the small elements $e < 2\lceil \lg n \rceil$ in unary representation, while the back of the bit vector contains all the large elements $e \geq 2\lceil \lg n \rceil$ in fixed width binary representation. When pushing small elements, data in the front of the bit vector grows from left to right, and when pushing large elements, data in the back of the bit vector grows from right to left. During the entire lifetime of the stack we maintain variables containing the index t_{s} of the rightmost bit used by a small element, as well as the index t_ℓ of the leftmost bit used by a large element. Since at any

time we have to know if the topmost element is a small element or a large element, we annotate each large element with the value of t_{s} at the time of pushing the large element. For both the large element and its annotation we use $\lceil \lg n \rceil$ bits each to store the binary representation with leading zeros, such that each large element including the annotation uses exactly $2\lceil \lg n \rceil$ bits of the bit vector.
Initially, all bits of $\mathcal{V}$ are set to $(0)_2$, index $t_{\mathrm{s}} \leftarrow 0$ points to the left of the first bit, and index $t_\ell \leftarrow n+1$ points to the right of the last bit. The operations are realized as follows:

***push*(*e*):** Depending on the size of the pushed element e we proceed as follows:

1. For small elements $e < 2\lceil \lg n \rceil$ we set $\mathcal{V}[t_{\mathrm{s}}+e]$ to $(1)_2$ and update the index $t_{\mathrm{s}} \leftarrow t_{\mathrm{s}}+e$ accordingly. The number of used bits increases exactly by e.

2. For large elements $e \geq 2\lceil \lg n \rceil$ we take the $\lceil \lg n \rceil$ bit binary representations of t_{s} and e (with leading zeros) and write them to the bit vector intervals $\mathcal{V}[t_\ell - 2\lceil \lg n \rceil..t_\ell - \lceil \lg n \rceil)$ and $\mathcal{V}[t_\ell - \lceil \lg n \rceil..t_\ell)$ respectively. We update the index $t_\ell \leftarrow t_\ell - 2\lceil \lg n \rceil$ accordingly. The number of used bits increases by $2\lceil \lg n \rceil \leq e$.

***top*():** When retrieving the topmost element e, we first need to find out if it is a small or a large element. Let t_{c} be the unsigned integer stored in binary representation in interval $\mathcal{V}[t_\ell..t_\ell + \lceil \lg n \rceil)$, i.e. t_{c} is the annotation of the topmost large element.

1. If $t_{\mathrm{c}} = t_{\mathrm{s}}$, then the topmost element is a large element, because no small element has been pushed after the considered large element. Its value is stored in binary representation in the interval $\mathcal{V}[t_\ell + \lceil \lg n \rceil..t_\ell + 2\lceil \lg n \rceil)$.

2. Otherwise, the topmost element is a small element. Its value is one larger than the number of trailing zeros of the bit string $\mathcal{V}[t_{\mathrm{s}} - \lceil 2 \lg n \rceil..t_{\mathrm{s}})$. We can determine it in constant time by using the de Bruijn method from Section 6.2.1 with $b = 2\lg n$ (or by using the `TZCNT` instruction if available).

***pop*():** We already showed how to retrieve the top element. Knowing its value, we can simply revert the corresponding push operation. Let e be the top element.

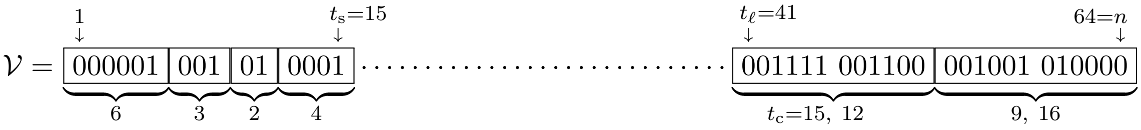
1. If $e < 2\lceil \lg n \rceil$ we set bit $\mathcal{V}[t_{\mathrm{s}}]$ to zero and decrease t_{s} by e.

2. Otherwise we overwrite $\mathcal{V}[t_\ell..t_\ell + 2\lceil \lg n \rceil)$ with zeros and increase t_ℓ by $2\lceil \lg n \rceil$.

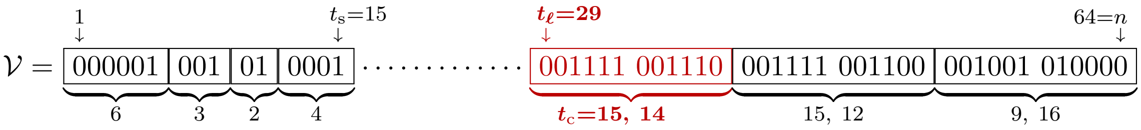
In the word RAM model we can write the $\lceil \lg n \rceil$ bit binary representations of elements and annotations to any position in the bit vector in constant time using only logical `AND`, `OR`, and bitshift operations. The same is true for retrieving those representations and for overwriting them with zeros. Therefore all described stack operations take constant time.

Since pushing an element e increases the number of used bits by at most e, and since popping an element reverts its push operation, the number of used bits never exceeds the sum of all elements on the stack. Using the de Bruijn method with $b = 2\lg n$ requires a lookup table of size $\mathcal{O}(\lg n \cdot \lg\lg n)$ bits and a one-time initialization time of $\mathcal{O}(\lg n)$ (see Lemma 6.2.1). □

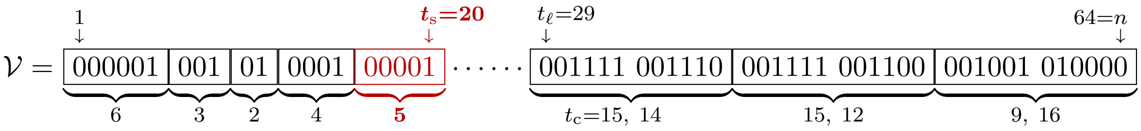
Example. We provide a detailed example of the unary stack, starting with the non-empty unary stack $\mathcal{U} = 6, 3, 16, 2, 4, 12$ and then performing some push and pop operations. The initial stack is depicted below. The bit vector has $n = 64$ bits, and elements are considered small, iff they are smaller than $2\lceil\lg n\rceil = 12$. For clarity, the bits belonging to each element are grouped in a box. For large elements, the leftmost six bits in the box are the annotation in binary representation, and the rightmost 6 bits are the value of the element in binary representation.



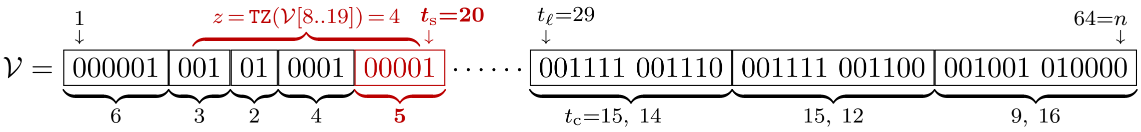
First we push the value 14, which is a large element. Therefore we write the binary representation $14 = (001110)_2$ to interval $\mathcal{V}[t_\ell - 6..t_\ell) = \mathcal{V}[35..40]$. We annotate the large element by writing the current value of $t_s = 15 = (001111)_2$ to interval $\mathcal{V}[t_\ell - 12..t_\ell - 6) = \mathcal{V}[29..34]$. The index t_ℓ gets decreased by $\lceil 2\lg n\rceil = 12$ (from 41 to 29).



Next we push value 5, which is a small element. We set the bit $\mathcal{V}[t_s + 5] = \mathcal{V}[20]$ to 1 and increase t_s by 5 (from 15 to 20).



Now we start popping values. Since $t_s = 20 > 15 = t_c$ holds, we know that the top element is a small element. Therefore, we count the trailing zeros $z = \mathtt{TZ}(\mathcal{V}[t_s - 12..t_s)) = \mathtt{TZ}(\mathcal{V}[8..19]) = 4$. The top element is $e = z + 1 = 5$. We set bit $\mathcal{V}[t_s] = \mathcal{V}[20]$ to zero and decrease t_s by $e = 5$ (from 20 to 15).



$$\mathcal{V} = \underbrace{\boxed{\overset{1}{\overset{\downarrow}{0}}00001}}_{6}\underbrace{\boxed{001}}_{3}\underbrace{\boxed{01}}_{2}\underbrace{\boxed{000\overset{t_s=15}{\overset{\downarrow}{1}}}}_{4}\ 00000\ \cdots\cdots\ \underbrace{\boxed{\overset{t_\ell=29}{\overset{\downarrow}{0}}01111\ 001110}}_{t_c=15,\ 14}\underbrace{\boxed{001111\ 001100}}_{15,\ 12}\underbrace{\boxed{001001\ 01000\overset{64=n}{\overset{\downarrow}{0}}}}_{9,\ 16}$$

If we perform another pop, we have $t_s = 15 = t_c$, indicating that the top element is a large element. Its value is $(\mathcal{V}[t_\ell + 6..t_\ell + 12))_2 = (001110)_2 = 14$. We overwrite the interval $\mathcal{V}[t_\ell..t_\ell + 12)$ with zeros and increase t_ℓ by $\lceil 2 \lg n \rceil = 12$ (from 29 to 41).

$$\mathcal{V} = \underbrace{\boxed{\overset{1}{\overset{\downarrow}{0}}00001}}_{6}\underbrace{\boxed{001}}_{3}\underbrace{\boxed{01}}_{2}\underbrace{\boxed{000\overset{t_s=15}{\overset{\downarrow}{1}}}}_{4}\ \cdots\cdots\cdots\cdots\ \underbrace{\boxed{\overset{t_\ell=29}{\overset{\downarrow}{0}}01111\ 001110}}_{t_c=15,\ 14}\underbrace{\boxed{001111\ 001100}}_{15,\ 12}\underbrace{\boxed{001001\ 01000\overset{64=n}{\overset{\downarrow}{0}}}}_{9,\ 16}$$

$$\mathcal{V} = \underbrace{\boxed{\overset{1}{\overset{\downarrow}{0}}00001}}_{6}\underbrace{\boxed{001}}_{3}\underbrace{\boxed{01}}_{2}\underbrace{\boxed{000\overset{t_s=15}{\overset{\downarrow}{1}}}}_{4}\ \cdots\cdots\cdots\cdots\ 000000\ 000000\ \underbrace{\boxed{\overset{t_\ell=41}{\overset{\downarrow}{0}}01111\ 001100}}_{t_c=15,\ 12}\underbrace{\boxed{001001\ 01000\overset{64=n}{\overset{\downarrow}{0}}}}_{9,\ 16}$$

Cache Efficiency and Dynamic Memory Allocation

There are some aspects of the unary stack that are not relevant in terms of theoretical bounds, but have important implications in practice. The left and the right part of the bit vector $\mathcal{V}$ can be seen as two individual stacks: On the left side we have a stack that contains small elements in unary representation, while on the right side we have a stack that contains annotated large elements in binary representation. Since we only require top access on both of these individual stacks, we have excellent cache locality in practice. Also, instead of using a statically allocated bit vector of size n bits, we can use two separate bit vectors which dynamically grow and shrink depending on their content. We evaluate the efficiency of static and dynamic stack implementations in Section 7.4.

6.2.3 Succinct Telescope Stack

When executing xss-real (or any of the other new algorithms), the stack H always contains only strictly monotonically increasing elements, because we iterate over the indices in ascending order and push index i in iteration i. Instead of storing absolute values, we can store each stack element as the difference to the element that lies on top of it. This keeps the sum of the stack elements below n at all times, allowing us to deploy a unary stack.

6.2.4 Corollary (Telescope Stack). *There is a stack with constant time push, pop, and top operations that stores up to n elements from the interval $[1, n]$ using $n + \mathcal{O}(\lg n \cdot \lg \lg n)$ bits of memory, if the elements on the stack are always strictly monotonically increasing. It can be initialized in $\mathcal{O}(\lg n)$ time.*

Proof. We use a unary stack $\mathcal{U}$ with bit vector length n. Instead of storing the strictly monotonically increasing elements $h_1, \ldots, h_k$, we store $\mathcal{U} = (h_2 - h_1), (h_3 - h_2), \ldots, (h_k - h_{k-1})$ and a dedicated variable $h_{\text{top}} = h_k$ for the topmost element. This can be achieved by implementing the operations as follows:

***push*(*e*):** Perform $\mathcal{U}.push(e - h_{\text{top}})$ and set $h_{\text{top}} \leftarrow e$.
***pop*():** Set $h_{\text{top}} \leftarrow (h_{\text{top}} - \mathcal{U}.top())$ and perform $\mathcal{U}.pop()$.
***top*():** Return h_{top}.

Let $(h_2 - h_1), (h_3 - h_2), \ldots, (h_k - h_{k-1})$ be the content of $\mathcal{U}$ at any point in time, then the sum of its elements is bound by $\sum_{i=2}^{k} h_i - h_{i-1} = h_k - h_1 < n$. Therefore, a bit vector of length n is large enough for $\mathcal{U}$. The $\mathcal{O}(\lg n)$ initialization time and additional memory of $\mathcal{O}(\lg n \lg\lg n)$ bits result from the use of the unary stack (see Lemma 6.2.3). □

6.3 Embedding $\mathcal{R}$ in the BPS

When performing the amortized look-ahead, we use an additional stack $\mathcal{R}$. We pop some of the topmost elements of H, and push them onto $\mathcal{R}$, which reverses their order. As we have seen in Section 5.2.3, these elements are from the interval $(i, i + \chi)$, where χ is the anchor that we have computed as described in Section 5.2.1. Once we have pushed the elements onto $\mathcal{R}$, they are strictly monotonically *decreasing*. With a slight modification, we can use a telescope stack as described in the previous section to represent $\mathcal{R}$.

We use the function $f(x) = i + \chi - x$ to map the elements on the interval $[1..\chi)$. Note that this function is self-inverse, i.e. we have $f^{-1}(x) = f(x)$. By applying f, the elements become strictly monotonically *increasing*, and therefore satisfy the requirements of Corollary 6.2.4. Let $\mathcal{F}$ be a telescope stack of length χ as shown in the corollary. Then we can realize the operations on $\mathcal{R}$ as follows:

***push*(*e*):** Perform $\mathcal{F}.push(f(e))$.
***pop*():** Perform $\mathcal{F}.pop()$.
***top*():** Return $f(\mathcal{F}.top())$.

The stack $\mathcal{F}$ needs $\chi + \mathcal{O}(\lg \chi \cdot \lg\lg \chi)$ bits of memory. However, even after performing the amortized look-ahead, there are at least 3χ more indices to process (see Section 5.2.1; we have $i + 4\chi < n$). Therefore, there are at least 6χ more parenthesis to write, and thus 6χ unused bits at the end of the BPS. For sufficiently large χ, we have $\chi + \mathcal{O}(\lg \chi \cdot \lg\lg \chi) < 6\chi$. Consequently, we can embed $\mathcal{R}$ into the unused space of the BPS.

6.3.1 Lemma. *We can embed the stack $\mathcal{R}$ of the amortized look-ahead into the so far unwritten part of the BPS. All operations can be performed in constant time, and the initialization time is $\mathcal{O}(\lg \chi)$. No additional memory is needed.*

Proof. The correctness follows from the description above. The initialization time is used for the telescope stack $\mathcal{F}$ (see Corollary 6.2.4). □

6.4 Maintaining L in $\lceil 4n/\delta \rceil + o(n)$ Bits

The sum of LCP values on the stack L of the new algorithms can reach $\Theta(n^2)$ (for example for the text "$\texttt{\$a}^{n-3}\texttt{b\$}$"), and the elements are not necessarily strictly monotonically increasing or decreasing. Therefore neither the unary stack nor the telescope stack seem like natural solutions for realizing L. However, a new parameterized representation of the elements of L allows us to fit them on a unary stack of bit vector length $\frac{4n}{\delta}$. Here, δ is an arbitrary constant positive integer that reduces the number of bits needed for the elements on the stack by rounding them down to the closest multiple of δ. When retrieving stack elements, we have to restore the information that got lost while rounding the elements, increasing the runtime of pop operations to $\mathcal{O}(\delta)$. Both push and top operations can still be performed in constant time. We first explain the new representation of LCP values, the underlying data structures of our succinct version of L, and the way we execute push, pop, and top operations. Then, we show that our representation in fact achieves the claimed space bounds. By the end of this section we will have proven:

6.4.1 Lemma (Succinct LCP Stack). *Let $\delta \in \mathbb{N}^+$ with $\delta \leq \sqrt{n}/(3\lg n)$. There is a representation of the stack L used in the algorithms xss-bps-lcp and xss-real that allows constant time push and top operations, as well as $\mathcal{O}(\delta)$ pop operations, while using only $4n/\delta + \mathcal{O}(\lg(n/\delta) \cdot \lg\lg(n/\delta))$ bits of memory. It can be initialized in $\mathcal{O}(\lg(n/\delta))$ time.*

6.4.1 Transformation of LCP values

Let $l_1, \ldots, l_k$ be the elements on the LCP stack L at any point in time, and let l_i with $i \in [1, k)$ be any element except for the topmost one. We define the *δ-representation* l_i^δ of l_i for $\delta \in \mathbb{N}^+$ as:

$$l_i^\delta = \begin{cases} \lfloor l_i/\delta \rfloor & \text{, iff } l_i < l_{i+1} \text{ (called \textit{absolute value})} \\ \lfloor (l_i - l_{i+1})/\delta \rfloor & \text{, iff } l_i \geq l_{i+1} \text{ (called \textit{relative value})} \end{cases}$$

For the topmost element l_k we define $l_k^\delta = 0$. Evidently, transforming l_i to l_i^δ takes constant time. Note that reverting the transformation is more complicated, because (without additional information) we do not know whether the δ-representation of an LCP value is an absolute or a relative value. Also we lose some information due to rounding during the transformation. Even if both l_i^δ and l_{i+1} are given, we only know that l_i is either from the interval $[\delta l_i^\delta, \delta l_i^\delta + \delta)$ (if it is an absolute value), or from the interval $[l_{i+1} + \delta l_i^\delta, l_{i+1} + \delta l_i^\delta + \delta)$

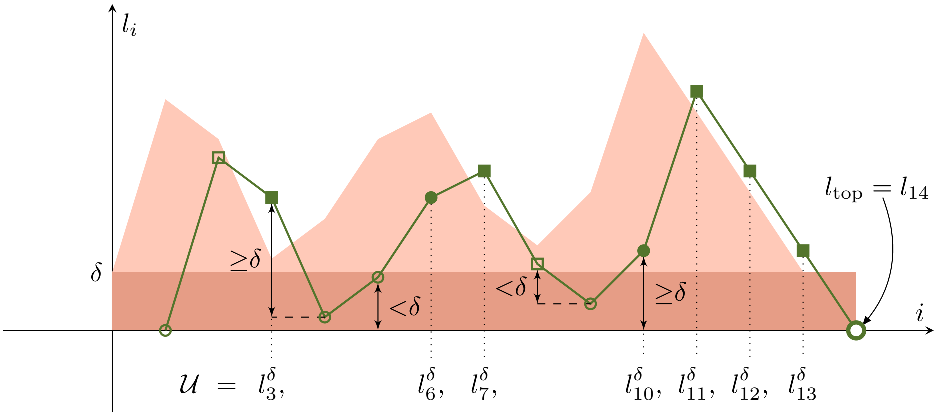


Figure 6.3: Storing the δ-representation of the elements $l_1, \ldots, l_{14}$ of L on a unary stack $\mathcal{U}$. The topmost element l_{14} is stored in the dedicated variable l_{top}. Each data point represents one element l_i. We use a circle marker, if l^δ_i is an absolute value (i.e. $l_i < l_{i+1}$), and a square marker, if l^δ_i is a relative value (i.e. $l_i \geq l_{i+1}$)). The dark red area contains the elements that are smaller than δ. The light red area (combined with the dark red area) contains the elements l_i with $l_i - l_{i+1} < \delta$. On the stack $\mathcal{U}$ we only store values with $l^\delta_i > 0$. For absolute values, this means that l_i must be at least δ. Therefore, the absolute values in the dark red area are not stored on $\mathcal{U}$. For relative values, $l_i - l_{i+1}$ must be at least δ. Therefore, the relative values in both the light and dark red area are not stored on $\mathcal{U}$. All elements that are stored on $\mathcal{U}$ have a filled marker.

(if it is a relative value). In the next section we show how to restore l_i from l^δ_i and l_{i+1} in $\mathcal{O}(\delta)$ time.

6.4.2 Using a Unary Stack

The core of our succint representation of L is a unary stack $\mathcal{U}$ of length $\frac{4n}{\delta}$. Let $l_1, \ldots, l_k$ be the elements of L at any point in time, and let $l_{d_1}, \ldots, l_{d_m}$ be exactly the elements whose δ-representation is greater than zero (retaining the order, i.e. $d_1 < d_2 < \cdots < d_m$). We store the values $l^\delta_{d_1}, \ldots, l^\delta_{d_m}$ on the unary stack $\mathcal{U}$, while a dedicated variable l_{top} stores the topmost LCP value l_k. Figure 6.3 visualizes the elements l_i and the stack $\mathcal{U}$. Since we maintain the variable l_{top} at all times, retrieving the top element of L in constant time is trivial. We now explain how to push and pop elements such that the stack always works as described.

***push*$(\boldsymbol{l_{k+1}})$:** Assume that $l_{\text{top}} = l_k$ is the topmost element on L and we want to push l_{k+1} on top of it. Clearly, using l_{k+1} we can easily compute the δ-representation l^δ_k of l_k. If it is greater than zero, then we push it onto $\mathcal{U}$. In any case, we set $l_{\text{top}} \leftarrow l_{k+1}$. This takes constant time.

***pop*$()$:** Assume that $l_{\text{top}} = l_k$ is the topmost element on L. After popping l_k, we want $l_{\text{top}} = l_{k-1}$ to hold. Therefore we have to somehow restore l_{k-1}. If its δ-representation

l^δ_{k-1} is greater than zero, then it lies on top of $\mathcal{U}$. Otherwise we have no information at all about l_{k-1}. We only know that l_{k-1} is the LCP value $\textsc{Lcp}_S(h_{k-1}, h_k)$. The topmost three elements on H are exactly h_{k-1}, h_k and h_{k+1}. Since we have constant time push and pop operations on H, we can easily lookup h_{k-1} and h_k in constant time.
We now describe how to restore l_{k-1} in $\mathcal{O}(\delta)$ time using the topmost element u of $\mathcal{U}$, the current top value $l_{\text{top}} = l_k$, and the indices h_{k-1} and h_k. Depending on the transformation l^δ_{k-1} of l_{k-1}, one of the following cases applies:

1. The transformation l^δ_{k-1} is an absolute value, i.e. we have $l^\delta_{k-1} = \lfloor l_{k-1}/\delta \rfloor$. Therefore, l_{k-1} is from the interval $[\delta l^\delta_{k-1}, \delta l^\delta_{k-1} + \delta)$, as visualized in Figure 6.4 (top). If $l^\delta_{k-1} > 0$ holds, then we have stored the element l^δ_{k-1} on the stack $\mathcal{U}$, and thus we have $u = \mathcal{U}.top() = l^\delta_{k-1}$. In this case, l_{k-1} is from the interval $\mathcal{C}_1 = [\delta u, \delta u + \delta)$. Otherwise, we have $l^\delta_{k-1} = 0$ and l_{k-1} is from the interval $\mathcal{C}_2 = [0, \delta)$.

2. The transformation l^δ_{k-1} is a relative value, i.e. we have $l^\delta_{k-1} = \lfloor (l_{k-1} - l_k)/\delta \rfloor$. Therefore, l_{k-1} is from the interval $[l_k + \delta l^\delta_{k-1}, l_k + \delta l^\delta_{k-1} + \delta)$, as visualized in Figure 6.4 (bottom). If $l^\delta_{k-1} > 0$ holds, then we have stored the element l^δ_{k-1} on the stack $\mathcal{U}$, and thus we have $u = \mathcal{U}.top() = l^\delta_{k-1}$. In this case, l_{k-1} is from the interval $\mathcal{C}_3 = [l_k + \delta u, l_k + \delta u + \delta)$. Otherwise, we have $l^\delta_{k-1} = 0$ and l_{k-1} is from the interval $\mathcal{C}_4 = [l_k, l_k + \delta)$.

At this point, we neither know if l^δ_{k-1} is an absolute or a relative value, nor do we know if $l^\delta_{k-1} > 0$ holds. However, we can define the candidate set $\mathcal{C}_{\text{all}} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \mathcal{C}_4$, which definitely contains l_{k-1}. Since by definition of the stack L we have $l_{k-1} = \textsc{Lcp}_S(h_{k-1}, h_k)$, we know that the suffixes $S_{h_{k-1}}$ and S_{h_k} share the longest common prefix $S[h_{k-1}..h_{k-1} + l_{k-1}) = S[h_k..h_k + l_{k-1})$, and then mismatch on the next character, i.e. we have $S[h_{k-1} + l_{k-1}] \neq S[h_k + l_{k-1}]$. Therefore, we can use the set $\mathcal{C}_{\text{all}}$ to determine the value of l_{k-1}:

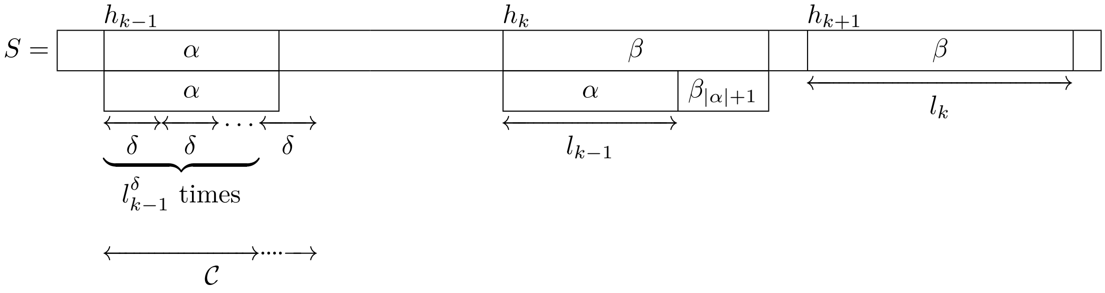
$$l_{k-1} = \min\{l \mid l \in \mathcal{C}_{\text{all}} \ \wedge \ S[h_{k-1} + l] \neq S[h_k + l]\}$$

There are only $|\mathcal{C}_{\text{all}}| \leq 4\delta = \mathcal{O}(\delta)$ candidates to consider, and each one of them can be validated in constant time. Therefore, we can determine l_{k-1} by trying all possible values in $\mathcal{O}(\delta)$ time. Once we know l_{k-1}, we can easily check if $l^\delta_{k-1} > 0$ holds. If it does, then l^δ_{k-1} is the topmost element on $\mathcal{U}$, and we pop it from $\mathcal{U}$. In any case we set $l_{\text{top}} \leftarrow l_{k-1}$.

6.4.3 Proving the Space Bound

It remains to be shown, that $4n/\delta$ bits are sufficient for $\mathcal{U}$. Let $h_1, \ldots, h_{k+1}$ be the elements of H and let $l_1, \ldots, l_k$ with $l_i = \textsc{Lcp}_S(h_i, h_{i+1})$ be the elements of L at any point in time

Absolute value $\boldsymbol{l^\delta_{k-1}}$: $l^\delta_{k-1} = \lfloor l_{k-1}/\delta \rfloor$ $\mathcal{C} = [\delta l^\delta_{k-1}, \delta l^\delta_{k-1} + \delta)$



Relative value $\boldsymbol{l^\delta_{k-1}}$: $l^\delta_{k-1} = \lfloor (l_{k-1} - l_k)/\delta \rfloor$ $\mathcal{C} = [l_k + \delta l^\delta_{k-1}, l_k + \delta l^\delta_{k-1} + \delta)$

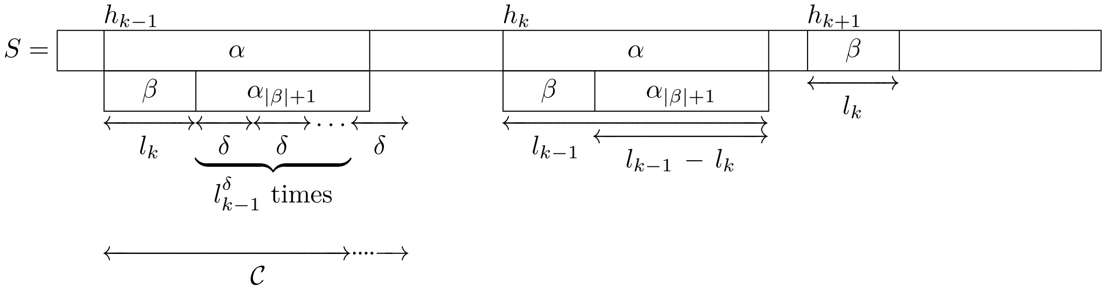


Figure 6.4: Candidate intervals for l_{k-1}. The suffixes S_{h_k} and $S_{h_{k+1}}$ share a longest common prefix β of length $l_k = \textsc{Lcp}_S(h_k, h_{k+1})$, while the suffixes $S_{h_{k-1}}$ and S_{h_k} share a longest common prefix α of length $l_{k-1} = \textsc{Lcp}_S(h_{k-1}, h_k)$. We differentiate between absolute values (top) and relative values (bottom). Using only the δ-representation l^δ_{k-1}, as well as l_k, we can define the interval $\mathcal{C}$, which contains l_{k-1}.

during the execution of xss-real or xss-bps-lcp. For simplicity, we assume that l^δ_i is defined for all $i \in [1, k)$, such that $\mathcal{U}$ contains exactly the elements $l^\delta_1, \dots, l^\delta_{k-1}$. We prove the space bound by showing that for each l^δ_i at least one of the following cases applies:

Case (1): The element l^δ_i uses at most $(2(h_{i+1} - h_i) + (h_{i+2} - h_{i+1}))/\delta$ bits. Even if we assume that this applies for every l^δ_i, then the total number of bits used by this case is less than $3n/\delta$:

$$\begin{aligned}
& (1/\delta) \cdot \sum_{i=1}^{k-1} 2(h_{i+1} - h_i) + (h_{i+2} - h_{i+1}) \\
< \; & (1/\delta) \cdot \sum_{i=1}^{k} 3(h_{i+1} - h_i) \\
= \; & (1/\delta) \cdot 3(h_{k+1} - h_1) \\
< \; & (1/\delta) \cdot 3n
\end{aligned}$$

Case (2): The element l_i^δ is one of the topmost $\sqrt{n}$ elements of $\mathcal{U}$. On a unary stack of length $\lceil 4n/\delta \rceil$ (as defined in the proof of Lemma 6.2.3), each element uses at most $\lceil 2\lg(4n/\delta)\rceil$ bits of the bit vector. For sufficiently large n and $\delta \geq 1$, we have $\lceil 2\lg(4n/\delta)\rceil < 3\lg n$. Assume that δ is at most $\sqrt{n}/(3\lg n)$, then we obtain the following upper bound for the number of bits used by the topmost $\sqrt{n}$ elements:

$$\underbrace{\sqrt{n}}_{\substack{\text{number}\\ \text{of elements}}} \cdot \underbrace{\lceil 2\lg(4n/\delta)\rceil}_{\substack{\text{max. bits}\\ \text{per element}}} < \sqrt{n} \cdot (3\lg n) = n/\underbrace{(\sqrt{n}/(3\lg n))}_{\geq\delta} \leq n/\delta$$

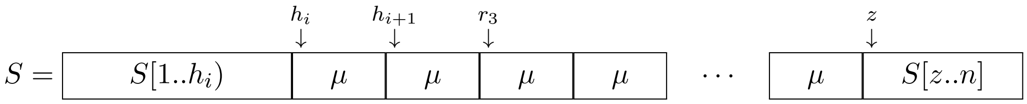
Clearly, if we can show that each element on $\mathcal{U}$ is covered by at least one of the two cases, then we have proven the space bound. Let l_i^δ be any element on $\mathcal{U}$, and let b be the number of bits it occupies in the bit vector of $\mathcal{U}$. We now show that if Case (1) does not apply, then Case (2) does apply. Assume that Case (1) does not apply, i.e.:

$$\text{Assumption } \boxed{\mathbf{A}}\mathbf{:}\ b > (2(h_{i+1} - h_i) + (h_{i+2} - h_{i+1}))/\delta$$

6.4.2 Observation. The definition of l_i^δ implies that $l_i \geq \delta l_i^\delta$ holds. Also, since every element e of a unary stack as defined in the proof of Lemma 6.2.3 uses at most e bits, we have $l_i^\delta \geq b$. Combining the two inequalities, we can show that l_i is at least twice as large as the length of interval $[h_i, h_{i+1})$:

$$l_i \;\geq\; \delta l_i^\delta \;\geq\; \delta b \;\underset{\boxed{\mathbf{A}}}{>}\; 2(h_{i+1} - h_i) + (h_{i+2} - h_{i+1}) \;>\; 2(h_{i+1} - h_i)$$

This observation has important consequences. Remember, that $l_i = \textsc{Lcp}_S(h_i, h_{i+1})$ is the length of the longest common prefix between the suffixes S_{h_i} and $S_{h_{i+1}}$. Therefore, $l_i > 2(h_{i+1} - h_i)$ implies that index h_i is part of a Lyndon run. More precisely, the substring $\mu = S[h_i, h_{i+1})$ repeats itself at least three times at the beginning of S_{h_i} (see Section 5.1.1 for details). Let $r_3 = h_{i+1} + |\mu|$ be the starting position of the third repetition and let $z = h_{i+1} + |\mu| \cdot \lfloor l_i/|\mu| \rfloor$ be the first index after the last repetition of μ.



Now we try to place h_{i+2} in the figure above. We step-by-step eliminate intervals that may contain h_{i+2}, until we have shown that $h_{i+2} \in (h_{i+1}, r_3)$ holds. Then, we can show that l_i^δ is one of the topmost $\sqrt{n}$ elements.

Eliminating $\boldsymbol{h_{i+2} \in [z, n]}$:
We have $|\mu| = h_{i+1} - h_i$. From Observation 6.4.2 we know that $l_i > 2|\mu| + (h_{i+2} - h_{i+1})$ holds. Using this inequality and the definition of z, we obtain an upper bound for h_{i+2}:

$$\begin{aligned} z &= h_{i+1} + |\mu| \cdot \lfloor l_i/|\mu| \rfloor \\ &\geq h_{i+1} + l_i - |\mu| \\ &> h_{i+1} + (2|\mu| + (h_{i+2} - h_{i+1})) - |\mu| \\ &= |\mu| + h_{i+2} \;>\; h_{i+2} \end{aligned}$$

Eliminating $\boldsymbol{h_{i+2} \in (r_3, z)}$:
This follows directly from Lemma 5.1.10. (If r_3 is not an element on H, but a larger index is, then H does not contain any element from the interval $[r_3, z]$.)

Eliminating $\boldsymbol{h_{i+2} = r_3}$:
Assume that $h_{i+2} = r_3$ holds. Then h_i is the starting position of the first repetition of μ, h_{i+1} is the starting position of the second repetition, and $h_{i+2} = r_3$ is the starting position of the third repetition. Consequently, the longest common prefix between $S_{h_{i+1}}$ and $S_{h_{i+2}}$ is exactly one repetition of μ (and thus $|\mu|$ characters) shorter than the longest common prefix between S_{h_i} and $S_{h_{i+1}}$:

$$l_{i+1} = \textsc{Lcp}_S(h_{i+1}, h_{i+2}) = \textsc{Lcp}_S(h_i, h_{i+1}) - |\mu| = l_i - |\mu|$$

This implies that l_i^δ is a relative value because $l_{i+1} < l_i$ holds. By definition of the δ-representation we have $l_i^\delta = \lfloor (l_{i+1} - l_i)/\delta \rfloor = \lfloor |\mu|/\delta \rfloor \leq (h_{i+1} - h_i)/\delta$. However, this means that l_i^δ also uses at most $(h_{i+1} - h_i)/\delta$ bits of the unary stack, which contradicts assumption $\boxed{\mathbf{A}}$.

Exploiting $\boldsymbol{h_{i+2} \in (h_{i+1}, r_3)}$:
The only case left is $h_{i+2} \in (h_{i+1}, r_3)$. It follows directly from Lemma 5.1.9, that the topmost element on H is smaller than r_3. Therefore, h_{i+1} is one of the topmost $(r_3 - h_{i+1}) = (h_{i+1} - h_i)$ elements. Remember, that each element on the unary stack uses at most $\lceil 2 \lg(4n/\delta) \rceil$ bits. Under assumption $\boxed{\mathbf{A}}$, and for sufficiently large n, we obtain the following bound for $h_{i+1} - h_i$:

$$h_{i+1} - h_i \underset{\boxed{\mathbf{A}}}{<} \delta b/2 \;\leq\; \delta \lceil 2 \lg(4n/\delta) \rceil / 2 \;<\; \delta \cdot 3 \lg n \;\leq\; \sqrt{n}$$

The last step of this chain of inequalities only holds for $\delta < \sqrt{n}/(3 \lg n)$, explaining the limitation of δ in Lemma 6.4.1. Since we have processed less than $\sqrt{n}$ indices after processing h_{i+1}, we know that l_i^δ is one of the topmost $\sqrt{n}$ elements on the unary stack $\mathcal{U}$. Thus we have proven that if Case (1) does not apply, then Case (2) does apply. Therefore,

$\lceil 4n/\delta \rceil$ bits are sufficient for $\mathcal{U}$. Consequently, the additional lower order memory term is $\mathcal{O}(\lg(n/\delta) \cdot \lg\lg/(n/\delta)))$, and the initialization time is $\mathcal{O}(\lg(n/\delta))$ (see Lemma 6.2.3). Thus, we have proven the correctness of Lemma 6.4.1

6.5 Combining the Pieces

The space efficient data structures of the previous sections finally allow us to lower the memory bound of xss-real. In order to achieve the lowest possible theoretical worst-case bounds, we use the $\mathcal{O}(\sqrt{n} \cdot \lg n)$ bit version of the index stack H (Lemma 6.1.1). Whenever we have to use the reversal stack $\mathcal{R}$ during the amortized look-ahead, we embed it in the unused space of the BPS, causing no additional memory usage (Lemma 6.3.1). Finally, we use the parameterized version of the LCP stack L, which uses $\lceil 4n/\delta \rceil + \mathcal{O}(\lg(n/\delta) \cdot \lg\lg/(n/\delta)))$ bits of memory (Lemma 6.4.1). Apart from the stacks, xss-real uses only $\mathcal{O}(1)$ words of additional memory for variables. Therefore, the total memory usage (without input and output) is bound by $\lceil 4n/\delta \rceil + \mathcal{O}(\sqrt{n} \cdot \lg n)$ bits.

Using the described stack representations, we can still perform constant time push and top operations. However, popping elements from L takes $\mathcal{O}(\delta)$ time. Since we pop at most n elements during the entire execution of xss-real, it follows that the time bound increases to $\mathcal{O}(\delta n)$. Thus, we have proven Theorem 4.0.1:

4.0.1 Theorem. *Let S be a string of length n and let $\delta \in [1, \lfloor \sqrt{n}/(3\lg n) \rfloor]$. The BPS of the PSS tree of S can be computed in $\mathcal{O}(\delta n)$ time using $\lceil 4n/\delta \rceil + \mathcal{O}(\sqrt{n} \cdot \lg n)$ bits of additional memory apart from the space needed for input and output.*

Chapter 7

Experimental Evaluation

7.1 Experimental Setup

We conduct our experiments on the LiDO3 cluster of the TU Dortmund[1]. For all benchmarks we use a standard compute node of the cluster, which comes with an Intel Xeon E5-2640V4 processor and 64 GB of memory. The processor runs at 2.4GHz (Intel Turbo Boost disabled) and features 25 MB of L3 cache. The word size is $w = 64$ bits. The implementation is written in C++17, and compiled with GCC using the compiler flags `-O3 -ffast-math -funroll-loops -march=native`. The same compiler and flags are used for algorithms that are part of third party libraries and repositories. Time measurements are performed using `std::chrono::high_resolution_clock`, while memory measurements utilize the `malloc_count` library[2]. All implemented algorithms and data structures are publicly available under https://github.com/jonas-ellert/xss-real.

7.2 Counting Trailing Zeros

First, we evaluate the query time of trailing zero queries, using the data structures shown in Section 6.2.1. We also measure the `TZCNT` instruction as a baseline. In practice we only want to answer queries for entire computer words, i.e. we have $b = 64$.

Implementing the de Bruijn Method

For the de Bruijn method we use the multiplier $a =$ `0x03f79d71b4ca8b09` (*Martin's constant*, see [Knuth, 2011, p. 142]) and the following lookup table of size 64 bytes:

[1] https://www.lido.tu-dortmund.de/cms/de/LiDO3/index.html

[2] https://github.com/bingmann/malloc_count

```
constexpr static uint64_t martins_constant = 0x03f79d71b4ca8b09;
constexpr static uint8_t lookup_[64] = {
    0,  1,  56, 2,  57, 49, 28, 3,  61, 58, 42, 50, 38, 29, 17, 4,
    62, 47, 59, 36, 45, 43, 51, 22, 53, 39, 33, 30, 24, 18, 12, 5,
    63, 55, 48, 27, 60, 41, 37, 16, 46, 35, 44, 21, 52, 32, 23, 11,
    54, 26, 40, 15, 34, 20, 31, 10, 25, 14, 19, 9,  13, 8,  7,  6
};
```

Assuming $x \neq 0$, we then retrieve the number of trailing zeros as follows:

```
constexpr inline static uint64_t ctz(const uint64_t x) {
    return lookup_[((x & -x) * martins_constant) >> 58];
}
```

Note that since we are operating on entire computer words, we do not need to use the modulo operation (cf. Section 6.2.1, $h(2^z) = ((a \cdot 2^z) \bmod 2^b) \gg (b - \lg b)$). Usually, the modulo acts as a mask that selects the lowest b bits. However, in practice the multiplication $a \cdot 2^z = (x \& -x) \cdot$ `martins_constant` overflows, and naturally only the lowest 64 bits remain.

Implementing the Binary Search

We try the values $2, 4, 8$ and 16 for the parameter c of the binary search, resulting in lookup tables of size $3, 9, 129$ and 32769 bytes respectively. For example, we have the following table for $c = 4$:

```
constexpr static uint8_t lookup_[9] = { 0, 0, 1, 0, 2, 0, 0, 0, 3 };
```

Assuming $x \neq 0$, we then retrieve the number of trailing zeros as follows:

```
constexpr static uint64_t c_div_2 = c >> 1;
constexpr inline static uint64_t ctz(uint64_t x) {
    x &= -x;
    uint64_t bits = 32;
    uint64_t mask = 0xffffffff;
    uint64_t result = 0;
    while (bits > c_div_2) {
        if ((x & mask) == 0) {
            result += bits;
            x >>= bits;
        }
        bits >>= 1;
        mask >>= bits;
    }
    return result + lookup_[x];
}
```

Additionally, we measure the binary search without using a lookup table.

Testing Methodology

We do not take the initialization time of lookup tables into account. All tables are small enough to be built at compile time, such that no additional time is needed at run time. For the same reason, we do not measure the memory usage of the different methods. Even the largest table of the binary search with $c = 16$ only occupies 32 KiB, which is insignificant in practice. When answering a query, we always have to check if the given word is zero. Since we have to catch this special case for all methods (including the `TZCNT` instruction, which is undefined for zero), we do not measure the overhead caused by zero-queries. Instead, we only use non-zero queries for the evaluation, such that we do not have to catch the special case at all.

Our benchmark works as follow: For each possible result of a trailing zero query, i.e. for $z \in [0, 63]$, we generate 2^{30} uniformly distributed random words that have z trailing zeros, i.e. words from the set $\{x \mid x \in [1, 2^{64}) \land \texttt{TZ}(x) = z\}$. Then, we take each method (de Bruijn, binary search, `TZCNT`) and answer these 2^{30} queries, measuring the total time. This procedure is repeated five times, of which we take the median time as the final result. After that, we calculate the throughput of each method in queries per nanosecond. We run one additional set of experiments per method, during which we measure the throughput for 2^{30} completely random queries, i.e. without setting a fixed number of trailing zeros.

Results

Figure 7.1 shows how the methods compare. As expected, the dedicated CPU instruction `TZCNT` is the fastest method, answering around 1.37 queries per nanosecond, regardless of the query result, and even for completely random queries. The de Bruijn method is only around 23% slower at roughly 1.05 queries per nanosecond. Again, the query result has no influence on the query time. Looking at completely random queries for the binary search, we see that higher values of c allow more throughput. This was expected, since with growing c fewer steps of the binary search are needed to reach the base case. The throughput varies between around 0.76 (for $c = 16$) and 0.11 (without using a lookup table) queries per nanosecond, which is between around 27% and 89% slower than the de Bruijn Method. It is not surprising, that the query time depends on the result of the queries, since branching left or right during the binary search is not equally expensive.

For the binary search with small lookup tables ($c = 2$, $c = 4$, no lookup table), there is a significant gap between the throughput for completely random queries and the throughput for fixed result queries. Taking $c = 2$ as an example, we answer around 0.19 completely random queries per nanosecond, but for any fixed result we answer more than 0.4 queries per nanosecond. Since the small lookup tables fit into a single cache line, we can rule out

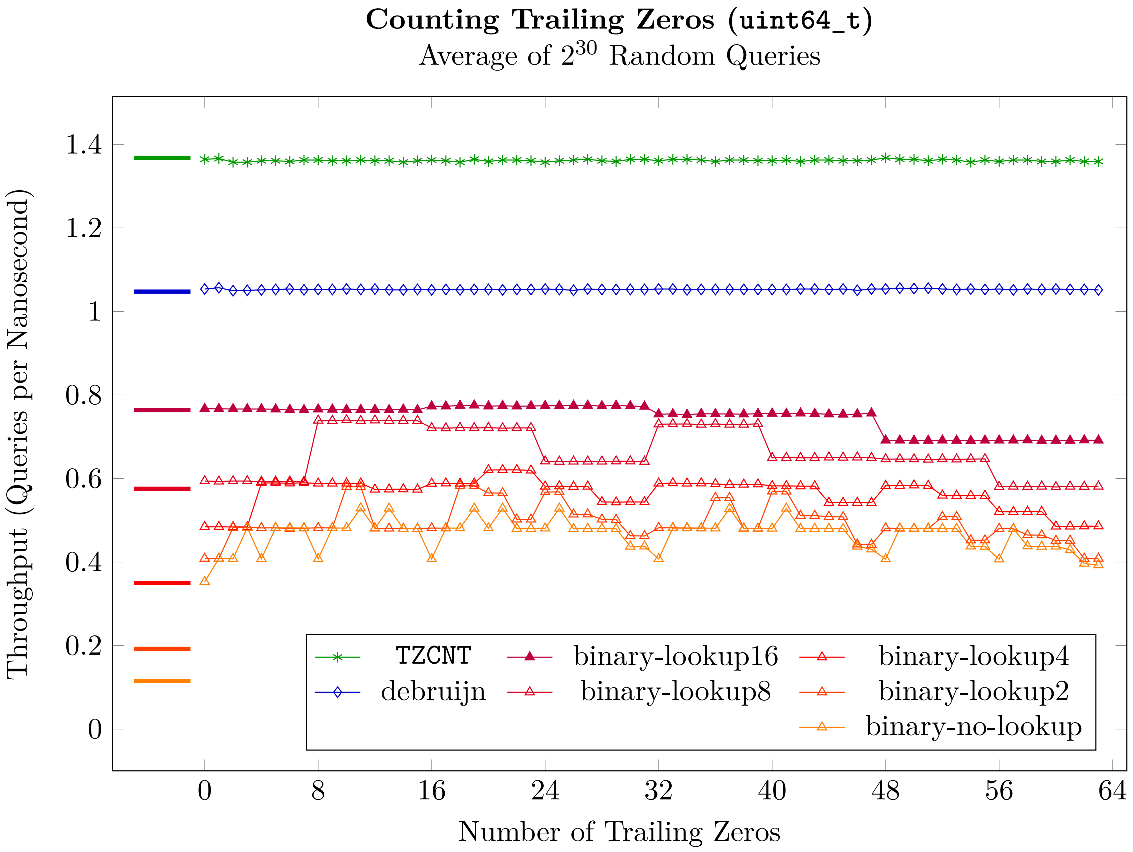


Figure 7.1: Counting Trailing Zeros on 64 bit words. Each data point represents the throughput in queries per nanosecond for a fixed number of trailing zeros. The thick horizontal bars on the left indicate the throughput for completely random queries without a fixed result. See Section 7.2 for details.

caching effects as the cause of this disparity. While we cannot be completely sure about the actual cause, it is a reasonable assumption, that the CPU's branch predictor performs significantly better, if we only answer queries of a fixed result, i.e. queries that branch completely identically during the binary search.

7.3 Introducing the Text Collection

In this section we introduce the texts that we use for most of the following experiments. There are three groups of texts that we consider, all of which can be found in the Pizza & Chili text collection[3]. The first group consists of real life texts that cover a wide range of application areas (`dna`, `english`, `pitches`, `proteins`, `sources`, and `xml`)[4]. The second group consists of highly repetitive real life texts that are less representative for most appli-

[3] http://pizzachili.dcc.uchile.cl/index.html

[4] http://pizzachili.dcc.uchile.cl/texts/

cations, but can reveal strengths and weaknesses of the algorithms that we compare (`cere`, `coreutils`, `ecoli`, `einstein.de`, `einstein.en`, `influenza`, `kernel`, `leaders`, `para`)[5]. Lastly, we also consider artificial texts that are generated with maximum repetitiveness in mind (`fib41`, `rs.13`, `tm29`)[6]. Detailed information on the real and artificial repetitive texts can be found on the official Pizza & Chili website[7].

Statistics

In order to select meaningful texts for the evaluation, we collected some statistical data that indicates how hard it is to compute the PSS tree for each instance (Table 7.1). Apart from the text and alphabet sizes, the table shows some more specific measures that are designed to work with our algorithms.

In the general area of string processing, there are many algorithms that struggle with highly repetitive texts. More often than not, this is due to the very large LCP values that occur in these texts. However, our algorithms only have to compute LCPs for a very narrow selection of indices. We can get a better understanding of the difficulty of each text by running the plain version `xss-bps` of our algorithm and determining the average and maximum LCP value that we actually have to compute during the execution. The higher the average LCP value, the longer we expect our algorithms to take for all necessary suffix comparisons. Interestingly, the artificial repetitive texts only reach average LCP values of 16 and lower (see Table 7.1). The hardest instances seem to be `cere` and `para` with respective average values of 156 and 32 (each of these instances contains a collection of yeast DNA). All of the other non-artificial instances only reach very small values between 0.2 and 1.05. The exceptions are `leaders` (a collection of CIA documents), `pitches` (pitch values of MIDI files), and `einstein.en` (all versions of the English Wikipedia article on Albert Einstein, up to Nov. 2006), for which we have average LCP values between 5.97 and 9.93.

Another indicator for the hardness of processing the texts is the maximum number of elements that we reach on the stack H. If this number is high, then we expect a high memory usage. Apart from the absolute maximum stack size per text, Table 7.1 also shows the relative maximum stack size, which is the maximum stack size divided by $n/100000$. The highest relative stack size occurs for `pitches`, which needs up to one element on the stack per 10000 characters of input.

Lastly, we ran `xss-real` on each text and determined the number of iterations that we skip by using the run extension and the amortized look-ahead. The table shows relative values,

[5] http://pizzachili.dcc.uchile.cl/repcorpus/real/

[6] http://pizzachili.dcc.uchile.cl/repcorpus/artificial/

[7] http://pizzachili.dcc.uchile.cl/repcorpus/statistics.pdf

Table 7.1: Statistics on the text collection. Large average LCP values imply less throughput for our algorithms. Large maximum stack sizes imply a higher memory usage.

			xss-bps		xss-bps		xss-real	
	n		computed LCPs		max. stack size		skippable iter.	
Text	MiB	σ	avg.	max.	abs.	per 100k	RE %	AL %
`fib41`	256	2	15.20	63 245 987	21	0.01	0.00	98.85
`rs.13`	207	2	9.54	41 395 035	27	0.01	0.00	93.99
`tm29`	256	2	8.78	67 108 863	20	0.01	0.00	91.42
`cere`	440	5	156.24	46 417	34 488	7.48	7.15	2.40
`coreutils`	196	236	0.76	1 593 942	3072	1.50	0.07	0.82
`ecoli`	107	15	0.68	104 863	104	0.09	0.01	0.03
`einstein.de`	88	117	0.59	192 872	176	0.19	0.18	14.07
`einstein.en`	446	139	7.30	726 916	730	0.16	0.87	13.55
`influenza`	148	15	0.79	21 640	624	0.40	0.38	1.98
`kernel`	246	160	0.51	811 399	282	0.11	0.02	0.72
`leaders`	45	89	9.93	50 228	122	0.26	0.65	0.44
`para`	409	5	31.73	17 271	16 162	3.77	3.83	0.03
`dna`	385	16	0.78	1 378 596	125	0.03	0.01	0.09
`english.1G`	1024	237	0.20	579 923	10 057	0.94	0.00	0.07
`pitches`	53	133	5.97	20 336	5771	10.34	7.77	1.19
`proteins`	1126	27	0.37	90 246	744	0.06	2.32	1.27
`sources`	201	230	1.05	98 698	6952	3.30	0.36	0.20
`xml`	282	97	0.25	50 228	552	0.19	0.01	0.00

i.e. the percentage of iterations that we can skip. Interestingly, the run extension is not applicable for the artificial instances. The text `rs.13` is a run-rich string sequence that contains over $0.92n$ runs[Franek et al., 2003]. However, the run extension only gets used for runs of at least three repetitions, which do not exist in `rs.13`. The repetitive nature of the artificial instances causes the amortized look-ahead to be very effective, skipping over 90% of all iterations. In general, we can skip more iterations for texts that have higher LCP values, with the exception of `para`, where we can only skip 3.86% of the iterations despite the high average LCP value of almost 32.

7.4 Comparing the Stacks

In terms of stack implementations, we use most of the techniques shown in Chapter 6. The only approach that we do not consider is the $\mathcal{O}(\sqrt{b} \cdot \lg n)$ bit version of the index stack H (see Section 6.1). Achieving an efficient implementation is complicated, and in practice we expect no significant benefit from using this approach over the $n + o(n)$ bit telescope stack representation of H (see Section 6.2.3). We have already seen that for real texts (and even for some highly repetitive artificial ones) the stack size stays low during the entire execution of our algorithms. For the same reason, we do not embed the reversal stack $\mathcal{R}$ in the BPS, but simply store it separately.

Implementing the Stacks

The provided implementations of the unary stack, the telescope stack, and the parameterized LCP stack works almost exactly as described in Chapter 6. For the unary stack we use the `TZCNT` instruction to count trailing zeros, which is the fastest available method (see Section 7.2) and does not need any additional space for lookup tables. Remember, that for the unary stack we differentiate between small elements, which have a value smaller than $2\lceil \lg n \rceil$, and large elements, which are at least $2\lceil \lg n \rceil$. The original motivation behind this differentiation was, that we have to count trailing zeros at most twice while popping elements or retrieving the topmost element. Since the `TZCNT` instruction allows us to count trailing zeros for entire computer words of size 64 bits, we do not make the threshold for small elements dependent on the input size. Instead, we simply define small elements to have at most value 127 (and large elements to have at least value 128).

Our implementation of the stack L works exactly as described in Section 6.4. The parameter δ can assume any positive integer value. Additionally, we provide a slightly different implementation, which is identified with $\delta = 0$ from now on. Essentially, it works the same way as the LCP stack with $\delta = 1$, but also stores the δ-transformations with value zero. We achieve this by simply incrementing all transformed values by one. Since this way the sum of elements on the underlying unary stack increases by at most n, we need n additional bits. Furthermore, we augment each stack element with a type-flag that indicates whether it is an absolute or a relative value, which increases the memory usage by another n bits. Therefore, the LCP stack with $\delta = 0$ uses $6n$ bits of memory. We need the same $4n$ bits as the LCP stack with $\delta = 1$, plus n bits to allow the value zero, plus n bits for the type-flags. The added type-flags are sufficient to restore each original LCP value from its δ-transformation, and all transformations are definitely stored on the underlying unary stack because we allow the value zero. Therefore, we do not need access to the input text or the index stack H anymore when popping elements from L. Thus, we expect faster stack operations than with $\delta = 1$.

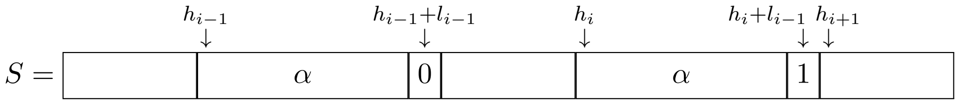
In Section 6.2.2 we claimed that in practice we could realize dynamic memory allocation for the stacks. In fact, we provide both static and dynamic implementations. The static version allocates n bits for H and $\lceil 4n/\delta \rceil$ bits for L prior algorithm execution (or $6n$ bits for L with $\delta = 0$). The underlying data structure is a custom bit vector implementation that is based on an array of unsigned integers. The dynamic version allocates almost no memory before the algorithm execution and uses two instances of `std::stack<uint64_t>` to dynamically grow and shrink both sides of the unary stack.

7.4.1 Artificial Instances

First, we show how the parameter δ influences the query time and memory usage. At the same time, we compare the difference in performance when using dynamic memory allocation instead of static memory allocation. As we have seen in Section 7.3, the stacks never grow very large for real texts. Therefore, in order to properly observe the influence of the parameter δ, we have to generate artificial test instances that put enough stress on the stacks. Since we can only pop elements of the stack L if both the index stack H as well as the input string S are given, we have to generate test instances that consist of a list $h_1, \ldots, h_{k+1}$ of indices, a list $l_1, \ldots, l_k$ of LCP values, and a string S that matches the indices and LCP values. The test instances must satisfy $\forall i \in [1, k] : l_i = \textsc{Lcp}_S(h_i, h_{i+1})$.

Generating Test Instances

Our generator expects the number k of LCP values that we want to generate, as well as a parameter $r \in \mathbb{N}^+$ that indicates the highest possible LCP value. First, we draw k uniformly distributed random LCP values $l_1, \ldots, l_k$ from the interval $[0, r)$. Let $h_1 = 1$, $h_2 = r + 1$, and $h_{i+1} = h_i + l_{i-1} + 1$ for $i \in [2, k]$.



Let S be a string of length $h_{k+1}+l_k = \Theta(\sum_{i=1}^{k} l_i)$. We fill the prefix $S[1..r]$ with uniformly distributed random characters from the binary alphabet $\{0, 1\}$. Then, we interpret the generated LCP values and indices as instructions to fill the rest of the string. For each h_i with $i \in [2, k + 1]$ we assign the substring $S[h_i..h_i + l_{i-1}) = S[h_{i-1}..h_{i-1} + l_{i-1})$, which enforces the shared prefix between $S_{h_{i-1}}$ and S_{h_i}. By assigning $S[h_i + l_{i-1}] = 1 - S[h_{i-1} + l_{i-1}]$, we also enforce the mismatch after the LCP.

Since we draw the LCP values uniformly at random, it is to be expected that roughly half of them are absolute values (as defined in Section 6.4).

Testing Methodology

Using the instance generator from the previous section, we generate a test instance of size $k = 2^{24}$ for all values $r \in \{8, 32, 128, 512\}$. For each instance we compare the following stack implementations, where we let δ assume all values from $\{0, 1, 2, 4, 8, 16, 32, 64\}$:

naive: Both H and L are represented by an instance of `std::stack`. All values are stored in plain binary representation (and the parameter δ does nothing).

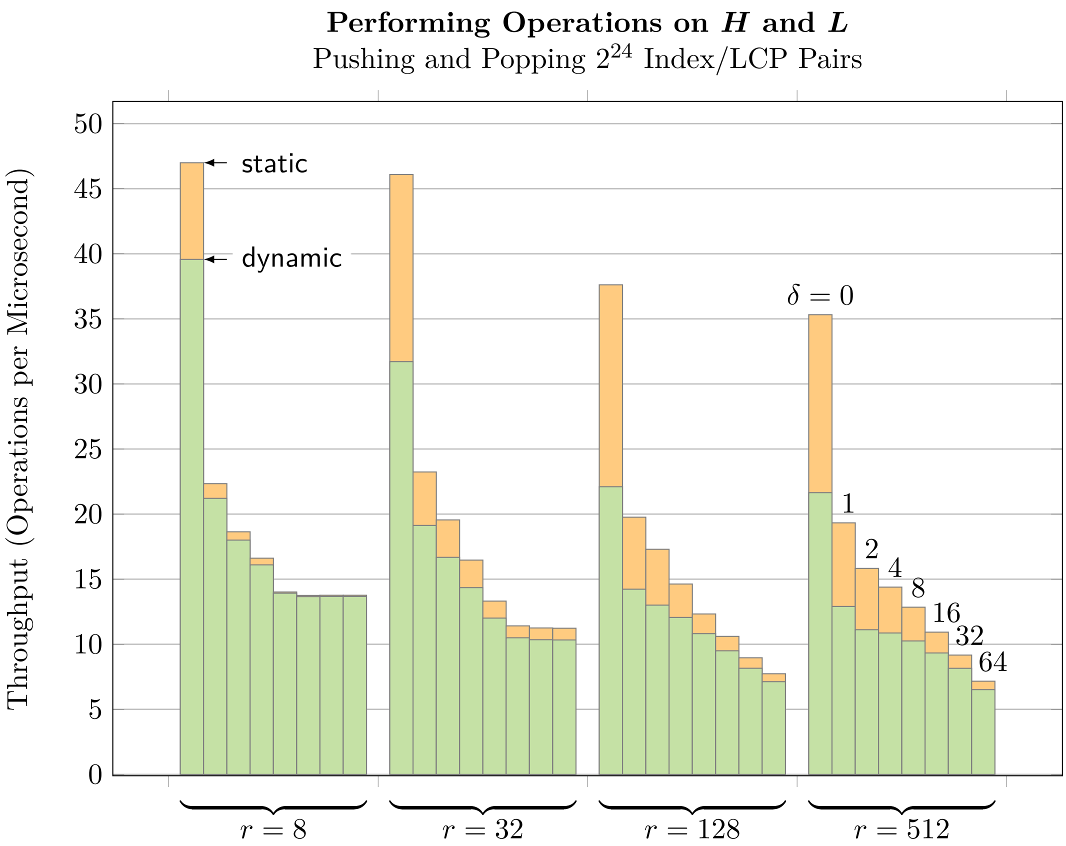


Figure 7.2: Throughput of the different stack implementations depending on δ and r (see Section 7.4.1 for details). Green bars represent dynamic stacks, while orange bars represent static ones. The bars are grouped by the value of r, and for each group they are sorted ascendingly by the value of δ.

static(δ): We use the telescope stack from Corollary 6.2.4 for H, and the succinct LCP stack from Lemma 6.4.1 with parameter δ for L. Both stacks use static memory allocation.

dynamic(δ): Same as static(δ), but both stacks use dynamic memory allocation.

For each combination of stack implementation and test instance we proceed as follows: We push all indices and LCP values of the instance onto the respective stacks, and then pop them again, measuring the combined total time for pushing and popping. Each experiment is repeated five times, of which we take the median as the final result. Using the final result we compute the average throughput in stack operations per microsecond (where one operation means either pushing an index/LCP pair, or popping one). For the dynamic(δ) implementation we also measure the memory usage of L after pushing all elements.

Results

Figure 7.2 shows the throughput of the different stacks grouped by the value of r. The diagram does not include the naive implementation, for which we measured around 82 operations per microsecond for all values of r. Choosing $\delta = 0$ yields the best results by a wide margin, reaching a throughput between 47 ($r = 8$) and 35 ($r = 512$) operations per microsecond in the static case, and between 39 ($r = 8$) and 21 ($r = 512$) operations per microsecond in the dynamic case. This is at most 75% slower than the naive stacks. We never have to sacrifice more than 41% of throughput when choosing the dynamic version over the static one. As expected, higher values of δ achieve less throughput. Particularly, we lose up to 52% of throughput when changing from $\delta = 0$ to $\delta = 1$, which shows that adding the type-flags and storing transformations with value zero is worth the additional effort.

For the smallest range $r = 8$ we can observe a *plateau* in the diagram: Choosing values of δ that are greater than eight does not decrease the throughput any further. This is not surprising, since for $\delta = 8$ and $r = 8$ the δ-transformation of all elements on L becomes zero, which essentially means that the underlying unary stack stays empty and we have to recompute all LCP values from scratch when popping elements. Since regardless of δ this recomputation always takes the same time, we do not become slower for increasing δ. The plateau becomes smaller for $r = 32$, and completely disappears for $r = 128$ and $r = 512$. Generally, the stacks perform better for smaller r, i.e. for smaller LCP values. Taking into account that the average LCP values of real texts are small (recall Table 7.1), this is a favorable property.

Next, we look at the memory usage of the dynamic LCP stack. On the naive stack we need either 32 bits or 64 bits per LCP value, depending on how many bits are required to address the input string. As seen in Figure 7.3, the dynamic implementation needs significantly less space for smaller values of r. Even for $\delta = 0$ we need only 4.2 and 12.2 bits per element for $r = 8$ and $r = 32$ respectively. Once again, considering the low LCP values in real texts, this is a strong result. In the worst case, we still need only 99.9 bits per element ($\delta = 0$, $r = 512$). As explained before, using $\delta = 0$ instead of $\delta = 1$ increases the number of bits per element exactly by two.

For all values of r, except for $r = 512$, the memory usage decreases linearly with increasing δ, i.e. multiplying δ by two cuts the memory usage in half. Once we start seeing more large elements ($r = 512$), the benefit of using larger values of δ becomes smaller.

Summarizing the results for artificial instances, the different stack implementations behave exactly as expected. Increasing δ reduces both throughput and memory usage. Using the dynamic implementation causes an acceptable amount of overhead.

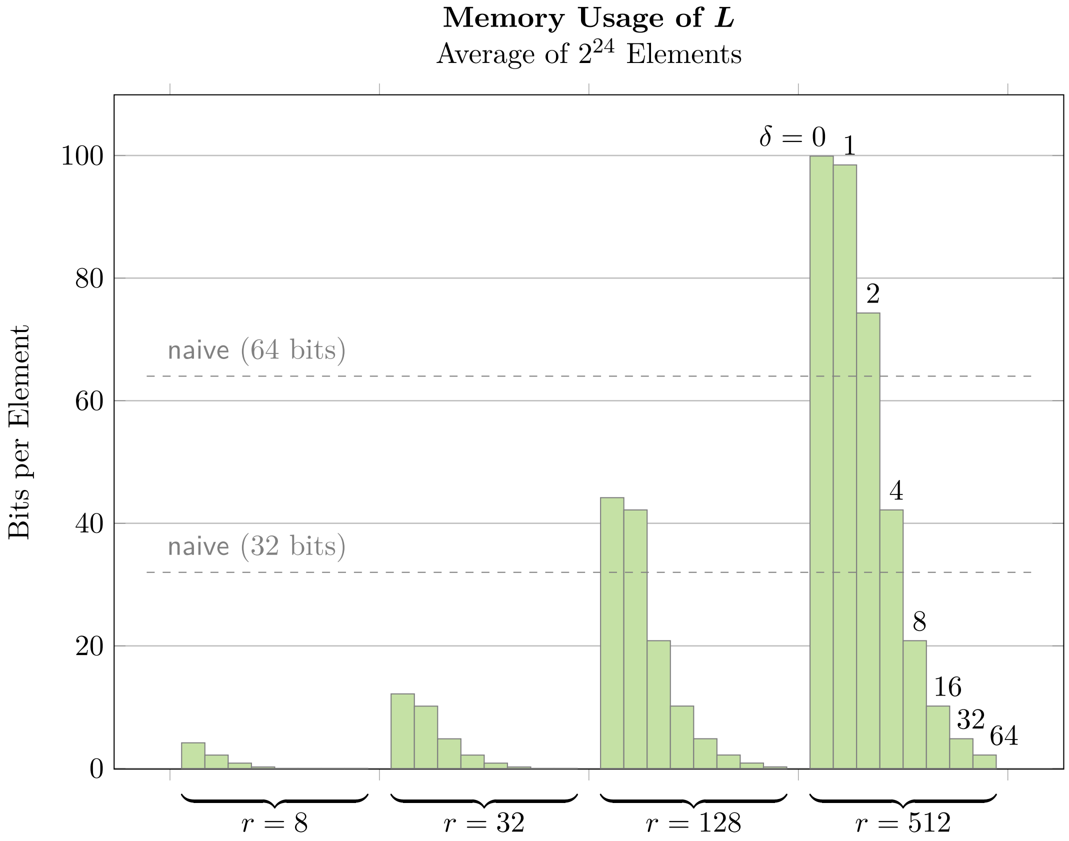


Figure 7.3: Memory usage of the different implementations of the stack L depending on δ and r (see Section 7.4.1 for details). The results are grouped by the value of r, and for each group the bars are sorted ascendingly by the value of δ.

7.4.2 Real Texts

Next, we analyze how the stacks perform in a more realistic setting. As observed before, the texts of our collection do not need many elements on the stacks. This motivates us to introduce one additional version of our stacks:

dynamic-buffered(δ): Same as dynamic(δ), but uses an additional dynamically sized buffer of at most n bits. The buffer contains the topmost elements on H and L in plain binary representation, which allows very fast operations. Only once the buffer is completely full, we take the bottommost half of the buffered elements and move them onto a dynamic(δ) stack. When the buffer becomes completely empty, we fill half of it with the topmost elements from the dynamic(δ) stack. We expect that on real texts the buffer never becomes full. Thus, we should observe a high throughput.

Testing Methodology

In order to compare the practical performance of the stacks, we plug each implementation into xss-real and measure the throughput that we achieve while computing the BPS of the PSS tree. For this benchmark we use the text `xml`, which turned out to achieve the highest overall throughput, as well as the text `leaders`, which turned out to achieve the lowest overall throughput. Additionally, we consider `pc-avg`, which is the average of the results for all texts from Table 7.1 (where each text weighs in equally, regardless of text length). For each text combined with each stack implementation we run xss-real five times, measuring the total execution time and taking the median as the final result. From the execution time we compute the average throughput in MiB/s. Also, we measure the maximum additional memory usage, which is the highest memory usage that we observed at any point during the algorithm execution, minus the space needed for input and output. All of our texts are stored using one byte per character, while the BPS has size $2n$ bits. Thus, input and output need a total of $10n$ bits of memory.

Results

Figure 7.4 shows how the different stack implementation compare. We see the throughput in MiB/s as well as the additional memory usage in n bits. As expected, the static version needs almost exactly $7n$ bits of additional memory for $\delta = 0$, and $4n/\delta + n$ bits for all other values of δ. This holds for the texts `xml` and `leaders`, and also for all other texts of Table 7.1. Conveniently, the naive, dynamic and dynamic-buffered stacks allow xss-real to run without significant memory overhead. In fact, there is not a single text in our collection for which we need more than $n/70$ additional bits of memory. If the text uses one byte per character, then this is equivalent to around 0.16% of the input size, which is negligible in practice.

In terms of throughput, the naive implementation is the fastest, achieving around 50 MiB/s for `xml`, around 32 MiB/s for `leaders`, and an average of almost 45 MiB/s for `pc-all`. We achieve similar results with the dynamic-buffered stacks, which are less than 15% slower than the naive implementation for the text `xml`, less than 10% slower for `leaders`, and less than 12% slower for `pc-avg`. As seen in all plots, the throughput achieved with the dynamic-buffered stacks does not decrease with growing δ, indicating that the buffer never becomes full.

The static and (not buffered) dynamic implementations typically reach between 30 and 55% of the naive throughput, but never less than 60% for $\delta = 0$. Compared to the artificial test instances, the performance gap between static and dynamic is much smaller. Choosing

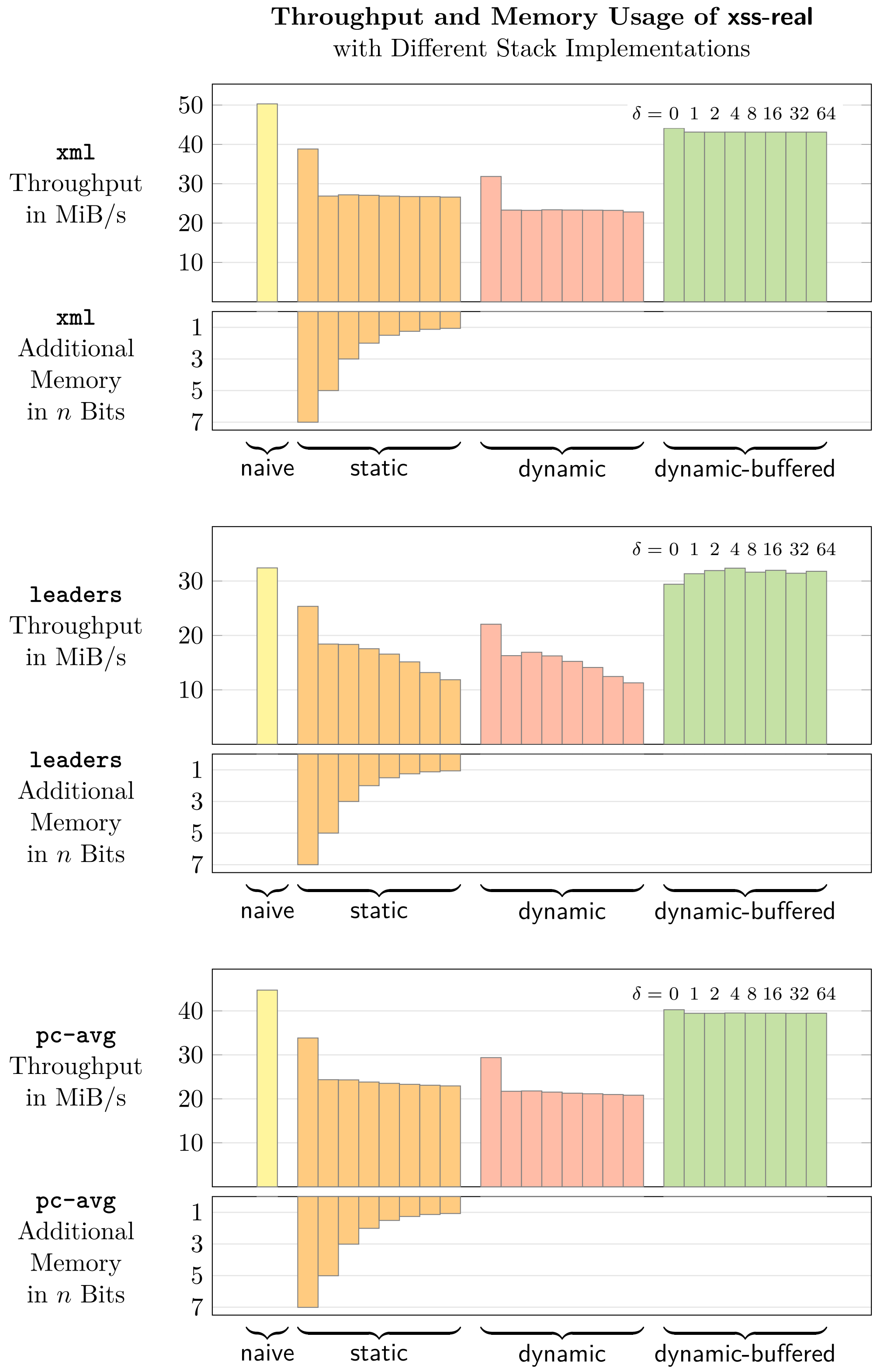


Figure 7.4: Throughput and memory usage of xss-real with different stack implementations and $\delta \in \{0, 1, 2, 4, 8, 16, 32, 64\}$. The results are grouped by the stack types. For each type, the bars are ordered ascendingly by the value of the parameter δ.

dynamic memory allocation over static memory allocation usually decreases the throughput by around 10%.

For xml, the parameter δ does not influence the throughput, which is not surprising because the average LCP is only 0.25. On the other hand, high values of δ cause lower throughputs for leaders because the average LCP of 9.93 is relatively high.

Summarizing the results for real instances, it seems like the naive implementation is the best choice for our text collection. It clearly achieves the highest throughput while imposing no significant memory overhead. However, an important aspect of xss-real is its guaranteed worst case memory bound. Therefore, we will only use dynamic-buffered(0), dynamic-buffered(4), dynamic(0), as well as dynamic(4) during the rest of the evaluation, which offer compelling worst-case memory bounds and a high throughput. While the parameter δ offers an interesting theoretical trade-off between memory requirement and execution time, the low LCP values of real texts make the choice of δ almost irrelevant in practice (especially when using the dynamic-buffered stacks).

7.5 Comparison Against Existing Algorithms

In this section, we compare the performance of a variety of Lyndon array construction algorithms. We consider the following approaches:

Algorithms from Chapter 4 and Chapter 5. We combine xss-bps, xss-bps-lcp, and xss-real with the dynamic-buffered(0), dynamic-buffered(4), dynamic(0), and dynamic(4) stacks. The result of these algorithms is the BPS of the PSS tree, while all other algorithms compute the actual Lyndon array or the NSS array. In order to allow a fair comparison, we also construct the support data structure for fast queries. This way, all algorithms produce a representation of the Lyndon array that allows constant time access. For the support data structure we use an implementation of Sadakane and Navarro's range min-max tree [Sadakane and Navarro, 2010] that is provided in the *Succinct Data Structure Library (SDSL)* [Gog et al., 2014][8].

Algorithms from the SDSL. As mentioned in Chapter 4, the algorithm xss-array is essentially identical to the PSV & NSV algorithms that are implemented in the first version of the SDSL[9], but with naive suffix comparisons instead of element comparisons. We consider the following modifications of the SDSL NSV algorithm:

- sdsl-naive: Uses naive suffix comparisons instead of element comparisons.

[8] https://github.com/simongog/sdsl-lite/blob/master/include/sdsl/bp_support_sada.hpp

[9] https://github.com/simongog/sdsl/blob/master/include/sdsl/algorithms.hpp, see functions calculate_psv and calculate_nsv

- sdsl-prezza: Achieves $\mathcal{O}(\lg n)$ time suffix comparisons by using a probabilistic LCE data structure that is based on Karp-Rabin fingerprinting [Policriti and Prezza, 2016]. Since this data structure is *in-place*, it requires no additional memory. We utilize an implementation by Nicola Prezza[10]. The version sdsl-1k-prezza realizes suffix comparisons by first naively comparing up to 1000 characters directly in the input text, and then only deploying the LCE data structure, if there was no mismatch in the first 1000 characters. Note that for sdsl-1k-prezza we keep both the LCE data structure *and* the input text in memory.

- sdsl-herlez: Similar to sdsl-prezza, but uses the LCE data structure presented in [Prezza, 2018]. We utilize an implementation by Alexander Herlez[11]. Analogous to sdsl-1k-prezza, there is a version sdsl-1k-herlez that performs up to 1000 naive character comparisons per suffix comparison.

- sdsl-isa-nsv: Builds the suffix array using the algorithm DivSufSort[12], then constructs the inverse suffix array, and finally runs the SDSL NSV algorithm on the inverse suffix array (see [Franek et al., 2016, Algorithm NSVISA]). Even though DivSufSort has no linear worst-case time bound, it appears to be the fastest suffix sorting algorithm in practice[Fischer and Kurpicz, 2017][13].

All of these algorithms produce the NSS array. At this point it shall be mentioned, that we could also equip the SDSL NSV algorithm with a data structure that allows constant time LCE queries (and thus constant time suffix comparisons). However, these data structures usually depend on the inverse suffix array (see for example [Fischer and Heun, 2006]). Instead of computing the inverse suffix array *and* the LCE data structure, it is faster to directly run sdsl-isa-nsv (which only requires the inverse suffix array).

Baier's Algorithm. As discussed in Section 1.2.3, Baier's algorithm is the only linear time Lyndon array construction algorithm that does not require the suffix array as a prerequisite, but computes the Lyndon array as a prerequisite of the suffix array. In the following, g-saca is Baier's original algorithm that computes the suffix array[14], and g-saca-lyn is a modified version of the first phase of Baier's algorithm that only computes the Lyndon array[15].

Non-Elementary Algorithms. All other linear time Lyndon array construction algorithms depend on the suffix array. Therefore, an inherent lower bound for their execution time is given by the fastest suffix sorting algorithm. Also, at some point they need to

[10] https://github.com/nicolaprezza/rk-lce
[11] https://github.com/herlez/lce-test
[12] https://github.com/y-256/libdivsufsort
[13] At the time of writing (July 2019)
[14] https://github.com/waYne1337/gsaca
[15] https://github.com/felipelouza/lyndon-array/tree/master/external/gsaca_cl

keep both the Lyndon array *and* the suffix array in memory. Assuming that the arrays use a 32-bit integer per entry, this means that the maximum memory usage is at least $64n$ bits. In the following experiments we represent the non-elementary algorithms with a single entry non-ele-lyn, for which we use the throughput of DivSufSort, as well as a fixed amount of $64n$ bits of memory. Clearly, using non-ele-lyn as a representative is in favor of the non-elementary algorithms.

7.5.1 Testing Methodology

For this benchmark, we use the text `english.1G`, which offers the lowest average LCP value of only 0.2, as well as the text `cere`, which offers the highest average LCP value of 156.24. Additionally, we consider `pc-avg`, which is the average of the results for all texts from Table 7.1 (where each text weighs in equally, regardless of text length). For each text we run each algorithm five times, measuring the total execution time and taking the median as the final result. From the execution time we compute the average throughput in MiB/s. Also, we measure the maximum additional memory usage, which is the highest memory usage that we observe at any point during the algorithm execution, minus the space needed for input and output. All of our texts are stored using one byte per character. Thus, the algorithms that compute the BPS of the PSS tree use $10n$ bits of memory for input and output. All other algorithms compute the Lyndon array or the NSS array in 32-bit integer format, i.e. they use $40n$ bits for input and output.

7.5.2 Results

Figure 7.5 shows how the different algorithms compare. First, we only look at the memory usage. In general, all of the considered algorithms use almost exactly the same amount of additional memory regardless of the text that we are looking at. In Section 7.4, we already observed that the additional memory usage of xss-bps, xss-bps-lcp, and xss-real is close to zero for all texts. This also holds for sdsl-naive, as well as the two in-place LCE variants sdsl-herlez and sdsl-prezza. If we store the LCE data structures separately, i.e. not in-place, then up to $8n$ bits of additional memory are needed for sdsl-1k-herlez and sdsl-1k-prezza. The non-elementary approach sdsl-isa-nsv needs around $24n$ bits of additional memory. This was to be expected, because the peak memory usage occurs when both the inverse suffix array and the Lyndon array are kept in memory at the same time, which requires $64n$ bits. As explained earlier, we also have a total memory usage of $64n$ bits for non-ele-lyn, which equals $24n$ bits of additional memory usage. Baier's algorithm g-saca (and its modification g-saca-lyn) has the by far highest additional memory usage of $96n$ bits.

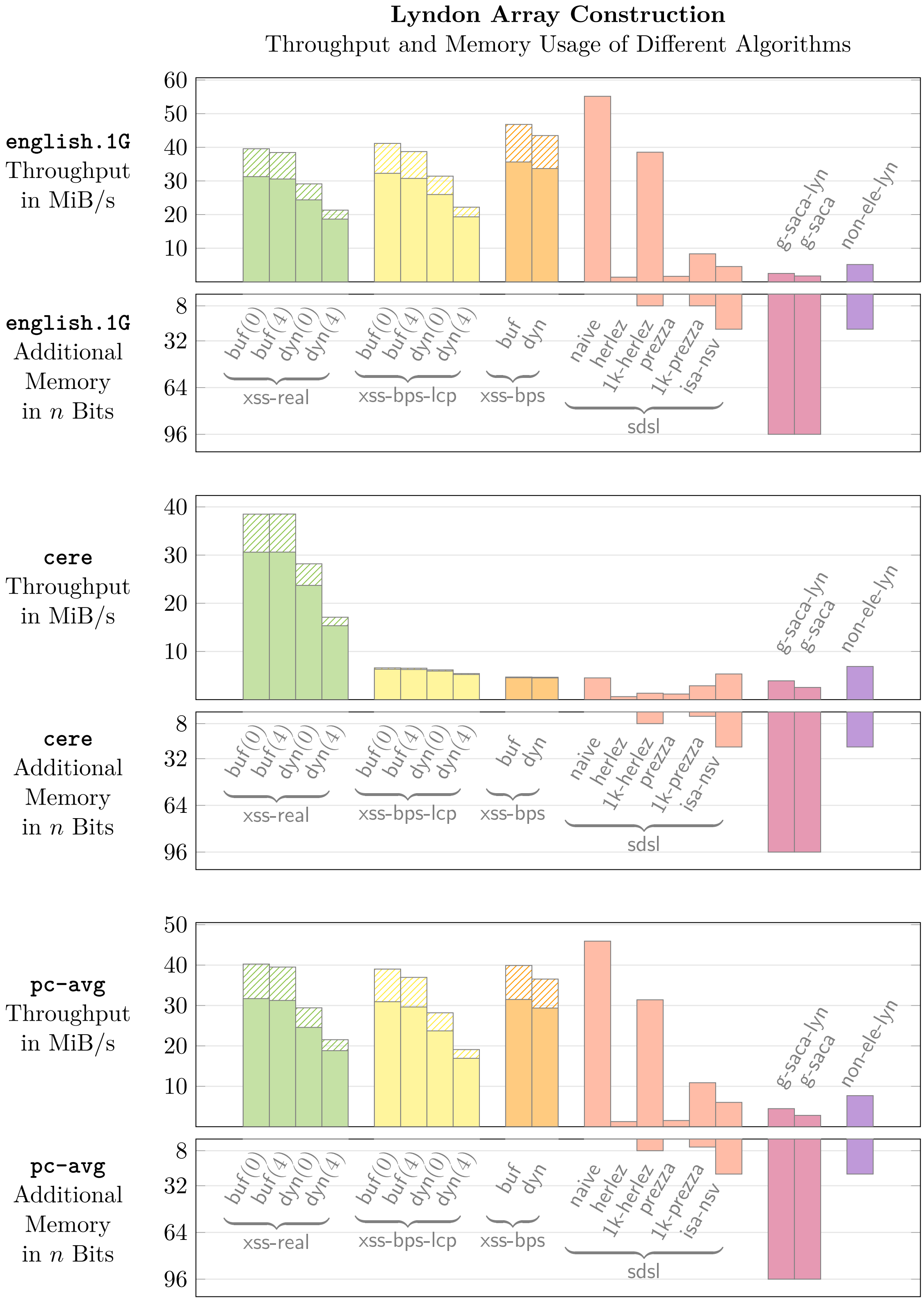


Figure 7.5: Throughput and memory usage of different Lyndon array construction algorithms. The hatched bars for xss-real, xss-bps-lcp, and xss-bps indicate the throughput without computing the support data structure, while the solid bars indicate the throughput including the construction of the support data structure.

Next, we look at the throughput of the elementary algorithms for `english.1G` and `pc-avg`. In general, the results are very similar, indicating that `english.1G` is a good representative for the text collection at hand. Interestingly, there is only a slight difference in throughput between xss-bps-lcp and xss-real (assuming that both algorithms use the same stack implementations). The dynamic-buffered stacks with $\delta = 0$ are the fastest option, reaching over 30 MiB/s of throughput for both xss-bps-lcp and xss-real. If we choose dynamic stacks with $\delta = 4$ instead, then the throughput decreases by around 40%, which is still more than 18 MiB/s. The naive algorithm xss-bps achieves roughly the same throughput as xss-real and xss-bps-lcp for `pc-avg`, and even a slightly higher throughput for `english.1G`. This shows, that maintaining the LCP stack L is not worth the additional effort, if the LCP values are low. The fastest algorithm for both `english.1G` and `pc-avg` is sdsl-naive with around 55 and 46 MiB/s respectively. It also outperforms the variants sdsl-herlez, sdsl-1k-herlez, sdsl-prezza, and sdsl-1k-prezza on all texts, showing that the LCE data structures at hand are not practical for the specific algorithmic setting.

If we look at the text `cere`, then the picture changes. The throughput of all algorithms, except for xss-real, drops below 7 MiB/s. In contrast, xss-real performs almost as well as for `english.1G` and `pc-avg`, still reaching over 30 MiB/s when using dynamic-buffered stacks with $\delta = 0$, and reaching over 15 MiB/s when using dynamic stacks with $\delta = 4$.

Finally, we consider Baier's algorithm and the non-elementary algorithms. For `english.1G`, `cere`, and `pc-avg`, all of these algorithms achieve less than 8 MiB/s of throughput. In the average case, sdsl-isa-nsv is only 22% slower than non-ele-lyn, which indicates that once the suffix array is given, it is not much additional effort to compute the Lyndon array. Both g-saca and g-saca-lyn are slower than sdsl-isa-nsv and non-ele-lyn.

On average, xss-real with dynamic-buffered stacks and $\delta = 0$ achieves four times more throughput than non-ele-lyn. The biggest difference in performance occurs for the text `rs.13`, where xss-real is over eight times faster than non-ele-lyn. On the other hand, the smallest difference in performance occurs for the text `leaders`, where xss-real is still over 33% faster than non-ele-lyn. This is also the text for which xss-real achieves its lowest throughput of around 24.5 MiB/s, while non-ele-lyn achieves its highest throughput of around 18.5 MiB/s. The results for all texts (including the ones that are not displayed in Figure 7.5) can be found in Appendix B.

7.5.3 Conclusion (Comparison Against Other Algorithms)

The algorithm xss-real with dynamic-buffered stacks and $\delta = 0$ performs very well in practice. On all texts of Table 7.1, it is both significantly faster and more memory efficient than the non-elementary algorithms. On average, it achieves four times more throughput while using almost no additional memory. For texts with small LCP values, the algorithm sdsl-naive is the fastest option.

7.6 Scalability

7.6.1 Repetitive Artificial Texts

In this section, we present some additional benchmarks that show the scalability of xss-real. First, we use some highly repetitive artificial texts that are built on Lyndon runs. There are three instance types that we consider:

- run-p1: Increasing Lyndon runs with period one, i.e. $\mathtt{a}^{n-1}\mathtt{z}$. Increasing runs are harder to process than decreasing ones, because we have to push all starting positions of repetitions onto H (see Section 5.1.2). Since the period of this particular run is one, we have to push almost n indices.
- run-p10: Increasing Lyndon runs with period ten, i.e. $(\mathtt{abcdefghij})^{n/10}$.
- run-rec-r10: Recursive Lyndon runs of ten repetitions. For example, the instance of length $n = 11110$ has the form $(\mathtt{a}(\mathtt{b}(\mathtt{cd}^{10})^{10})^{10})^{10}$. In order to obtain instances of arbitrary length, we may pad the string with additional repetitions.

These strings are of particular interest, because they yield very high artificial LCP values. In fact, even for only a few megabytes of input, it is practically impossible to compute the Lyndon array for run-p1 and run-p2 using xss-bps-lcp, xss-bps, or sdsl-naive. Also, the algorithms sdsl-herlez and sdsl-prezza achieve very low throughputs (in the order of KiB/s rather than MiB/s). Therefore, we do not even attempt to compare xss-real with the other algorithms, but only show how the performance of xss-real scales with the input size. Ideally, we want the same throughput for different sizes of the same instance type.

Testing Methodology

In terms of stack implementations, we equip xss-real with dynamic-buffered(0), dynamic-buffered(4), dynamic(0), and dynamic(4) stacks. As before, we repeat each experiment five times and take the median as the final result. The considered input sizes per instance are 128, 256, 512, 1024, 2048, 4096 and 8192 MiB (using one byte per character).

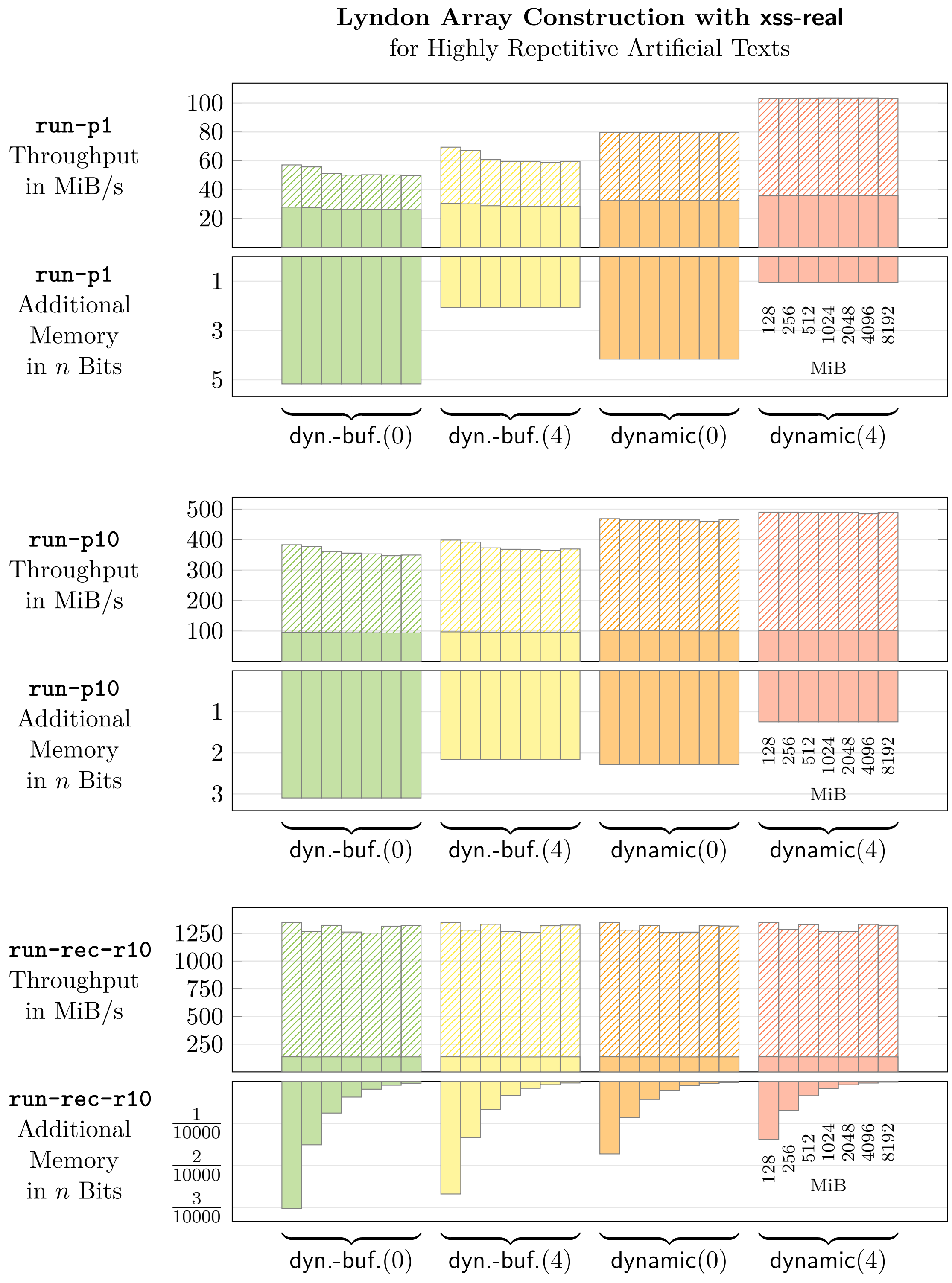


Figure 7.6: Throughput and memory usage of xss-real for different input sizes of highly repetitive artificial texts. Hatched bars indicate the throughput without computing the support data structure, while solid bars indicate the throughput including the computation of the support data structure. The results are grouped by stack implementation. For each of the stack implementations, the bars are order ascendingly by the input size. We consider the input sizes 128, 256, 512, 1024, 2048, 4096, and 8192 MiB.

Results

Figure 7.6 displays the throughput and additional memory usage of xss-real for the three instance types. In the following, we only consider the throughput *without* computing the support data structure (i.e. the hatched bars in the plot). This way, we can more accurately assess the performance differences between different stack implementations, instance types, and instance sizes.

First, we only focus on the results for `run-rec-r10`. The throughput for different input sizes varies slightly, but there appears to be no correlation with the input size. Regardless of the stack implementation, we always reach a very high throughput of over 1.25 GiB/s, which demonstrates the efficiency of the run extension. In terms of memory usage, we need at most 0.0003n additional bits apart from input and output, which is insignificant in practice. Doubling the input size cuts the *relative* additional memory usage in half. Therefore, the *absolute* additional memory usage is constant regardless of the input size.

Next, we focus on `run-p1` and `run-p10`. If we look at any of the stack implementations, xss-real scales very well, achieving almost the same throughput for all input sizes. For the dynamic-buffered stacks there is a slightly higher throughput for smaller instances. However, even if we change from 128 MiB to 8 GiB of input, the throughput decreases by at most 15%. In terms of memory usage, each stack implementation scales perfectly, using the same relative amount of additional memory for all input sizes. Due to the high number of elements that we have to push onto the stacks (roughly n elements for `run-p1` and roughly $n/10$ elements for `run-p10`), the buffer of the dynamic-buffered implementation becomes full, and we have to move elements from the buffer to a dynamic stack. Therefore, it is not surprising that the throughput for dynamic is higher than for dynamic-buffered. The largest throughput difference occurs for $\delta = 4$ and instance type `run-p1`, where changing from dynamic-buffered to dynamic increases the throughput by around 70%.

Finally, we discuss some interesting aspects that are not related to scaling, but are still worth mentioning. First of all, we can observe a significant amount of additional memory usage for `run-p1` and `run-p10`. The highest usage occurs for `run-p1` with dynamic-buffered stacks and $\delta = 0$, which causes an additional memory usage of $5.16n$ bits, which is relatively close to the theoretical maximum of $8n$ bits (n bits for H, $6n$ bits for L, n bits for the buffer). Without the buffer, we need almost exactly n bits less. Changing from $\delta = 0$ to $\delta = 4$ reduces the memory usage by slightly over $3n$ bits for both the dynamic-buffered and the dynamic stacks. Similar results can be observed for the `run-p10` instances.

The arguably most peculiar result of the experiments is, that choosing $\delta = 4$ over $\delta = 0$ *increases* the throughput for `run-p1` and `run-p10` instances. This is surprising, because in all previous benchmarks we observed less throughput for higher values of δ. The exact cause of this irregularity is unknown (due to time constraints it was not possible to find a conclusive answer).

7.6.2 Real Texts

Lastly, we also take a look at the scalability of xss-real for large real texts. We only consider dynamic-buffered stacks with $\delta = 0$, which is the most practical configuration for real texts. The test instances at hand are:

- `dna-large`: Our test instance uses a variety of FASTQ files[16] from the *1000 Genomes* project[17] that were cleaned such that the effective alphabet size is four, i.e. our test instance contains only the characters `ACGT`. The size of the final instance is around 210 GiB.
- `proteins-large`: The *Universal Protein Resource (UniProt)*[18] offers a large collection of protein sequences. Our test instance consists of two files[19], from which all non-sequence information like whitespaces was removed. The resulting instance has a size of around 47 GiB.
- `cc-large`: *Common Crawl*[20] is a non profit organization that provides a text corpus obtained by crawling the internet. Our test instance was generated from a collection of multiple files[21]. The WARC meta information was removed[22] and the files were concatenated to a single text of around 184 GiB.
- `wiki-large`: The *Wikipedia* is a popular online encyclopedia. Our test instance consists of the textual information of all Wikipedia pages in German, English, Spanish, and French, where the content was fetched on March 3, 2019[23]. The resulting instance has a size of around 230 GiB.

[16] ftp://ftp.sra.ebi.ac.uk/vol1/fastq/DRR000/DRR`#ID`, where `#ID` is in the range from 000001 to 000426_1

[17] http://www.internationalgenome.org/

[18] https://www.uniprot.org/

[19] ftp://ftp.uniprot.org/pub/databases/uniprot/current_release/knowledgebase/complete uniprot_`#ID`.dat.gz, where `#ID` is either *sprot* or *trembl*

[20] http://commoncrawl.org

[21] crawl-data/CC-MAIN-2019-09/segments/1550247479101.30/wet/CC-MAIN-20190215183319-20190215205319-`#ID`.warc.wet, where `#ID` assumes all values in the range from 00000 to 000600

[22] https://iipc.github.io/warc-specifications/specifications/warc-format/warc-1.1/

[23] https://dumps.wikimedia.org/`#ID`wiki/20190320/`#ID`wiki-20190320-pages-meta-current.xml.bz2, where `#ID` is *de*, *en*, *es*, or *fr*

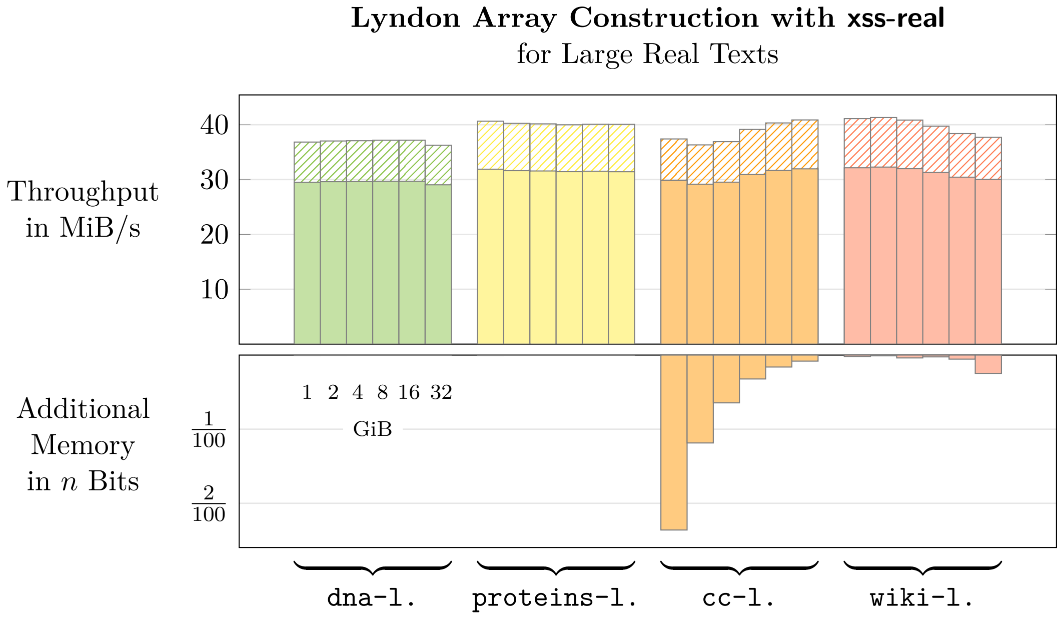


Figure 7.7: Throughput and memory usage of xss-real with dynamic-buffered stacks and $\delta = 0$ for large real texts. The results are grouped by text. For each text, we consider the prefixes of size 1, 2, 4, 8, 16, and 32 GiB. The bars are ordered ascendingly by prefix size.

Testing Methodology

We consider prefixes of size 1, 2, 4, 8, 16, and 32 GiB for each input text. For each text and prefix size, we run xss-real five times, taking the median time as the final result. As before, we compute the throughput in MiB/s and the additional memory in n bits.

Results

Figure 7.7 shows how xss-real scales for the different input sizes. We need less than $0.025n$ bits of memory for all texts and input sizes. In general, xss-real achieves a throughput between 36 and 41 MiB/s (without the computation of the support data structure). This equals around 30 MiB/s including the computation of the support data structure, which matches our previous observations for small real texts. There appears to be no significant correlation between the input size and the throughput for `dna-large` and `proteins-large`. In contrast, we see a positive correlation for `cc-large`, where larger input sizes achieve a higher throughput. However, doubling the input size increases the throughput by at most 6%. The opposite can be observed for `wiki-large`, where we lose some throughput when increasing input sizes. However, we never lose more than 3.5% of throughput when doubling the input size.

7.6.3 Conclusion (Scalability)

In general, xss-real scales very well with the input size of both repetitive artificial and real texts. The throughput varies only slightly, even when significantly changing the input size. At this point it is also worth mentioning, that when processing real texts of size 32 GiB, the total memory usage *including* input and output never exceeded 40.05 GiB. This demonstrates the practical advantage of the succinct Lyndon array representation for large inputs. The actual Lyndon array for the same input size needs $\lg(32 \cdot 2^{30}) = 35$ bits per entry, which equals around 150 GiB of memory. The total memory usage of a non-elementary algorithm that uses the suffix array would have been at least 300 GiB.

Chapter 8

Conclusion

We presented the first elementary algorithm that computes a representation of the Lyndon array in linear time. Representing the Lyndon array as a PSS tree allows us to store it in $2n+o(n)$ bits, which is asymptotically optimal. Despite the compact size, we can simulate constant time access to the Lyndon array, and even answer more sophisticated queries like range minimum suffix queries in constant time. This is another advantage over the naive representation of the Lyndon array, which cannot answer these queries efficiently.

In practice, the new construction algorithm is not only over four times faster than the existing non-elementary algorithms, but also needs significantly less memory. On most real texts, the additional memory usage is less than 1% of the input size. In terms of theoretical bounds, the algorithm uses at most $\lceil 4n/\delta \rceil + \mathcal{O}(\sqrt{n} \lg n)$ bits of memory and runs in $\mathcal{O}(\delta n)$ time, where δ is a freely choosable parameter. This is a big improvement over the non-elementary algorithms, for which the space requirements are usually expressed in multiples of n bytes, rather than in fractions of n bits.

8.0.1 Future Work

Even though the new construction algorithm yields compelling practical results, it is complicated in theoretical detail. The existence of a linear time elementary algorithm gives reason to believe, that a simpler solution is possible. Also, it would be interesting to see a version of xss-real that computes the actual Lyndon array instead of the BPS of the PSS tree. The next logical step would be using xss-real to accelerate the first phase of Baier's suffix sorting algorithm. Since the phase computes a partially sorted version of the Lyndon array, we would need to find an efficient way of establishing this order either on the PSS tree or on the (non-sorted) Lyndon array.

Another interesting question is, if there are efficient Lyndon array construction algorithms in other models of computation, e.g. in the external memory model or in the area of parallel computing. Particularly, since there already exist optimal parallel algorithms for the highly related NSV and PSV problems [Berkman et al., 1993], it seems likely that the construction of the NSS and PSS arrays is efficiently parallelizable.

Lastly, there are not yet many practical applications that are known for the Lyndon array. With faster construction algorithms, it might become more feasible as a prerequisite for other algorithms and data structures. It is an open question, to which extend the Lyndon array can be used as a tool in classical string processing fields like text indexing and compression. Hopefully, it will become equally useful and important as the suffix array.

Appendix A

Detecting Extended Lyndon Runs

In this section, we explain how to efficiently detect if a string is an extended Lyndon according to Definition 5.2.1. Recall, that such a string has the form $S = \mathrm{suf}(\mu) \cdot \mu^t \cdot \mathrm{pre}(\mu)$ with $t \geq 2$, where μ is a Lyndon word, and $\mathrm{suf}(\mu)$ and $\mathrm{pre}(\mu)$ are a proper suffix and prefix of μ respectively.

$$S = \boxed{\mathrm{suf}(\mu)}\underbrace{\boxed{\mu}\boxed{\mu}\boxed{\cdots}\boxed{\mu}\boxed{\mu}}_{t\text{ times}}\boxed{\mathrm{pre}(\mu)}$$

We use a simple modification of [Duval, 1983, Algorithm 2.1] to realize the detection mechanism. Originally, Duval's algorithm computes the uniquely defined *Lyndon factorization* (sometimes called *Lyndon decomposition*) of a string:

A.0.1 Lemma (Chen et al., 1958). *Let S be a non-empty string. There exists a decomposition of S into non-empty factors $s_1, s_2, \ldots, s_m$ such that all of the following conditions hold:*

1. $S = s_1 \cdot s_2 \cdot \ldots \cdot s_m$
2. $\forall i \in [1, m] : s_i$ *is a Lyndon word*
3. $\forall i \in [2, m] : s_{i-1} \geq_{\mathrm{lex}} s_i$

There is exactly one such factorization for each string.

Now we show, that the longest factor in the Lyndon factorization of an extended Lyndon run is exactly the repeating Lyndon word μ of the run. This makes it easy to detect if a string is an extended Lyndon run: We can simply compute the Lyndon factorization and determine the length of the longest factor. After that, a trivial postprocessing is sufficient to determine if the string actually is an extended Lyndon run.

A.0.2 Lemma. *Let $S = \mathrm{suf}(\mu) \cdot \mu^t \cdot \mathrm{pre}(\mu)$ be an extended Lyndon run. Let $x_1, \dots, x_{k_1}$ be the Lyndon factorization of* $\mathrm{suf}(\mu)$*, and let $y_1, \dots, y_{k_2}$ be the Lyndon factorization of* $\mathrm{pre}(\mu)$*. Then the Lyndon factorization of S is given by:*

$$S = x_1 \cdot \ \dots \ \cdot x_{k_1} \ \cdot \ \underbrace{\mu \cdot \mu \cdot \ \dots \ \cdot \mu \cdot \mu}_{t \text{ times}} \ \cdot \ y_1 \cdot \ \dots \ \cdot y_{k_2}$$

Proof. Clearly, the first two conditions of Lemma A.0.1 are satisfied. We only have to prove the third one. Since we defined $x_1, \dots, x_{k_1}$ and $y_1, \dots, y_{k_2}$ to be the Lyndon factorizations of $\mathrm{suf}(\mu)$ and $\mathrm{pre}(\mu)$ respectively, we already know that $\forall i \in [2, k_1] : x_{i-1} \geq_{\mathrm{lex}} x_i$ and $\forall i \in [2, k_2] : y_{i-1} \geq_{\mathrm{lex}} y_i$ hold. Also, we trivially have $\mu \geq \mu$. Therefore, in order to prove that the third condition of Lemma A.0.1 is satisfied, we only have to show that $x_{k_1} \geq_{\mathrm{lex}} \mu \geq_{\mathrm{lex}} y_1$ 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 from Lemma 2.1.7 that $x_{k_1} >_{\mathrm{lex}} \mu$ 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 $\mu >_{\mathrm{lex}} y_1$ holds. Therefore, the third property of Lemma A.0.1 is satisfied. □

If we look at the factors x_i and y_i of the factorization in the lemma, then each one of them is shorter than μ, which means that μ is the longest factor of the factorization. Next, we explain how to exploit this property to detect if a string is an extended Lyndon run.

A.0.1 Algorithmic Approach

We start by taking a closer look at Duval's algorithm. Let S be a string with Lyndon factorization $s_1, \dots, s_m$. In Duval's original version of the algorithm, each factor is represented by its end position, i.e. the algorithm outputs a list $d_1, \dots, d_m$ of indices with $\forall i \in [1, m] : d_i = \sum_{j=1}^{i} |s_j|$. Our algorithm for the detection of extended Lyndon runs uses this list as a prerequisite. Pseudocode is provided in Algorithm A.1.

Given any string that is not a Lyndon run, the algorithm outputs $\perp$. If however an extended Lyndon run is given, the algorithm outputs the period $|\mu|$ of the run, as well as the starting position $|\mathrm{suf}(\mu)| + 1$ of the first full repetition of μ. The algorithmic approach is simple: First, we only try to find the longest factor and its starting position. This can be achieved by processing the indices $d_1, \dots, d_m$ from left to right. Clearly, the length of factor s_i is exactly $d_i - d_{i-1}$ (if we define $d_0 = 0$). The starting position of factor s_i is $d_{i-1} + 1$. Initially, the longest known factor is s_1 with length $l = d_1$ and starting position $z = 1$ (lines 2–3). Then, we look at one factor at a time (line 4) and update the values of l and z, whenever we find a factor that is longer than all previous ones (lines 5–7). After the last iteration of the loop, we know the length and starting position of the longest factor. Note that if the given string actually is an extended Lyndon run, then the starting position z belongs to the *first* occurrence of μ. This holds, because we process the factors in left-to-right order, and we specifically do *not* update z when finding a factor of equal length.

Next, we have to verify if the computed values of l and z belong to an extended Lyndon run. Since such a run must have at least two repetitions, the period cannot be larger than $\lfloor n/2 \rfloor$. Therefore, we first check if $2l > n$ holds, and return $\perp$ if that is the case (lines 8–9). Otherwise, we perform a single scan over S and check for each character if it equals the character that is located l positions before (lines 10–11). If we find a mismatch, then we return $\perp$ (line 12). Otherwise, the string is an extended Lyndon run and we return l and z (line 13).

Algorithm A.1 Detection of Extended Lyndon Runs

Input: A string S of length n

Output: If S is an extended Lyndon run: Period $|\mu|$ and suffix length $|\mathrm{suf}(\mu)|$.
If S is not an extended Lyndon run: $\perp$.

```
 1: $d_1, \dots, d_m \leftarrow$ end positions of all factors of $S$
 2: $l \leftarrow d_1$
 3: $z \leftarrow 1$
 4: for $i \in [2, m]$ in ascending order do
 5:     if $d_i - d_{i-1} > l$ then
 6:         $l \leftarrow d_i - d_{i-1}$
 7:         $z \leftarrow d_{i-1} + 1$
 8: if $2l > n$ then
 9:     return $\perp$
10: for $i \in [l + 1, n]$ do
11:     if $S[i - l] \neq S[i]$ then
12:         return $\perp$
13: return $l, z$
```

A.0.3 Lemma. *Algorithm A.1 detects if a string of length n is an extended Lyndon run in $\mathcal{O}(n)$ time using $\mathcal{O}(1)$ words of memory apart from input and output.*

Proof. The correctness follows from Lemma A.0.2 and the description above. We only have to prove the time and space bounds. In terms of execution time, we use Duval's algorithm to compute the indices $d_1, \dots, d_m$, which takes $\mathcal{O}(n)$ time [Duval, 1983, Algorithm 2.1, Theorem 2.1]. This clearly dominates the execution time of Algorithm A.1. It remains to be shown that $\mathcal{O}(1)$ words of memory are sufficient. Duval's algorithm computes the indices $d_1, \dots, d_m$ in a greedy manner, i.e. it outputs the indices one at a time and in left-to-right order. Since Algorithm A.1 also processes the indices in left-to-right order, it is never necessary to keep more than two indices in memory at the same time.

Therefore, we can interleave the execution of Duval's algorithm and Algorithm A.1 such that we only compute the next index d_i once it is actually needed. Apart from input and ouput, Duval's algorithm uses $\mathcal{O}(1)$ words of memory [Duval, 1983, Theorem 2.1]. Since Algorithm A.1 only needs to keep the variables l, z, and two indices d_i and d_{i-1} in memory, the additional memory usage is bound by $\mathcal{O}(1)$ words. □

Appendix B

Additional Experimental Results

The following pages contain the complete results for the experiments from Section 7.5. Each algorithm was executed five times per text. The median is the final result. The throughput of xss-real, xss-bps-lcp, and xss-bps is given both *without* constructing the support data structure (value in parentheses), and *with* computing the support data structure.

Text	**Algorithm**		**Throughput in MiB/s**		**Memory in n bits additional / total**	
fib41	xss-real	(dyn.-buf., $\delta = 0$)	(43.40)	33.69	0.0002	10.0002
fib41	xss-real	(dyn.-buf., $\delta = 4$)	(40.32)	31.80	0.0002	10.0002
fib41	xss-real	(dynamic, $\delta = 0$)	(33.77)	27.58	0.0001	10.0001
fib41	xss-real	(dynamic, $\delta = 4$)	(31.77)	26.24	0.0001	10.0001
fib41	xss-bps-lcp	(dyn.-buf., $\delta = 0$)	(45.86)	**35.15**	0.0002	10.0002
fib41	xss-bps-lcp	(dyn.-buf., $\delta = 4$)	(43.29)	33.62	0.0001	10.0001
fib41	xss-bps-lcp	(dynamic, $\delta = 0$)	(34.15)	27.84	0.0001	10.0001
fib41	xss-bps-lcp	(dynamic, $\delta = 4$)	(21.56)	18.86	0.0001	10.0001
fib41	xss-bps	(dyn.-buf.)	(33.19)	27.19	0.0001	10.0001
fib41	xss-bps	(dynamic)	(30.83)	25.59	0.0000	**10.0000**
fib41	sdsl-naive			34.19	0.0000	40.0000
fib41	sdsl-prezza			1.43	**0.0000**	40.0000
fib41	sdsl-1k-prezza			18.53	2.0000	42.0000
fib41	sdsl-herlez			1.17	0.0000	40.0000
fib41	sdsl-1k-herlez			24.68	8.0000	48.0000
fib41	sdsl-isa-nsv			3.82	24.0000	64.0000
fib41	g-saca-lyn			12.58	96.0000	136.0000
fib41	g-saca			5.32	96.0000	136.0000
fib41	non-ele-lyn			4.26	0.0079	40.0079
rs.13	xss-real	(dyn.-buf., $\delta = 0$)	(48.23)	36.52	0.0003	10.0003
rs.13	xss-real	(dyn.-buf., $\delta = 4$)	(44.69)	34.46	0.0002	10.0002
rs.13	xss-real	(dynamic, $\delta = 0$)	(36.71)	29.51	0.0001	10.0001
rs.13	xss-real	(dynamic, $\delta = 4$)	(33.36)	27.31	0.0001	10.0001
rs.13	xss-bps-lcp	(dyn.-buf., $\delta = 0$)	(53.36)	39.39	0.0002	10.0002
rs.13	xss-bps-lcp	(dyn.-buf., $\delta = 4$)	(50.13)	37.60	0.0002	10.0002
rs.13	xss-bps-lcp	(dynamic, $\delta = 0$)	(36.42)	29.32	0.0001	10.0001
rs.13	xss-bps-lcp	(dynamic, $\delta = 4$)	(22.47)	19.55	0.0001	10.0001
rs.13	xss-bps	(dyn.-buf.)	(44.44)	34.31	0.0001	10.0001
rs.13	xss-bps	(dynamic)	(39.80)	31.47	0.0000	**10.0000**
rs.13	sdsl-naive			**47.11**	0.0000	40.0000
rs.13	sdsl-prezza			1.70	**0.0000**	40.0000
rs.13	sdsl-1k-prezza			21.20	2.0000	42.0000
rs.13	sdsl-herlez			1.26	0.0000	40.0000
rs.13	sdsl-1k-herlez			31.74	8.0000	48.0000
rs.13	sdsl-isa-nsv			3.94	24.0000	64.0000
rs.13	g-saca-lyn			6.66	96.0000	136.0000
rs.13	g-saca			3.53	96.0000	136.0000
rs.13	non-ele-lyn			4.39	0.0097	40.0097
tm29	xss-real	(dyn.-buf., $\delta = 0$)	(45.88)	35.18	0.0002	10.0002
tm29	xss-real	(dyn.-buf., $\delta = 4$)	(42.39)	33.10	0.0002	10.0002
tm29	xss-real	(dynamic, $\delta = 0$)	(34.71)	28.22	0.0001	10.0001
tm29	xss-real	(dynamic, $\delta = 4$)	(31.60)	26.13	0.0001	10.0001
tm29	xss-bps-lcp	(dyn.-buf., $\delta = 0$)	(56.35)	41.03	0.0002	10.0002
tm29	xss-bps-lcp	(dyn.-buf., $\delta = 4$)	(52.55)	38.98	0.0001	10.0001
tm29	xss-bps-lcp	(dynamic, $\delta = 0$)	(39.05)	31.02	0.0001	10.0001
tm29	xss-bps-lcp	(dynamic, $\delta = 4$)	(23.99)	20.70	0.0001	10.0001
tm29	xss-bps	(dyn.-buf.)	(48.02)	36.43	0.0001	10.0001
tm29	xss-bps	(dynamic)	(42.51)	33.17	0.0000	**10.0000**
tm29	sdsl-naive			**48.34**	0.0000	40.0000
tm29	sdsl-prezza			1.75	**0.0000**	40.0000
tm29	sdsl-1k-prezza			21.13	2.0000	42.0000
tm29	sdsl-herlez			1.26	0.0000	40.0000
tm29	sdsl-1k-herlez			31.22	8.0000	48.0000
tm29	sdsl-isa-nsv			4.24	24.0000	64.0000
tm29	g-saca-lyn			6.39	96.0000	136.0000
tm29	g-saca			3.40	96.0000	136.0000
tm29	non-ele-lyn			4.88	0.0078	40.0078

Text	Algorithm		Throughput in MiB/s		Memory in n bits additional / total	
cere	xss-real	(dyn.-buf., $\delta = 0$)	(38.50)	30.56	0.0100	10.0100
cere	xss-real	(dyn.-buf., $\delta = 4$)	(38.52)	**30.58**	0.0100	10.0100
cere	xss-real	(dynamic, $\delta = 0$)	(28.19)	23.69	0.0003	10.0003
cere	xss-real	(dynamic, $\delta = 4$)	(17.10)	15.34	0.0001	**10.0001**
cere	xss-bps-lcp	(dyn.-buf., $\delta = 0$)	(6.59)	6.31	0.0100	10.0100
cere	xss-bps-lcp	(dyn.-buf., $\delta = 4$)	(6.52)	6.25	0.0100	10.0100
cere	xss-bps-lcp	(dynamic, $\delta = 0$)	(6.18)	5.93	0.0003	10.0003
cere	xss-bps-lcp	(dynamic, $\delta = 4$)	(5.41)	5.22	0.0001	**10.0001**
cere	xss-bps	(dyn.-buf.)	(4.69)	4.54	0.0050	10.0050
cere	xss-bps	(dynamic)	(4.64)	4.50	0.0001	**10.0001**
cere	sdsl-naive			4.51	0.0048	40.0048
cere	sdsl-prezza			1.16	**0.0000**	40.0000
cere	sdsl-1k-prezza			2.87	3.0048	43.0048
cere	sdsl-herlez			0.63	0.0048	40.0048
cere	sdsl-1k-herlez			1.33	8.0047	48.0048
cere	sdsl-isa-nsv			5.32	24.0048	64.0048
cere	g-saca-lyn			3.90	96.0000	136.0000
cere	g-saca			2.53	96.0000	136.0000
cere	non-ele-lyn			6.88	0.0046	40.0046
coreutils	xss-real	(dyn.-buf., $\delta = 0$)	(39.94)	31.43	0.0022	10.0022
coreutils	xss-real	(dyn.-buf., $\delta = 4$)	(39.41)	31.11	0.0021	10.0021
coreutils	xss-real	(dynamic, $\delta = 0$)	(29.08)	24.29	0.0006	10.0006
coreutils	xss-real	(dynamic, $\delta = 4$)	(20.80)	18.23	0.0006	10.0006
coreutils	xss-bps-lcp	(dyn.-buf., $\delta = 0$)	(41.35)	32.30	0.0022	10.0022
coreutils	xss-bps-lcp	(dyn.-buf., $\delta = 4$)	(39.33)	31.06	0.0021	10.0021
coreutils	xss-bps-lcp	(dynamic, $\delta = 0$)	(30.66)	25.39	0.0006	10.0006
coreutils	xss-bps-lcp	(dynamic, $\delta = 4$)	(21.48)	18.75	0.0006	10.0006
coreutils	xss-bps	(dyn.-buf.)	(47.80)	36.11	0.0011	10.0011
coreutils	xss-bps	(dynamic)	(44.27)	34.06	**0.0005**	**10.0005**
coreutils	sdsl-naive			**60.54**	0.0008	40.0008
coreutils	sdsl-prezza			1.54	0.0009	40.0009
coreutils	sdsl-1k-prezza			8.35	8.0009	48.0009
coreutils	sdsl-herlez			1.37	0.0009	40.0009
coreutils	sdsl-1k-herlez			38.26	8.0008	48.0009
coreutils	sdsl-isa-nsv			6.83	24.0008	64.0008
coreutils	g-saca-lyn			3.62	96.0000	136.0000
coreutils	g-saca			2.71	96.0000	136.0000
coreutils	non-ele-lyn			8.26	0.0103	40.0103
ecoli	xss-real	(dyn.-buf., $\delta = 0$)	(37.02)	29.77	0.0005	10.0005
ecoli	xss-real	(dyn.-buf., $\delta = 4$)	(36.91)	29.70	0.0005	10.0005
ecoli	xss-real	(dynamic, $\delta = 0$)	(26.67)	22.69	0.0003	10.0003
ecoli	xss-real	(dynamic, $\delta = 4$)	(16.35)	14.76	0.0002	10.0002
ecoli	xss-bps-lcp	(dyn.-buf., $\delta = 0$)	(39.63)	31.43	0.0004	10.0004
ecoli	xss-bps-lcp	(dyn.-buf., $\delta = 4$)	(37.21)	29.89	0.0004	10.0004
ecoli	xss-bps-lcp	(dynamic, $\delta = 0$)	(28.77)	24.19	0.0002	10.0002
ecoli	xss-bps-lcp	(dynamic, $\delta = 4$)	(16.92)	15.23	0.0002	10.0002
ecoli	xss-bps	(dyn.-buf.)	(42.44)	33.18	0.0002	10.0002
ecoli	xss-bps	(dynamic)	(38.63)	30.80	0.0001	**10.0001**
ecoli	sdsl-naive			**46.42**	0.0024	40.0024
ecoli	sdsl-prezza			1.80	**0.0000**	40.0000
ecoli	sdsl-1k-prezza			13.15	4.0024	44.0024
ecoli	sdsl-herlez			1.38	0.0024	40.0024
ecoli	sdsl-1k-herlez			33.83	8.0024	48.0024
ecoli	sdsl-isa-nsv			5.50	24.0024	64.0024
ecoli	g-saca-lyn			4.17	96.0000	136.0000
ecoli	g-saca			2.75	96.0000	136.0000
ecoli	non-ele-lyn			7.25	0.0187	40.0187

			Throughput		Memory in n bits	
Text	Algorithm		in MiB/s		additional / total	
einstein.de	xss-real	(dyn.-buf., $\delta = 0$)	(42.00)	32.92	0.0006	10.0006
einstein.de	xss-real	(dyn.-buf., $\delta = 4$)	(41.34)	32.51	0.0006	10.0006
einstein.de	xss-real	(dynamic, $\delta = 0$)	(29.58)	24.77	0.0004	10.0004
einstein.de	xss-real	(dynamic, $\delta = 4$)	(22.26)	19.42	0.0004	10.0004
einstein.de	xss-bps-lcp	(dyn.-buf., $\delta = 0$)	(43.11)	33.60	0.0006	10.0006
einstein.de	xss-bps-lcp	(dyn.-buf., $\delta = 4$)	(40.77)	32.16	0.0006	10.0006
einstein.de	xss-bps-lcp	(dynamic, $\delta = 0$)	(31.50)	26.10	0.0003	10.0003
einstein.de	xss-bps-lcp	(dynamic, $\delta = 4$)	(22.04)	19.26	0.0003	10.0003
einstein.de	xss-bps	(dyn.-buf.)	(48.10)	36.55	0.0003	10.0003
einstein.de	xss-bps	(dynamic)	(43.53)	33.85	0.0001	**10.0001**
einstein.de	sdsl-naive			**55.12**	0.0001	40.0001
einstein.de	sdsl-prezza			1.68	**0.0000**	40.0000
einstein.de	sdsl-1k-prezza			9.18	7.0002	47.0002
einstein.de	sdsl-herlez			1.39	0.0002	40.0002
einstein.de	sdsl-1k-herlez			38.20	8.0002	48.0002
einstein.de	sdsl-isa-nsv			6.80	24.0001	64.0001
einstein.de	g-saca-lyn			3.48	96.0000	136.0000
einstein.de	g-saca			2.52	96.0000	136.0000
einstein.de	non-ele-lyn			8.51	0.0227	40.0227
einstein.en	xss-real	(dyn.-buf., $\delta = 0$)	(41.29)	32.27	0.0003	10.0003
einstein.en	xss-real	(dyn.-buf., $\delta = 4$)	(40.56)	31.83	0.0003	10.0003
einstein.en	xss-real	(dynamic, $\delta = 0$)	(29.72)	24.75	0.0004	10.0004
einstein.en	xss-real	(dynamic, $\delta = 4$)	(22.04)	19.18	0.0003	10.0003
einstein.en	xss-bps-lcp	(dyn.-buf., $\delta = 0$)	(34.32)	27.85	0.0003	10.0003
einstein.en	xss-bps-lcp	(dyn.-buf., $\delta = 4$)	(32.85)	26.88	0.0003	10.0003
einstein.en	xss-bps-lcp	(dynamic, $\delta = 0$)	(26.71)	22.63	0.0004	10.0004
einstein.en	xss-bps-lcp	(dynamic, $\delta = 4$)	(19.41)	17.16	0.0003	10.0003
einstein.en	xss-bps	(dyn.-buf.)	(33.98)	27.63	**0.0001**	**10.0001**
einstein.en	xss-bps	(dynamic)	(31.66)	26.07	0.0002	10.0002
einstein.en	sdsl-naive			**37.66**	0.0066	40.0066
einstein.en	sdsl-prezza			1.64	0.0066	40.0066
einstein.en	sdsl-1k-prezza			8.18	8.0066	48.0066
einstein.en	sdsl-herlez			1.38	0.0066	40.0066
einstein.en	sdsl-1k-herlez			33.81	8.0066	48.0066
einstein.en	sdsl-isa-nsv			5.50	24.0066	64.0066
einstein.en	g-saca-lyn			2.84	96.0000	136.0000
einstein.en	g-saca			2.11	96.0000	136.0000
einstein.en	non-ele-lyn			6.87	0.0045	40.0045
influenza	xss-real	(dyn.-buf., $\delta = 0$)	(39.18)	30.99	0.0007	10.0007
influenza	xss-real	(dyn.-buf., $\delta = 4$)	(38.81)	30.76	0.0007	10.0007
influenza	xss-real	(dynamic, $\delta = 0$)	(27.73)	23.36	0.0002	10.0002
influenza	xss-real	(dynamic, $\delta = 4$)	(17.16)	15.38	0.0002	10.0002
influenza	xss-bps-lcp	(dyn.-buf., $\delta = 0$)	(42.29)	32.90	0.0007	10.0007
influenza	xss-bps-lcp	(dyn.-buf., $\delta = 4$)	(39.88)	31.43	0.0007	10.0007
influenza	xss-bps-lcp	(dynamic, $\delta = 0$)	(29.78)	24.80	0.0002	10.0002
influenza	xss-bps-lcp	(dynamic, $\delta = 4$)	(17.53)	15.68	0.0001	**10.0001**
influenza	xss-bps	(dyn.-buf.)	(44.72)	34.36	0.0004	10.0004
influenza	xss-bps	(dynamic)	(41.30)	32.30	0.0001	**10.0001**
influenza	sdsl-naive			**51.71**	0.0002	40.0002
influenza	sdsl-prezza			1.80	**0.0000**	40.0000
influenza	sdsl-1k-prezza			13.45	4.0003	44.0003
influenza	sdsl-herlez			1.38	0.0003	40.0003
influenza	sdsl-1k-herlez			36.19	8.0003	48.0003
influenza	sdsl-isa-nsv			6.28	24.0002	64.0002
influenza	g-saca-lyn			4.18	96.0000	136.0000
influenza	g-saca			2.75	96.0000	136.0000
influenza	non-ele-lyn			8.04	0.0136	40.0136

Text	Algorithm		Throughput		Memory in n bits	
			in MiB/s		additional /	total
kernel	xss-real	(dyn.-buf., $\delta = 0$)	(41.33)	32.30	0.0003	10.0003
kernel	xss-real	(dyn.-buf., $\delta = 4$)	(40.89)	32.04	0.0003	10.0003
kernel	xss-real	(dynamic, $\delta = 0$)	(28.48)	23.88	0.0002	10.0002
kernel	xss-real	(dynamic, $\delta = 4$)	(20.47)	17.98	0.0001	**10.0001**
kernel	xss-bps-lcp	(dyn.-buf., $\delta = 0$)	(43.60)	33.68	0.0003	10.0003
kernel	xss-bps-lcp	(dyn.-buf., $\delta = 4$)	(41.46)	32.38	0.0003	10.0003
kernel	xss-bps-lcp	(dynamic, $\delta = 0$)	(30.55)	25.32	0.0002	10.0002
kernel	xss-bps-lcp	(dynamic, $\delta = 4$)	(21.06)	18.44	0.0001	**10.0001**
kernel	xss-bps	(dyn.-buf.)	(49.12)	36.88	0.0001	**10.0001**
kernel	xss-bps	(dynamic)	(43.32)	33.51	0.0001	**10.0001**
kernel	sdsl-naive			**58.16**	**0.0000**	40.0000
kernel	sdsl-prezza			1.56	0.0001	40.0001
kernel	sdsl-1k-prezza			8.27	8.0001	48.0001
kernel	sdsl-herlez			1.39	0.0001	40.0001
kernel	sdsl-1k-herlez			39.65	8.0001	48.0001
kernel	sdsl-isa-nsv			6.39	24.0000	64.0000
kernel	g-saca-lyn			3.32	96.0000	136.0000
kernel	g-saca			2.49	96.0000	136.0000
kernel	non-ele-lyn			8.50	0.0082	40.0082
leaders	xss-real	(dyn.-buf., $\delta = 0$)	(29.28)	24.58	0.0014	10.0014
leaders	xss-real	(dyn.-buf., $\delta = 4$)	(32.36)	26.73	0.0013	10.0013
leaders	xss-real	(dynamic, $\delta = 0$)	(22.11)	19.32	0.0010	10.0010
leaders	xss-real	(dynamic, $\delta = 4$)	(16.26)	14.71	0.0009	10.0009
leaders	xss-bps-lcp	(dyn.-buf., $\delta = 0$)	(33.38)	27.41	0.0010	10.0010
leaders	xss-bps-lcp	(dyn.-buf., $\delta = 4$)	(31.68)	26.26	0.0009	10.0009
leaders	xss-bps-lcp	(dynamic, $\delta = 0$)	(20.93)	18.42	0.0008	10.0008
leaders	xss-bps-lcp	(dynamic, $\delta = 4$)	(16.57)	14.95	0.0007	10.0007
leaders	xss-bps	(dyn.-buf.)	(30.49)	25.44	0.0005	**10.0005**
leaders	xss-bps	(dynamic)	(27.93)	23.62	0.0005	**10.0005**
leaders	sdsl-naive			**31.97**	0.0003	40.0003
leaders	sdsl-prezza			0.55	**0.0000**	40.0000
leaders	sdsl-1k-prezza			8.46	7.0004	47.0004
leaders	sdsl-herlez			1.12	0.0003	40.0003
leaders	sdsl-1k-herlez			27.77	8.0003	48.0003
leaders	sdsl-isa-nsv			12.15	24.0003	64.0003
leaders	g-saca-lyn			5.67	96.0000	136.0000
leaders	g-saca			3.28	96.0000	136.0000
leaders	non-ele-lyn			18.44	0.0448	40.0448
para	xss-real	(dyn.-buf., $\delta = 0$)	(38.00)	30.25	0.0051	10.0051
para	xss-real	(dyn.-buf., $\delta = 4$)	(38.00)	**30.25**	0.0051	10.0051
para	xss-real	(dynamic, $\delta = 0$)	(27.47)	23.18	0.0002	10.0002
para	xss-real	(dynamic, $\delta = 4$)	(16.64)	14.96	0.0001	**10.0001**
para	xss-bps-lcp	(dyn.-buf., $\delta = 0$)	(19.82)	17.48	0.0051	10.0051
para	xss-bps-lcp	(dyn.-buf., $\delta = 4$)	(19.23)	17.02	0.0051	10.0051
para	xss-bps-lcp	(dynamic, $\delta = 0$)	(16.37)	14.74	0.0002	10.0002
para	xss-bps-lcp	(dynamic, $\delta = 4$)	(11.83)	10.95	0.0001	**10.0001**
para	xss-bps	(dyn.-buf.)	(16.29)	14.67	0.0025	10.0025
para	xss-bps	(dynamic)	(15.66)	14.16	0.0001	**10.0001**
para	sdsl-naive			16.41	0.0024	40.0024
para	sdsl-prezza			1.44	**0.0000**	40.0000
para	sdsl-1k-prezza			6.42	3.0024	43.0024
para	sdsl-herlez			0.94	0.0024	40.0024
para	sdsl-1k-herlez			4.36	8.0024	48.0024
para	sdsl-isa-nsv			5.19	24.0024	64.0024
para	g-saca-lyn			3.86	96.0000	136.0000
para	g-saca			2.53	96.0000	136.0000
para	non-ele-lyn			6.71	0.0049	40.0049

Text	Algorithm		Throughput in MiB/s		Memory in n bits additional / total	
dna	xss-real	(dyn.-buf., $\delta = 0$)	(34.33)	27.86	0.0002	10.0002
dna	xss-real	(dyn.-buf., $\delta = 4$)	(34.16)	27.75	0.0002	10.0002
dna	xss-real	(dynamic, $\delta = 0$)	(26.32)	22.34	0.0001	10.0001
dna	xss-real	(dynamic, $\delta = 4$)	(15.86)	14.32	0.0001	10.0001
dna	xss-bps-lcp	(dyn.-buf., $\delta = 0$)	(36.39)	29.20	0.0001	10.0001
dna	xss-bps-lcp	(dyn.-buf., $\delta = 4$)	(34.37)	27.88	0.0001	10.0001
dna	xss-bps-lcp	(dynamic, $\delta = 0$)	(28.30)	23.75	0.0001	10.0001
dna	xss-bps-lcp	(dynamic, $\delta = 4$)	(16.36)	14.73	0.0001	10.0001
dna	xss-bps	(dyn.-buf.)	(40.42)	31.74	0.0001	10.0001
dna	xss-bps	(dynamic)	(37.97)	30.20	0.0000	**10.0000**
dna	sdsl-naive			**45.92**	0.0001	40.0001
dna	sdsl-prezza			1.77	**0.0000**	40.0000
dna	sdsl-1k-prezza			11.29	5.0001	45.0001
dna	sdsl-herlez			1.38	0.0001	40.0001
dna	sdsl-1k-herlez			33.74	8.0001	48.0001
dna	sdsl-isa-nsv			4.75	24.0001	64.0001
dna	g-saca-lyn			3.43	96.0000	136.0000
dna	g-saca			2.27	96.0000	136.0000
dna	non-ele-lyn			5.81	0.0052	40.0052
english.1G	xss-real	(dyn.-buf., $\delta = 0$)	(39.57)	31.21	0.0013	10.0013
english.1G	xss-real	(dyn.-buf., $\delta = 4$)	(38.45)	30.51	0.0013	10.0013
english.1G	xss-real	(dynamic, $\delta = 0$)	(29.13)	24.33	0.0003	**10.0003**
english.1G	xss-real	(dynamic, $\delta = 4$)	(21.32)	18.63	0.0003	**10.0003**
english.1G	xss-bps-lcp	(dyn.-buf., $\delta = 0$)	(41.15)	32.19	0.0013	10.0013
english.1G	xss-bps-lcp	(dyn.-buf., $\delta = 4$)	(38.72)	30.69	0.0013	10.0013
english.1G	xss-bps-lcp	(dynamic, $\delta = 0$)	(31.42)	25.92	0.0003	**10.0003**
english.1G	xss-bps-lcp	(dynamic, $\delta = 4$)	(22.19)	19.30	0.0003	**10.0003**
english.1G	xss-bps	(dyn.-buf.)	(46.80)	35.55	0.0006	10.0006
english.1G	xss-bps	(dynamic)	(43.49)	33.61	0.0003	**10.0003**
english.1G	sdsl-naive			**55.16**	**0.0000**	40.0000
english.1G	sdsl-prezza			1.65	0.0000	40.0000
english.1G	sdsl-1k-prezza			8.33	8.0000	48.0000
english.1G	sdsl-herlez			1.39	0.0000	40.0000
english.1G	sdsl-1k-herlez			38.54	8.0000	48.0000
english.1G	sdsl-isa-nsv			4.56	24.0000	64.0000
english.1G	g-saca-lyn			2.51	96.0000	136.0000
english.1G	g-saca			1.75	96.0000	136.0000
english.1G	non-ele-lyn			5.16	0.0020	40.0020
pitches	xss-real	(dyn.-buf., $\delta = 0$)	(40.22)	31.81	0.0143	10.0143
pitches	xss-real	(dyn.-buf., $\delta = 4$)	(40.03)	31.69	0.0142	10.0142
pitches	xss-real	(dynamic, $\delta = 0$)	(29.45)	24.67	0.0007	10.0007
pitches	xss-real	(dynamic, $\delta = 4$)	(19.36)	17.18	0.0006	10.0006
pitches	xss-bps-lcp	(dyn.-buf., $\delta = 0$)	(34.85)	28.35	0.0143	10.0143
pitches	xss-bps-lcp	(dyn.-buf., $\delta = 4$)	(33.22)	27.26	0.0142	10.0142
pitches	xss-bps-lcp	(dynamic, $\delta = 0$)	(24.34)	20.98	0.0007	10.0007
pitches	xss-bps-lcp	(dynamic, $\delta = 4$)	(17.52)	15.71	0.0004	10.0004
pitches	xss-bps	(dyn.-buf.)	(34.71)	28.26	0.0071	10.0071
pitches	xss-bps	(dynamic)	(33.40)	27.39	**0.0002**	**10.0002**
pitches	sdsl-naive			**40.46**	0.0049	40.0049
pitches	sdsl-prezza			1.25	0.0049	40.0050
pitches	sdsl-1k-prezza			7.40	8.0049	48.0050
pitches	sdsl-herlez			1.22	0.0049	40.0049
pitches	sdsl-1k-herlez			25.15	8.0049	48.0049
pitches	sdsl-isa-nsv			8.51	24.0049	64.0049
pitches	g-saca-lyn			4.86	96.0000	136.0000
pitches	g-saca			3.81	96.0000	136.0000
pitches	non-ele-lyn			11.66	0.0377	40.0377

Text	Algorithm		Throughput in MiB/s		Memory in n bits additional / total	
proteins	xss-real	(dyn.-buf., $\delta = 0$)	(41.49)	32.41	0.0001	**10.0001**
proteins	xss-real	(dyn.-buf., $\delta = 4$)	(41.04)	32.13	0.0001	**10.0001**
proteins	xss-real	(dynamic, $\delta = 0$)	(29.68)	24.72	0.0001	**10.0001**
proteins	xss-real	(dynamic, $\delta = 4$)	(21.30)	18.62	0.0001	**10.0001**
proteins	xss-bps-lcp	(dyn.-buf., $\delta = 0$)	(42.86)	33.23	0.0001	**10.0001**
proteins	xss-bps-lcp	(dyn.-buf., $\delta = 4$)	(40.71)	31.93	0.0001	**10.0001**
proteins	xss-bps-lcp	(dynamic, $\delta = 0$)	(31.57)	26.02	0.0001	**10.0001**
proteins	xss-bps-lcp	(dynamic, $\delta = 4$)	(21.84)	19.03	0.0001	**10.0001**
proteins	xss-bps	(dyn.-buf.)	(49.00)	36.82	0.0001	**10.0001**
proteins	xss-bps	(dynamic)	(44.05)	33.95	0.0001	**10.0001**
proteins	sdsl-naive			**56.80**	0.0001	40.0001
proteins	sdsl-prezza			1.75	**0.0000**	40.0000
proteins	sdsl-1k-prezza			11.73	5.0000	45.0001
proteins	sdsl-herlez			1.39	0.0001	40.0001
proteins	sdsl-1k-herlez			38.65	8.0000	48.0001
proteins	sdsl-isa-nsv			4.36	24.0001	64.0001
proteins	g-saca-lyn			2.47	96.0000	136.0000
proteins	g-saca			1.79	96.0000	136.0000
proteins	non-ele-lyn			4.80	0.0018	40.0018
sources	xss-real	(dyn.-buf., $\delta = 0$)	(40.58)	31.84	0.0046	10.0046
sources	xss-real	(dyn.-buf., $\delta = 4$)	(40.19)	31.59	0.0046	10.0046
sources	xss-real	(dynamic, $\delta = 0$)	(28.95)	24.21	0.0015	10.0015
sources	xss-real	(dynamic, $\delta = 4$)	(20.70)	18.15	0.0014	**10.0014**
sources	xss-bps-lcp	(dyn.-buf., $\delta = 0$)	(41.55)	32.43	0.0046	10.0046
sources	xss-bps-lcp	(dyn.-buf., $\delta = 4$)	(39.66)	31.27	0.0046	10.0046
sources	xss-bps-lcp	(dynamic, $\delta = 0$)	(29.69)	24.72	0.0015	10.0015
sources	xss-bps-lcp	(dynamic, $\delta = 4$)	(21.19)	18.53	0.0014	**10.0014**
sources	xss-bps	(dyn.-buf.)	(46.92)	35.61	0.0023	10.0023
sources	xss-bps	(dynamic)	(43.29)	33.48	0.0014	**10.0014**
sources	sdsl-naive			**59.04**	**0.0001**	40.0001
sources	sdsl-prezza			1.54	0.0001	40.0001
sources	sdsl-1k-prezza			8.34	8.0001	48.0001
sources	sdsl-herlez			1.38	0.0001	40.0001
sources	sdsl-1k-herlez			39.81	8.0001	48.0001
sources	sdsl-isa-nsv			7.23	24.0001	64.0001
sources	g-saca-lyn			3.46	96.0000	136.0000
sources	g-saca			2.60	96.0000	136.0000
sources	non-ele-lyn			9.31	0.0100	40.0100
xml	xss-real	(dyn.-buf., $\delta = 0$)	(44.34)	34.15	0.0004	10.0004
xml	xss-real	(dyn.-buf., $\delta = 4$)	(43.11)	33.41	0.0003	10.0003
xml	xss-real	(dynamic, $\delta = 0$)	(31.84)	26.22	0.0003	10.0003
xml	xss-real	(dynamic, $\delta = 4$)	(23.37)	20.19	0.0003	10.0003
xml	xss-bps-lcp	(dyn.-buf., $\delta = 0$)	(45.74)	34.97	0.0004	10.0004
xml	xss-bps-lcp	(dyn.-buf., $\delta = 4$)	(43.68)	33.75	0.0003	10.0003
xml	xss-bps-lcp	(dynamic, $\delta = 0$)	(31.05)	25.68	0.0003	10.0003
xml	xss-bps-lcp	(dynamic, $\delta = 4$)	(24.27)	20.86	0.0003	10.0003
xml	xss-bps	(dyn.-buf.)	(57.12)	41.26	0.0002	**10.0002**
xml	xss-bps	(dynamic)	(51.50)	38.24	0.0003	10.0003
xml	sdsl-naive			**77.23**	0.0000	40.0000
xml	sdsl-prezza			1.64	**0.0000**	40.0000
xml	sdsl-1k-prezza			9.68	7.0000	47.0000
xml	sdsl-herlez			1.39	0.0000	40.0000
xml	sdsl-1k-herlez			48.39	8.0000	48.0000
xml	sdsl-isa-nsv			6.98	24.0000	64.0000
xml	g-saca-lyn			3.11	96.0000	136.0000
xml	g-saca			2.34	96.0000	136.0000
xml	non-ele-lyn			8.66	0.0071	40.0071

Bibliography

U. Baier. Linear-time suffix sorting - A new approach for suffix array construction. Master's thesis, Ulm University, 2015. URL `https://www.uni-ulm.de/fileadmin/website_uni_ulm/iui.inst.190/Mitarbeiter/baier/gsaca.pdf`.

U. 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. URL `https://doi.org/10.4230/LIPIcs.CPM.2016.23`.

H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta. The "Runs" theorem. *SIAM Journal on Computing*, 46(5):1501–1514, 2017. URL `https://doi.org/10.1137/15M1011032`.

D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005. URL `https://doi.org/10.1007/s00453-004-1146-6`.

O. Berkman, B. Schieber, and U. Vishkin. Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. *Journal of Algorithms*, 14(3):344–370, 1993. URL `https://doi.org/10.1006/jagm.1993.1018`.

M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation, 1994.

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. URL `https://doi.org/10.2307/1970044`.

M. Crochemore and L. M. Russo. Cartesian and Lyndon trees. *Theoretical Computer Science*, In Press, Corrected Proof, 2018. URL `https://doi.org/10.1016/j.tcs.2018.08.011`.

P. Davoodi, R. Raman, and S. R. Satti. On succinct representations of binary trees. *Mathematics in Computer Science*, 11(2):177–189, Jun 2017. URL `https://doi.org/10.1007/s11786-017-0294-4`.

N. G. de Bruijn. A combinatorial problem. In *Proceedings of the Koninklijke Nederlandse Akademie van Wetenschappen*, pages 758–764, 1946.

O. Delpratt, N. Rahman, and R. Raman. Engineering the LOUDS succinct tree representation. In *Proceedings of the 5th International Workshop on Experimental and Efficient Algorithms (WEA 2006)*, pages 134–145, Cala Galdana, Spain, May 2006. URL `https://doi.org/10.1007/11764298_12`.

J. P. Duval. Factorizing words over an ordered alphabet. *Journal of Algorithms*, 4(4): 363–381, 1983. URL `https://doi.org/10.1016/0196-6774(83)90017-2`.

J. 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, Apr. 2010. URL `https://doi.org/10.1007/978-3-642-12200-2_16`.

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

J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM 2006)*, pages 36–48, Barcelona, Spain, July 2006. URL `https://doi.org/10.1007/11780441_5`.

J. Fischer and F. Kurpicz. Dismantling DivSufSort. In *Proceedings of the 25th Prague Stringology Conference (PSC 2017)*, pages 62–76, Prague, Czech Republic, Aug. 2017.

J. Fischer, V. Mäkinen, and G. Navarro. An(other) entropy-bounded compressed suffix tree. In *Proceedings of the 19th Annual Symposium on Combinatorial Pattern Matching (CPM 2008)*, pages 152–165, Pisa, Italy, June 2008.

C. Flye Sainte-Marie. Solution to question nr. 48. *L'Intermédiaire des Mathématiciens*, 1894.

F. Franek, R. Simpson, and W. Smyth. The maximum number of runs in a string. In *Proceedings of the 14th Australasian Workshop on Combinatorial Algorithms (AWOCA 2003)*, Seoul, Korea, July 2003.

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

F. Franek, A. Paracha, and W. 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, Aug. 2017.

F. Franek, M. Liut, and W. F. Smyth. On Baier's sort of maximal Lyndon substrings. In *Proceedings of the 22nd Prague Stringology Conference (PSC 2018)*, pages 63–78, Prague, Czech Republic, Aug. 2018.

S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *Proceedings of the 13th International Symposium on Experimental Algorithms (SEA 2014)*, pages 326–337, Copenhagen, Denmark, June 2014. URL `https://doi.org/10.1007/978-3-319-07959-2_28`.

K. Goto and H. Bannai. Simpler and faster lempel ziv factorization. In *Proceedings of the 2013 Data Compression Conference (DCC 2013)*, pages 133–142, Snowbird, UT, USA, Mar. 2013. URL `https://doi.org/10.1109/DCC.2013.21`.

T. 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, Feb. 1998. URL `https://doi.org/10.1007/BFb0028575`.

C. Hierholzer and C. Wiener. Ueber die möglichkeit, einen linienzug ohne wiederholung und ohne unterbrechung zu umfahren. *Mathematische Annalen*, 6(1):30–32, 1873. URL `https://doi.org/10.1007/BF01442866`.

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

G. Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS 1989)*, pages 549–554, Research Triangle Park, NC, USA, Oct. 1989. URL `https://doi.org/10.1109/SFCS.1989.63533`.

D. E. Knuth. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1.* Addison-Wesley Professional, 2011. ISBN 0201038048.

C. Leiserson, H. Prokop, and K. Randall. Using de Bruijn sequences to index a 1 in a computer word, 1970. URL `http://supertech.csail.mit.edu/papers/debruijn.pdf`. Published at the MIT Laboratory for Computer Science.

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

F. A. Louza, S. Mantaci, G. Manzini, M. Sciortino, and G. P. Telles. Inducing the Lyndon array, 2019. URL `https://arxiv.org/abs/1905.12987`. Version [v1].

U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *Proceedings of the 1st Annual Symposium on Discrete Algorithms (SODA 1990)*, pages 319–327, Philadelphia, PA, USA, 1990. URL http://dl.acm.org/citation.cfm?id=320176.320218.

U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. URL https://doi.org/10.1137/0222058.

J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001. URL https://doi.org/10.1137/s0097539799364092.

G. Nong. Practical linear-time O(1)-workspace suffix sorting for constant alphabets. *ACM-Transactions on Information Systems*, 31(3):15:1–15:15, 2013. URL https://doi.org/10.1145/2493175.2493180.

S. G. Park, A. Amir, G. M. Landau, and K. Park. Cartesian tree matching and indexing, 2019. URL https://arxiv.org/abs/1905.08974. To be released in: Proceedings of the 30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019).

A. Policriti and N. Prezza. Fast longest common extensions in small space, 2016. URL https://arxiv.org/abs/1607.06660.

N. Prezza. In-place sparse suffix sorting. In *Proceedings of the 29th Annual Symposium on Discrete Algorithms (SODA 2018)*, pages 1496–1508, New Orleans, LA, USA, Jan. 2018. URL https://doi.org/10.1137/1.9781611975031.98.

K. Sadakane and G. Navarro. Fully-functional succinct trees. In *Proceedings of the 21st Annual Symposium on Discrete Algorithms (SODA 2010)*, pages 134–149, Austin, TX, USA, Jan. 2010. URL https://doi.org/10.1137/1.9781611973075.13.

Eidesstattliche Versicherung

Ellert, Jonas	157195
Name, Vorname	Matr.-nr.

Ich versichere hiermit an Eides statt, dass ich die vorliegende Masterarbeit mit dem Titel

Efficient Computation of Nearest Smaller Suffixes

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Dortmund, den 6. Juli 2019

Ort, Datum

Unterschrift

Belehrung:

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/ die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -)

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird gfls. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin“) zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Dortmund, den 6. Juli 2019

Ort, Datum

Unterschrift