2017

Succinct Data Structures for Parameterized Pattern Matching and Related Problems

Arnab Ganguly
Louisiana State University and Agricultural and Mechanical College, ju.arnab@gmail.com

SUCCINCT DATA STRUCTURES FOR PARAMETERIZED PATTERN MATCHING AND RELATED PROBLEMS

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Arnab Ganguly
B.E., Jadavpur University, 2009
August 2017

Dedicated to Maa, Baba, and Kaku

Acknowledgments

First, I express my sincere gratitude to Dr. Rahul Shah, my advisor, for inspiration and continuous support. His brilliant intuition has helped me to solve difficult problems, leading to important results. His constant urge to look into increasingly more challenging problems has proved to be the source of improvement in my research aptitude. Without him, this dissertation would not have been possible.

I thank my co-advisor Dr. Sukhamay Kundu for the guidance during my initial years in LSU. His insightful questions and meticulous approach to writing has been pivotal in the successful completion of this dissertation. Needless to say, I owe a lot to him.

I also thank Dr. Jianhua Chen for agreeing to be on my committee and for her careful reading of this dissertation. A special token of gratitude goes to Sharma Thankachan, a good friend and a faculty member at the University of Central Florida. Sharma, besides being a collaborator on numerous papers, has been crucial in developing my interest and aptitude in the world of pattern matching. Another special mention goes to Dr. Wing-Kai Hon of National Tsing Hua University, for collaborating with me and for explaining important concepts using the simplest of forms.

I feel truly blessed to have my parents and uncle for the wonderful people they are. I thank Sudip, my collaborator, colleague, and friend. I will always cherish the wonderful weekends, the awesome food, and the vacations that I shared with my friends Ananya, Arghya, Ishita, Saikat, Satadru, Sayan, Subhajit, and Trina. A special note of gratitude is reserved for my old friends Anik, Arko, and Vivek for being there through thick-and-thin. Last but not the least, I would like to thank Sujana for putting up with me. She is clearly my best discovery at LSU.

Table of Contents

ACKNOWLEDGMENTS iii

ABSTRACT vi

CHAPTER

1 INTRODUCTION 1
- 1.1 Motivation 2
- 1.2 Contribution 3
- 1.3 Roadmap 10

2 PRELIMINARIES 12
- 2.1 Linear vs Compact vs Succinct 12
- 2.2 Suffix Tree and Suffix Array 13
- 2.3 Burrows-Wheeler Transform and FM-Index 14
- 2.4 Rank and Select on Bit-Vectors 16
- 2.5 Wavelet Tree 17
- 2.6 Succinct Trees with Full Functionality 17
- 2.7 Succinctly Indexable Dictionaries 18

3 SUCCINCT INDEX FOR NON-OVERLAPPING PATTERN MATCHING 19
- 3.1 Overview of Techniques 20
- 3.2 Definitions 21
- 3.3 The Querying Process 21

4 SUCCINCT INDEX FOR PARAMETERIZED PATTERN MATCHING 26
- 4.1 Overview of Techniques 27
- 4.2 Parameterized Suffix Tree 29
- 4.3 Parameterized Burrows-Wheeler Transform 30
- 4.4 Parameterized LF Mapping 33
- 4.5 Implementing Parameterized LF Mapping 35
- 4.6 Finding Suffix Range via Backward Search 42

5 SUCCINCT INDEX FOR PARAMETERIZED DICTIONARY MATCHING 46
- 5.1 Overview of Techniques 46
- 5.2 Idury and Schäffer's Linear Index 48
- 5.3 Representing States Succinctly 49
- 5.4 Handling next Transitions Succinctly 51
- 5.5 Handling failure and report Transitions Succinctly 62
- 5.6 The Final Query Procedure 63

6 COMPACT INDEX FOR ORDER-PRESERVING PATTERN MATCHING 64

6.1 Overview of Techniques . . . 64
6.2 Order-preserving Indexing . . . 67
6.3 LF Successor . . . 69
6.4 Some Useful Definitions . . . 72
6.5 Successor Pair First Disagree Before their LCP . . . 74
6.6 Successor Pair First Disagree After their LCP . . . 75
6.7 Data Structure Toolkit . . . 82
6.8 Wrapping Up . . . 84

7 COMPACT INDEX FOR ORDER-PRESERVING DICTIONARY MATCHING . . . 87
7.1 Overview of Techniques . . . 87
7.2 Linear Space Index . . . 89
7.3 Representing States Succinctly . . . 91
7.4 Handling next Transitions Compactly . . . 91
7.5 Handling failure and report Transitions Succinctly . . . 95
7.6 The Final Query Procedure . . . 95

8 OPEN PROBLEMS . . . 96
8.1 Compact Construction . . . 96
8.2 Compressed Indexes for $2D$ Pattern Matching . . . 96
8.3 Streaming Algorithms . . . 97

REFERENCES . . . 98

VITA . . . 106

Abstract

Let T be a fixed text-string of length n and P be a varying pattern-string of length $|P| \leq n$. Both T and P contain characters from a totally ordered alphabet Σ of size $\sigma \leq n$. Suffix tree is the ubiquitous data structure for answering a pattern matching query: *report all the positions i in T such that $T[i+k-1] = P[k]$, $1 \leq k \leq |P|$*. Compressed data structures support pattern matching queries, using much lesser space than the suffix tree, mainly by relying on a crucial property of the leaves in the tree. Unfortunately, in many suffix tree variants (such as parameterized suffix tree, order-preserving suffix tree, and 2-dimensional suffix tree), this property does not hold. Consequently, compressed representations of these suffix tree variants have been elusive.

We present the first compressed data structures for two important variants of the pattern matching problem: **(1)** *Parameterized Matching* – report a position i in T if $T[i+k-1] = f(P[k])$, $1 \leq k \leq |P|$, for a one-to-one function f that renames the characters in P to the characters in $T[i, i+|P|-1]$, and **(2)** *Order-preserving Matching* – report a position i in T if $T[i+j-1]$ and $T[i+k-1]$ have the same relative order as that of $P[j]$ and $P[k]$, $1 \leq j < k \leq |P|$. For each of these two problems, the existing suffix tree variant requires $\Theta(n \log n)$ bits of space and answers a query in $O(|P| \log \sigma + occ)$ time, where occ is the number of starting positions where a match exists. We present data structures that require $O(n \log \sigma)$ bits of space and answer a query in $O((|P| + occ)\,\mathrm{poly}(\log n))$ time. As a byproduct, we obtain compressed data structures for a few other variants, as well as introduce two new techniques (of independent interest) for designing compressed data structures for pattern matching.

Chapter 1
Introduction

Given a string T (called *text*) and a string P (called *pattern*), the pattern matching problem [Gus97] is to answer the following query: find all starting positions (or simply, occurrences) of substrings of T that match exactly with P. Earlier works [BM77, KR87, KJP77] concentrated on the scenario in which both text and pattern were provided at query time. Needless to say, in this scenario, the text had to be read entirely to report the occurrences. In most cases, however, the text is a long fixed string and patterns are much shorter varying strings. This motivated the development of *full-text indexes* so as to facilitate pattern matching efficiently. More specifically, the objective is to pre-process T and build a data structure (called *index*), such that given a pattern P, all occurrences of P in T can be reported without having to read the entire text T. *Suffix tree* [Ukk95, Wei73] (resp. *Suffix array* along with *Longest Common Prefix array* (LCP array) [MM93]) are the classical full-text indexes supporting pattern matching in time $O(|P|+occ)$ (resp. $O(|P|+\log n+occ)$[1]), where n is the length of T, $|P|$ is the length of P, and occ is the number of occurrences of P in T. Although fast and arguably simple to implement, the suffix tree/array of the text T occupies $\Theta(n)$ words or equivalently $\Theta(n \log n)$ bits[2] of space. This is linear in the number of words (and hence called *linear-space indexes*), but not in number of bits, as T itself can be stored in $n\lceil \log \sigma \rceil$ bits, where σ is the size of the alphabet Σ from which characters in T and P are chosen[3]. Typically $n \gg \sigma$, leading to a complexity gap (when $\log \sigma = o(\log n)$). In fact, suffix trees are too large for most practical purposes. For example, the human genome (where $\sigma = 4$) occupies space $\approx$ 700 MB, whereas the suffix tree of the genome, even with a space-efficient implementation such as in [Kur99], requires space $\approx$ 40 GB. Consequently, the main question was: "*Can we design an index that occupies space close to the size of* T (*i.e.,* $n\lceil \log \sigma \rceil$ *bits*), *and yet supports pattern matching in time*

[1] All logarithms are in base 2.
[2] We use the standard Word-RAM model with poly-logarithmic word size.
[3] Each character in T is from an alphabet of size σ and can be encoded in $\lceil \log \sigma \rceil$ bits.

within poly-logarithmic penalty factors of the suffix tree/array?". Grossi and Vitter [GV00], and Ferragina and Manzini [FM00] answered this question by presenting succinct/compact indexes[4] named *Compressed Suffix Array* (CSA) and *FM-Index* respectively. Subsequently, the field of compressed text indexing[5] was established. We refer the reader to [NM07] for a comprehensive survey on compressed text indexing.

1.1 Motivation

Pattern matching, in its traditional definition, is often not sufficient to capture many real-world applications. This led to the formulation of many variants, such as

1. document indexing [Nav13], where we are given a collection of texts (called documents) and the objective is to report the (*top-k*) documents that contain the pattern,

2. parameterized pattern matching [Bak93], where two strings are a match if one can be transformed to the other by using a one-to-one function that renames the characters in the former string to those in the latter, and

3. order-isomorphic pattern matching [KEF$^+$14], where two strings are a match if the relative order of the characters at any two positions is the same for both the strings.

Following the discovery of the CSA and the FM-Index, the main challenge was to design compressed indexes for other pattern matching variants (such as the ones listed above). On a broad scale, these variants can be categorized based on two types of applications – **Type 1** and **Type 2**. In Type 1 applications (such as document indexing), the underlying key component of the index is often the suffix tree, which is typically augmented with additional information to cater to the specific application. This allows the use of CSA and FM-Index to design space-efficient indexes for these problems, which has led to rich developments [BN14, Fis10, GST15, HPST13, MNN$^+$14, Nav13, NM07, NT13, OGK10, RNO11]. The idea is to

[4] Indexes that occupy space very close to the text itself are called *succinct*, whereas indexes that occupy space within some constant multiplicative factor are called *compact* – a formal definition is in Section 2.1. Often with such indexes, the query time must be within poly-logarithmic factors of the linear space index.

[5] Collectively, we call succinct and compact indexes as *space-efficient or compressed indexes.*

first replace the suffix tree by a compressed text-index (such as FM Index or CSA), and then to represent the augmenting information in minimal space. On the other hand, Type 2 applications require one to develop variants of the suffix tree. Designing space-efficient data structures for such applications has remained elusive throughout the development of compressed text indexes, primarily because these suffix tree variants do not possess some properties of the classical suffix tree that are crucial for compression. Hence, CSA and FM-Index no longer apply.

The motivation behind this dissertation is to design compressed indexes for a large collection of important Type 2 variants of the pattern matching problem, which are not only theoretically intriguing, but also have many practical consequences.

1.2 Contribution

We achieve positive breakthroughs on the following important Type 2 problems – *Parameterized Pattern/Dictionary Matching* [Bak93, IS94], and *Order-preserving Pattern/Dictionary Matching* [CIK$^+$13, KEF$^+$14]. In most cases, other than a box-full of new ideas, we employ fairly complex succinct data structure tricks. Therefore, to aid the reader's intuition in pattern matching terminologies and (succinct) data structures, we start with a much simpler Type 1 problem, known as *Non-overlapping Pattern Matching* [CP09]. For each problem, the space-time complexities of our data structure versus the existing ones are summarized at the end of this section in Table 1.1. Another (and probably the more) important contribution lies in the introduction of two new compressed text indexing techniques –

- We heavily use the topology of the suffix tree variants, which is a balanced parenthesis representation of the tree (for the particular Type 2 application) augmented with other succinct data structures [NS14]. This technique forms the backbone of our compressed data structures for all the Type 2 pattern matching problems.

- Efficiently evaluating the lexicographic rank of a suffix (among all the suffixes of T), given the rank of the immediate next suffix in text-order, has been the single-most

important idea behind all the existing compressed text indexes [NM07]. Traditionally, the approach to this is based on some sort of a reversible transformation of the text, such as the Burrows-Wheeler Transform (BWT) [BW94]. In the Type 2 pattern matching variants, typically a suffix (in the context of the suffix tree variant) is some encoded form of the original suffix [Bak93, CIK$^+$13, Gia93, KEF$^+$14]. In this case, the following property (unlike normal suffixes) may no longer hold: the (encoding of the) suffix starting at position t is a suffix of the (encoding of the) suffix starting at position $t-1$. Consequently, a BWT-like approach no longer works (at least obviously) for the Type-2 variants, necessitating alternate approaches. To this end, we present a new line of attack, which we call *LF Successor*. As an example illustrating the usefulness, we focus on order-preserving matching. Yet the most impressive aspect of LF successor is that it gives us a generic technique for deriving compressed data structures for all the Type 2 pattern matching problems discussed in this dissertation.

We believe that these techniques, besides being of independent interest, can be extended to obtain compressed representations of other suffix tree variants. We now discuss our contributions to each problem separately.

Type 1 Applications

After the advent of compressed text indexing structures in the form of CSA and FM Index, *compressed suffix trees* [RNO11, Sad07] replaced the older suffix tree required in these applications. The focus was now on how to compress the augmenting data in informational theoretic minimal space. One of the first such work was by Sadakane [Sad02b] who showed that given a (compressed) suffix array, the Longest Common Prefix (LCP) array can be maintained in an additional $2n + o(n)$ bits. Fischer [Fis10] improved to this to $o(n)$ bits. The *Range Minimum Query* (RMQ) data structure is an integral component to many applications related to suffix trees/arrays. Fischer and Heun [FH07] showed how to maintain an RMQ index in $2n + o(n)$ bits. In another direction, Sadakane [Sad02a]

considered the problem of document retrieval, and presented succinct indexes. Later, other improvements (see [Nav13] for a survey) culminated in the best-known document listing indexes [NT13, Tsu13]. Other problems such as property matching [HPST13] and matching statistics [OGK10] also fall under the same theme.

1.2.1 Non-overlapping Pattern Matching

The first problem we study (in Chapter 3) is popularly known as *Non-overlapping Pattern Matching* [CP09]. The task is to report a set of the maximum number of non-overlapping occurrences of P in T – two occurrences are non-overlapping if and only if they are separated by at least $|P|$ characters. Reporting such occurrences finds application in fields such as speech recognition, data compression, etc. For instance, one can compress a text by replacing each non-overlapping occurrence of P by a single character from a new alphabet, where each symbol in the new alphabet is the image obtained by applying a hash-function on P. Linear space (in words) and optimal query time index has been known for quite a while [CP09]. We show that any text-index (such as the CSA or the FM-Index or the suffix tree/array) can be used to report non-overlapping occurrences simply by using a different querying algorithm [GST15]. An immediate corollary of our result is a succinct index.

Type 2 Applications

In the remainder of the dissertation (Chapters 4 through 7), we focus on problems catering to Type 2 applications that demand variants of the classical suffix tree. These variants may not follow some of the structural properties of the classical suffix tree. One such structural property, crucial to compressed text indexing, is called *suffix link* (resp. *reverse suffix link*), which has the following *rank-preserving* property – two suffixes having the same first character (resp. preceding character) x retain their relative lexicographic rank when x is truncated (resp. x is prepended). For numerous suffix tree variants [Bak93, CIK$^+$13, Gia93, KKP03, Shi00, Shi10], called *suffix trees with missing suffix links* [CH03], this rank-preserving property does not hold, primarily because instead of us-

ing the suffixes directly, we use some encoded form of them. This brings in new challenges in how to represent these suffix tree variants space-efficiently (akin to CSA and FM-Index), and even 15 years after the introduction of the CSA and the FM-Index, there has not been any significant progress. Also, it has been largely unknown whether succinct/compact data structures are even possible. We develop compressed indexes for numerous problems of this category, thereby answering this long-standing question affirmatively.

1.2.2 Parameterized Pattern Matching

We begin in Chapter 4 with the *Parameterized Pattern Matching* problem [Bak93]. Here, Σ is partitioned into two disjoint sets: Σ_s containing static characters (s-characters) and Σ_p containing parameterized characters (p-characters). Two equal-length strings S and S' are a parameterized match (p-match) iff $S[i] \in \Sigma_s \iff S'[i] \in \Sigma_s$, $S[i] = S'[i]$ when $S[i] \in \Sigma_s$, and there exists a one-to-one function f such that $S[i] = f(S'[i])$ when $S[i], S'[i] \in \Sigma_p$. We are interested in the indexing problem: given a fixed text $T[1, n]$, find the starting positions of all substrings of T that are a p-match with a pattern P.

Baker [Bak93] introduced the parameterized matching concept, with the main motivation being software plagiarism detection. Additionally, they introduced the *Parameterized Suffix Tree* (p-suffix tree) for the indexing problem. The role of the problem and the usefulness of p-suffix trees was presented using a program called *Dup* [Bak95]. Subsequently, the methodology became an integral part of various tools for software version management and clone detection, where identifiers and/or literals are renamed. The state-of-the-art approaches [BYdM$^+$98, KFF06, RD04] to clone detection use a hybrid approach, such as a combination of parse trees, which converts literals into p-characters, and a p-suffix tree on top of these p-characters. Unfortunately, as with traditional suffix trees, the space occupied by p-suffix trees is too large for most practical purposes, bringing in the demand for space-efficient variants. In fact, one of the available tools (CLICS [CLI]) for detecting software clones clearly acknowledges that the major space consumption is due to the suffix tree. Some other tools [HJHC10] are based on the inverted index. Although less space con-

suming, no theoretical guarantees are possible on query-times in these tools. Since p-suffix trees accommodate substring match, they can also be used in recognizing smaller code fragment clones. Following are a few other works that have used p-suffix trees: finding relevant information based on regular expressions in sequence databases [dMRS05, dMRS07], detecting cloned web pages [DLDPFG01], detecting similarities in JAVA sources from bytecodes [BM98], etc. Further generalizations [AAC+03, Shi00] have found applications in computational biology.

Parameterized matching (p-matching) has seen constant development since its inception by Baker [Bak93] in 1993. Baker presented a $\Theta(n \log n)$-bit index, known as the *Parameterized Suffix Tree*, that can find all *occ* occurrences in $O(|P| \log \sigma + occ)$ time. The construction time was $O(n\sigma)$ in the worst-case, which was later improved to $O(n \log \sigma)$ by Kosaraju [Kos95]. Later, Cole and Hariharan [CH03] discovered an $O(n)$ time randomized construction algorithm. Amir, Farach-Colton, and Muthukrishnan [AFM94] presented an algorithm for the scenario when both T and P are provided at query time. Shibuya [Shi00] considered additional requirements for applications in RNA structural matching. Idury and Schäffer considered the parameterized dictionary matching problem [IS94]. Ganguly et al. [GHS+16a, GHS16b] presented compressed indexes for the dictionary problem and for its dynamic variant. Amir et al. [AAC+03] considered the two-dimensional p-matching problem. Hazay, Lewenstein, and Sokol introduced approximate version of the problem [HLS04]. Amir and Navarro [AN09] gave an algorithm for p-matching on non-linear structures. Jalsenius, Porat and Sach presented a solution for p-matching in the streaming model [JPS13]. We refer the reader to [Lew15, MP15] for related surveys.

Although there have been attempts at solving p-pattern matching in compressed space, such as in [BA13, HSSY10], none of these are truly "efficient", i.e., they do not match the querying time of the p-suffix tree within poly-logarithmic penalty factors. We present the first succinct index [GST17] for this problem with provable poly-logarithmic penalty guarantees. At the core of our index, lies a *new* Burrows-Wheeler-like transform, which we

call the *Parameterized Burrows-Wheeler Transform*. Our major contribution is the use of the p-suffix tree topology to obtain the succinct index – a novel approach in compressed text indexing. This technique proves invaluable in obtaining succinct/compact indexes for all of the remaining problems in this dissertation. Additionally, we remark that the techniques can be easily modified to obtain a succinct representation of the structural suffix tree of Shibuya [Shi00], with similar guarantees.

1.2.3 Parameterized Dictionary Matching

In the parameterized dictionary indexing problem, we are given a collection of (fixed) patterns $\{P_1, P_2, \ldots, P_d\}$ of total length n characters. The task is to pre-process them and create a data structure such that given a text T, we can find all pairs $\langle i, j \rangle$, where P_i is a parameterized match with $T[j, j - |P_i| + 1]$. Idury and Schäffer [IS94] presented a $\Theta(n \log n)$-bit data structure that offers a query time of $O(|T| \log \sigma + occ)$, where occ is the number of pairs to be reported. Recently, Ganguly et al. [GHS$^+$16a] presented an $O(n \log \sigma + d \log n)$-bit index with $O(|T|(\log \sigma + \log_\sigma n) + occ)$ query time (see [GHS16b] for its dynamic version). By employing a reversible transform that closely resembles the xBWT of Ferragina et al. [FLMM09] and the topology of the automaton of Idury and Schäffer [IS94], we present a succinct data structure [GST17], which matches the time complexity of Idury and Schäffer [IS94], but uses much lesser space. This also improves the result in [GHS$^+$16a], both in time and space.

1.2.4 Order-preserving Pattern Matching

Both CSA [GV05] and FM-Index [FM05] have relied (solely) on some sort of a reversible transformation of the text (for e.g., the Burrow-Wheeler Transform [BW94]), coupled with their key components: Ψ function of the CSA (which simulates suffix links) and LF mapping of the FM-index (which simulates reverse suffix links). Similar remarks hold for the succinct index for the parameterized pattern matching problem discussed before [GST17]. Unfortunately, the techniques do not seem to generalize to other variants, such as order-preserving pattern matching [KEF$^+$14], leading us to the following question. "*Can we*

represent suffix trees with missing suffix links in $o(n \log n)$ bits of space and answer queries in $O((|P| + occ)\,\mathrm{polylog}(n))$ time when $\log \sigma = o(\log n)$?". We make the first progress towards answering this question by presenting compressed data structures for some instances.

Our focus is on order-preserving matching – two strings S and S' over an integer alphabet are order-preserving iff $|S| = |S'|$, and for any two positions $i, j \in [1, |S|]$, the relative order of $S[i]$ and $S[j]$ is the same as that of $S'[i]$ and $S'[j]$. Although recently introduced, the problem (and its variants) has received significant attention [CGT15, CT14, CNPS15, CIK$^+$13, GM16, GHS$^+$16a, GU16, HKC$^+$15, KEF$^+$14], not only due to its simplistic formulation, but also due to applications in music analysis and analyzing stock prices, where the relative ordering of characters has to be matched. However, limited progress has been made towards a space-efficient data structure for the indexing problem: given a text $T[1, n]$ over an integer alphabet $\Sigma = \{1, 2, \ldots, \sigma\}$, find the occurrences of all substrings of T that are order-preserving with a pattern P. The existing $\Theta(n \log n)$-bit index [CIK$^+$13], called the *order-preserving suffix tree*, can report all the *occ* occurrences in time $O(|P| \log \sigma + occ)$. The only known data structure for this problem that occupies $o(n \log n)$ bits (when $\log \sigma = o(\log n)$) is by Gagie and Manzini [GM16]. However, it can only answer queries when $|P| = O(\mathrm{polylog}\, n)$. Moreover, it can only detect whether there is an occurrence or not, and returns an arbitrary occurrence if there is one. The space and time complexities are $O(n \log \log n)$ bits (in addition to the space for storing T) and $O(|P| \log^4 n)$. We present the first compact data structure for the problem.

As opposed to the traditional reliance on reversible transforms, we employ a new line of attack for implementing LF mapping. We call this *LF Successor*. Once LF successor is implemented, standard techniques lead to LF mapping and the compact index. We believe that our data structure for implementing LF successor is simple; the difficulty lies in proving some of the key properties of order-preserving matching. The surprising aspect of LF successor is its generality. Specifically, LF successor implementation for order-preserving matching can be easily modified (in fact, simplified) to handle the previously

introduced parameterized pattern/dictionary matching problems [Bak93, IS94] and the structural pattern matching problem [Shi00]. Unfortunately, the results are asymptotically inferior to those attained via the reversible transform based techniques. Nevertheless, we hope that our techniques can be generalized and/or improved to obtain compressed representations of other suffix trees with missing suffix links, and the space can be improved to attain the holy-grail of succinct indexes.

1.2.5 Order-preserving Dictionary Matching

In the order-preserving dictionary indexing problem [KEF+14], we are given a collection of (fixed) patterns $\{P_1, P_2, \ldots, P_d\}$ of total length n characters. The task is to pre-process them and create a data structure such that given a text T, we can find all pairs $\langle i, j \rangle$, where P_i is an order-preserving match with $T[j, j - |P_i| + 1]$. Kim et al. [KEF+14] presented a $\Theta(n \log n)$-bit data structure that offers a query time of $O(|T| \log \sigma + occ)$, where occ is the number of such pairs. Ganguly et al. [GHS+16a] presented an $O(n \log \sigma + d \log n)$-bit index with $O(|T| \log n + occ)$ query time, which is based on a sparisification technique of Hon et al. [HLS+08]. By slightly modifying the LF-successor approach for the order-preserving pattern matching problem, we design a compact index for this problem and improve the above results in terms of space occupied.

1.3 Roadmap

The rest of the dissertation is organized as follows. In Chapter 2, we revisit some standard pattern matching terminologies and data structures. Chapter 3 focuses on non-overlapping pattern matching. Chapter 4 introduces the parameterized pattern matching problem and our succinct index for this problem. Chapter 5 discusses our succinct index for the parameterized dictionary matching problem. Chapter 6 presents our compact index for order-preserving pattern matching. Chapter 7 discusses the compact index for order-preserving dictionary matching. We conclude by presenting some open problems in Chapter 8.

Table 1.1: Our Contribution

Problem	**Previous Result**	**Our Result**
Non-overlapping	**Time:** $O(\|P\| + occ)$	**Time:** $O(\|P\| + occ \cdot \log^{1+\epsilon} n)$, $\epsilon \in (0, 1]$
Pattern Matching	**Space:** $\Theta(n \log n)$ bits	**Space:** $n \log \sigma + o(n \log \sigma)$ bits
Parameterized	**Time:** $O(\|P\| \log \sigma + occ)$	**Time:** $O(\|P\| \log \sigma + occ \cdot \log \sigma \log n)$
Pattern Matching	**Space:** $\Theta(n \log n)$ bits	**Space:** $n \log \sigma + O(n)$ bits
Parameterized	**Time:** $O(\|T\| \log \sigma + occ)$	**Time:** $O(\|T\| \log \sigma + occ)$
Dictionary Matching	**Space:** $\Theta(n \log n)$ bits	**Space:** $n \log \sigma + O(n + d \log \frac{n}{d})$ bits
Order-preserving	**Time:** $O(\|P\| \log \sigma + occ)$	**Time:** $O(\|P\| \log \sigma \log \log n + (\log \log n + occ) \log n \log_\sigma n)$
Pattern Matching	**Space:** $\Theta(n \log n)$ bits	**Space:** $O(n \log \sigma)$ bits
Order-preserving	**Time:** $O(\|T\| \log \sigma + occ)$	**Time:** $O(\|T\| \log n + occ)$
Dictionary Matching	**Space:** $\Theta(n \log n)$ bits	**Space:** $O(n \log \sigma + d \log \frac{n}{d})$ bits

Chapter 2
Preliminaries

We refer the reader to [Gus97] for standard definitions and terminologies. We employ the standard Word-RAM model of computation with poly-logarithmic word size and unit cost for simple CPU operations and memory access. Let T be a string (called text) having n characters and P be a string (called pattern) having $|P|$ characters. The characters in T and P are chosen from a totally ordered alphabet Σ having σ characters. Thus, the space occupied by the text is $n\lceil \log \sigma \rceil$ bits. We assume, without loss of generality, that $\sigma \leq n$. Also, assume that T terminates in a unique special character \$. Let $T[i, j]$ be the substring of T from i to j (both inclusive) and $T[i]$ be the ith character of T. Further, T_i is the circular suffix that starts at i. Specifically, $T_i = T$ if $i = 1$; otherwise, $T_i = T[i, n] \circ T[1, i-1]$, where $\circ$ denotes concatenation. Lastly, ϵ is an arbitrarily small positive constant.

The pattern matching problem asks to answer the following query: *report the starting positions (occurrences) i of all substrings of T such that $T[i + k - 1] = P[k]$, $1 \leq k \leq |P|$.* In the indexing problem, the text T is fixed, and the objective is to pre-process it and then create a data structure, such that we can answer the above query without having to read T entirely. We now present some useful definitions and discuss some key data structures that are pivotal in (compressed) text indexing. For dictionary matching, we leave the burden of definitions to the respective chapters.

2.1 Linear vs Compact vs Succinct

An index of T is a data structure that allows efficient pattern matching queries on T. Here, efficient means that the time to report all *occ* occurrences is $O((|P| + occ)\,\mathrm{polylog}(n))$. An index is *linear* if it occupies $\Theta(n \log n)$ bits (or equivalently, $\Theta(n)$ words), *compact* if it occupies $\Theta(n \log \sigma)$ bits, and *succinct* if it occupies $n \log \sigma + o(n \log \sigma) + O(n)$ bits[1].

[1] We deviate, albeit very slightly, from the original definition of a succinct index in which the space is $n \log \sigma + o(n \log \sigma)$ bits. If we wish to stick to the standard definition, for parameterized pattern matching, assume $\sigma = \omega(1)$. Consequently, the aberration does not violate the original definition as $n = o(n \log \sigma)$. In the case of a constant σ, we first create all possible copies of P, where each copy is obtained by replacing

2.2 Suffix Tree and Suffix Array

A suffix tree, denoted by ST, is a compact trie that stores all the (non-empty) suffixes of T. Leaves in the suffix tree are numbered in the lexicographic order of the suffix they represent, and each edge in ST is labeled by a substring of T. For any node u in the suffix tree, the first character on each edge from u to its child (if any) is unique. Let $\mathsf{path}(u)$ be the string formed by concatenating the edge labels from the root to u; let $\mathsf{strDepth}(u) = |\mathsf{path}(u)|$. The locus of a pattern P, denoted by $\mathsf{locus}(P)$, is the highest node u such that P is a prefix of $\mathsf{path}(u)$. The suffix range of P is denoted by $[sp, ep]$, where sp (resp. ep) is the leftmost (resp. rightmost) leaf in the subtree of ST rooted at the locus of P. (See Figure 2.1 for an illustration.) Usually, each node is equipped with perfect hashing [FKS84] such that given a character x, we can find the outgoing edge whose label begins with x in $O(1)$ time. Then, the locus node (or equivalently, the suffix range) of a pattern P is computed in time $O(|P|)$. Without hashing, the suffix range can be computed in time $O(|P| \log \sigma)$ via a binary search on the first character of the outgoing edges (recall that leaves are arranged in lexicographic order of the suffixes; hence edges are also arranged in lexicographic order).

The suffix array, denoted by SA, is an array of length n that maintains the lexicographic arrangement of all the suffixes of T. More specifically, if the ith smallest suffix of T starts at j, then $\mathsf{SA}[i] = j$ and $\mathsf{SA}^{-1}[j] = i$. The former is referred to as the *suffix array value* and the latter as the *inverse suffix array value*. (See Table 2.1 for an illustration.) The suffix value $\mathsf{SA}[\cdot]$ and the inverse suffix value $\mathsf{SA}^{-1}[\cdot]$ can be found in constant time.

Thus, given the suffix tree and suffix array combination, we can find all occ occurrences of a pattern P in $O(|P| + occ)$ time – first find the suffix range $[sp, ep]$ of P using the suffix tree, and then decode $\mathsf{SA}[i]$ for every $i \in [sp, ep]$ to report the occurrences. The suffix tree

the symbols in P with a subset of the symbols in Σ in a one-to-one fashion. Now, for each copy issue an exact pattern matching query on any traditional compressed index [NM07], and get the answers. Since the copies are basically all possible parameterized matches of P to strings over Σ, we will get the desired occurrences. Note that the number of copies is at most $\sigma!$. Hence, the query time is affected by a $\sigma!$ multiplicative factor, which is still a constant, resulting in succinct indexes with slightly different time complexities (as compared to our results for these problems). Hence, we stick to the modified definition and exclude ourselves from dealing with this rather "boring" case. For order-isomorphic matching, we present a compact index; therefore, the succinct definition is irrelevant.

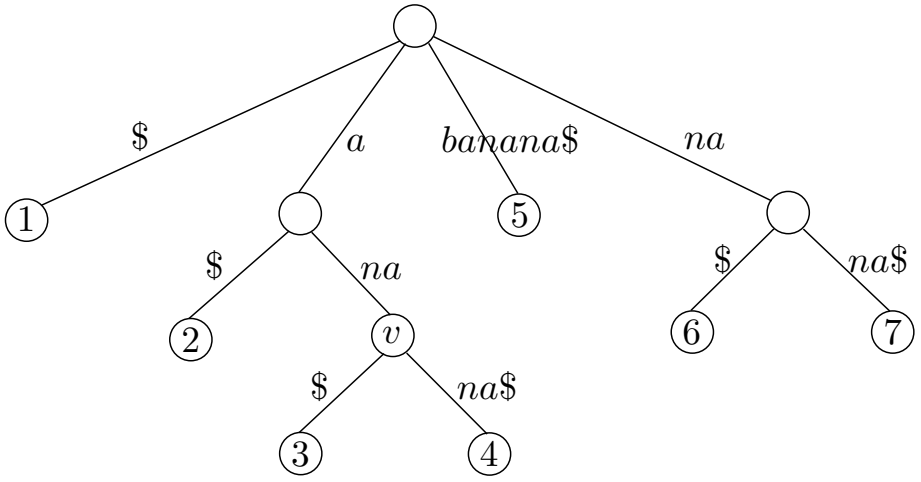

Figure 2.1: Suffix Tree for text *banana*\$. We assume $\$ \prec a \prec b \prec n$, where $\prec$ denotes the total lexicographic order on the alphabet set $\{\$, a, b, n\}$. Here, $\mathsf{locus}(an) = v$ and $\mathsf{path}(v) = ana$. Suffix range of an is $[3, 4]$.

contains n leaves (one per each suffix) and at most $(n-1)$ internal nodes. The suffix array is a permutation on n. Therefore, the space required by these data structures is $\Theta(n \log n)$ bits, or equivalently $\Theta(n)$ words.

2.3 Burrows-Wheeler Transform and FM-Index

Compressed Suffix Arrays/FM Index reduce the space occupancy of suffix trees/arrays from $\Theta(n \log n)$ bits to $O(n \log \sigma)$ bits (or close to the size of the text) with a slowdown in query time. Using these data structures, for any pattern P, we can find all occ occurrences in time $O((|P| + occ) \log^c n)$ for some constant $c > 0$. We present a brief outline of the FM Index (see [FM05] for more details).

Burrows and Wheeler [BW94] introduced a reversible transformation of the text, known as the Burrows-Wheeler Transform (BWT). Recall that T_x is the circular suffix starting at position x. Then, BWT of T is obtained as follows: first create a conceptual matrix M, such that each row of M corresponds to a unique circular suffix, and then lexicographically sort all rows. Thus the ith row in M is given by $T_{\mathsf{SA}[i]}$. The BWT of the text T is the last column L of M, i.e., $\mathsf{BWT}[i] = T_{\mathsf{SA}[i]}[n]$. (See Table 2.1 for an illustration.) Note that BWT is essentially a permutation of T.

Table 2.1: Here the text is $T[1,7] = banana\$$, where $\Sigma = \{a, b, n\}$. The total lexicographic order on Σ is $\$ \prec a \prec b \prec n$.

i	T_i	$T_{\mathsf{SA}[i]}$ (Matrix M)	$\mathsf{SA}[i]$	$\mathsf{SA}^{-1}[i]$	$\mathsf{BWT}[i] = T[\mathsf{SA}[i]-1]$	$\mathsf{LF}(i)$
1	banana\$	\$banana	7	5	a	2
2	anana\$b	a\$banan	6	4	n	6
3	nana\$ba	ana\$ban	4	7	n	7
4	ana\$ban	anana\$b	2	3	b	5
5	na\$bana	banana\$	1	6	\$	1
6	a\$banan	na\$bana	5	2	a	3
7	\$banana	nana\$ba	3	1	a	4

2.3.1 Last-to-First Column Mapping

The underlying principle that enables pattern matching using an FM-Index [FM00] is the last-to-first column mapping (in short, LF mapping). For any $i \in [1, n]$, $\mathsf{LF}(i)$ is the row j in the matrix M where $\mathsf{BWT}[i]$ appears as the first character in $T_{\mathsf{SA}[j]}$. Specifically, $\mathsf{LF}(i) = \mathsf{SA}^{-1}[\mathsf{SA}[i]-1]$, where $\mathsf{SA}[0] = \mathsf{SA}[n]$. Once the BWT is obtained, $\mathsf{LF}(i)$ for any suffix i is computed as:

$$\mathsf{LF}(i) = \mathsf{count}(1, n, 1, \mathsf{BWT}[i]-1) + \mathsf{count}(1, i, \mathsf{BWT}[i], \mathsf{BWT}[i])$$

Here, $\mathsf{count}(i, j, x, y)$ counts the number of positions $k \in [i, j]$ that satisfy $x \leq \mathsf{BWT}[k] \leq y$. By maintaining a Wavelet-Tree [GGV03] over $\mathsf{BWT}[1, n]$ in $n \log \sigma + o(n)$ bits, $\mathsf{count}(i, j, x, y)$ is computed in $O(\log \sigma)$ time (see Fact 2.3 for more details). Hence, we can implement LF mapping in the same space-and-time bounds.

2.3.2 Simulating Suffix Array via LF Mapping

We can decode $\mathsf{SA}[i]$ in $O(\log^{1+\epsilon} n)$ time by using LF mapping and by maintaining a sampled-suffix array, which occupies $o(n \log \sigma)$ bits in total. The idea is to explicitly store $\mathsf{SA}[i]$ iff $\mathsf{SA}[i] \in \{1, 1+\Delta, 1+2\Delta, \dots\}$, where $\Delta = \lceil \log_\sigma n \log^\epsilon n \rceil$. The space needed is $O(\frac{n}{\Delta} \log n) = o(n \log \sigma)$ bits. Then, $\mathsf{SA}[i]$ can be obtained directly if the value has been

Algorithm 1 Backward Search

```
1: procedure backwardSearch(P[1, p])
2:     c ← P[p], i ← p
3:     sp ← 1 + count(1, n, 1, c − 1), ep ← count(1, n, 1, c)
4:     while (sp ≤ ep and i ≥ 2) do
5:         c ← P[i − 1]
6:         sp ← 1 + count(1, n, 1, c − 1) + count(1, sp − 1, c, c)
7:         ep ← count(1, n, 1, c − 1) + count(1, ep, c, c)
8:     end while
9:     if (sp < ep) then "no match found" else return [sp, ep]
10: end procedure
```

explicitly stored; otherwise, it can be computed via at most Δ number of LF mapping operations in time $O(\Delta \cdot \log \sigma) = O(\log^{1+\epsilon} n)$.

2.3.3 Backward Search

Ferragina and Manzini [FM00] showed that using LF mapping, the suffix range $[sp, ep]$ of a pattern $P[1, p]$ can be found by reading P starting from the last character. Specifically, for $i > 1$, suppose the suffix range of $P[i, p]$ is known. Then, the suffix range of $P[i-1, p]$ can be obtained using LF mapping; see Algorithm 1 for details.

Once the suffix range $[sp, ep]$ of P is known, each $i \in [sp, ep]$ corresponds to an occurrence of P in T. Each occurrence can be reported in time $O(\log^{1+\epsilon} n)$ as discussed before. Therefore, we arrive at the following well-known result.

Fact 2.1 ([FM05, GV05]). *By using an $n \log \sigma + o(n \log \sigma)$-bit index of T, we can find all occurrences of P in T in $O(|P| \log \sigma + occ \cdot \log^{1+\epsilon} n)$ time.*

2.4 Rank and Select on Bit-Vectors

Fact 2.2 ([Mun96]). *Let $B[1, m]$ be a bit-vector. By maintaining an $o(m)$-bit data structure, we can find the answer to the following queries in $O(1)$ time:*

(a) $\mathsf{rank}_B(i, x)$ *= the number of occurrences of x in $B[1, i]$.*

(b) $\mathsf{select}_B(k, x)$ *= the minimum position $i \in [1, m]$ such that $\mathsf{rank}_B(i, x) = k$.*

(c) $\mathsf{rank}_B(i, j, x)$ *= the number of occurrences of x in $B[i, j]$.*

(d) $\mathsf{select}_B(i, k, x)$ *= the minimum position* j *in* $[i, m]$ *such that* $\mathsf{rank}_B(i, j, x) = k$.

We drop the subscript B *when the context is clear.*

2.5 Wavelet Tree

Fact 2.3 ([FM05, GGG+07, GGV03, Nav14])**.** *The wavelet tree (WT) data structure generalizes the* rank *and* select *queries over bit-vectors. Specifically, given an array* $A[1, m]$ *over an alphabet* Σ *of size* σ*, by using a data structure of size* $m \log \sigma + o(m)$ *bits, the following queries can be answered in* $O(1 + \frac{\log \sigma}{\log \log m})$ *time:*

(a) $A[i]$.

(b) $\mathsf{rank}_A(i, x)$ *= the number of occurrences of* x *in* $A[1, i]$.

(c) $\mathsf{select}_A(k, x)$ *= the minimum position* i *such that* $\mathsf{rank}_A(i, x) = k$.

(d) $\mathsf{count}_A(i, j, x, y)$ *= the number of positions* $k \in [i, j]$ *such that* $x \leq A[k] \leq y$.

Additionally, the following queries can be answered in $O(\log \sigma)$ *time:*

(a) $\mathsf{predecessor}_A(i, W)$ *= rightmost position* $j < i$ *such that* $A[j] \leq W$.

(b) $\mathsf{prevVal}_A(L, R)$ *= rightmost position* $j \in [L, R)$ *such that* $A[j]$ *equals the maximum value in* $A[L, R-1]$ *that is at most* $A[R]$.

(c) $\mathsf{nextVal}_A(L, R)$ *= rightmost position* $j \in [L, R)$ *such that* $A[j]$ *equals the minimum value in* $A[L, R-1]$ *that is at least* $A[R]$.

We drop the subscript A *when the context is clear.*

2.6 Succinct Trees with Full Functionality

Fact 2.4 ([NS14])**.** *A tree having* m *nodes can be represented in* $2m + o(m)$ *bits, such that if each node is labeled by its pre-order rank, we can compute the following in* $O(1)$ *time:*

(a) $\mathsf{pre\text{-}order}(u)/\mathsf{post\text{-}order}(u)$ *= pre-order/post-order rank of node* u.

(b) $\mathsf{parent}(u)$ *= the parent of node* u.

(c) $\mathsf{nodeDepth}(u)$ *= the number of edges on the path from the root to* u.

(d) $\mathsf{child}(u, q)$ = *the* q*th leftmost child of node* u.

(e) $\mathsf{sibRank}(u)$ = *number of children of* $\mathsf{parent}(u)$ *to the left of* u.

(f) $\mathsf{lca}(u, v)$ = *the lowest common ancestor (LCA) of two nodes* u *and* v.

(g) $\mathsf{lmostLeaf}(u)$/$\mathsf{rmostLeaf}(u)$ = *the leftmost/rightmost leaf in the subtree rooted at* u.

(h) $\mathsf{levelAncestor}(u, D)$ = *the ancestor of* u *such that* $\mathsf{nodeDepth}(u) = D$.

(i) the pre-order rank of the i*th leftmost leaf*

(j) $\mathsf{leafNumber}(\ell)$ = *the number of leaves that lie to the left of the leaf* ℓ.

2.7 Succinctly Indexable Dictionaries

Fact 2.5 ([RRS07])**.** *A set* $\mathcal{S}$ *of* k *integer keys from a universe of size* U *can be stored in* $k \log(U/k) + O(k)$ *bits of space to support the following two operations in* $O(1)$ *time:*

(a) return the key of rank i *in the natural order of integers.*

(b) If $j \in \mathcal{S}$*, return the rank of key* j *in the natural order of integers, else return* -1.

Chapter 3
Succinct Index for Non-overlapping Pattern Matching

We begin by formally defining the *Non-overlapping Indexing* problem.

Problem 3.1 (Non-overlapping Indexing)**.** *Two occurrences of a pattern P in a text $T[1, n]$ are non-overlapping iff they are separated by at least $|P|$ positions, i.e., for two occurrences t and t', $|t - t'| \geq |P|$. The task is to index T such that we can efficiently report a set containing the maximum number of non-overlapping occurrences of P in T.*

For example, if $T = ababaxyaba$ and $P = aba$, then we have to report the position 8 and either 1 or 3 (but not both). We observe that there can be multiple sets of maximum non-overlapping occurrences. Our objective is to report any one set.

Cohen and Porat [CP09] presented the first optimal time solution to this problem. Their index, consisting of a suffix tree of T and an additional $O(n)$-word data structure, can report all the *nocc* non-overlapping occurrences in time $O(|P| + nocc)$. However, it was left unanswered, whether Problem 3.1 can be handled in succinct (or, compact) space, or not. We answer this affirmatively by showing that the problem can be solved efficiently using any index of T alone, as summarized in the following theorem.

Theorem 3.1. *Let* CSA *be a full-text index of T. Using* CSA*, let (i) $\mathsf{search}(P) = \Omega(|P|)$ be the time in which we can compute the suffix range of P, and (ii) $\mathsf{t_{SA}} = \Omega(1)$ be the time in which we can compute a suffix array or inverse suffix array value. By using* CSA *alone, we can find a set containing the maximum number, say nocc, of non-overlapping occurrences of P in time $O(\mathsf{search}(P) + nocc \cdot (\mathsf{t_{SA}} + \log nocc))$.*

Thus, Problem 3.1 can be solved using any text-index (succinct, compact, or linear space) for the traditional problem of reporting all occurrences. Furthermore, by avoiding the use of any additional data structures, we ensure that various space and time trade-offs

can be easily obtained. For example, if we use a suffix tree, $O(|P| + nocc \log nocc)$ time can be obtained. This is very similar to the result by Cohen and Porat [CP09], and in fact, by using some additional $O(n \log n)$-bit data structures (used by Cohen and Porat as well), the query time can be improved to optimal $O(|P| + nocc)$. On the other hand, an $n \log \sigma + o(n \log \sigma)$-bit and $O(|P| + nocc \cdot \log^{1+\epsilon} n)$ time index can be obtained by using the compressed suffix array of Belazzougui and Navarro [BN14]; note that $nocc \leq n$. Recall that σ is the size of the alphabet and $\epsilon > 0$ is an arbitrary small constant.

3.1 Overview of Techniques

The non-overlapping occurrences can be found as follows: find all *occ* occurrences in sorted order, report the last occurrence, then perform a right to left scan of the occurrences, and report an occurrence if it is at least $|P|$ characters away from the latest reported occurrence.

Consider the text in Figure 3.1. The occurrences (in sorted order) of the pattern $P = aba$ are the positions $4, 9, 11, 13, 21, 23$. Following the procedure above, we first report 23. Then, skip 21, report 13, skip 11, and report 9. Finally, we report 4 and terminate.

The complexity of this procedure is $O(\mathsf{search}(P) + occ \cdot \mathsf{t_{SA}} + occ \cdot \log occ)$; the first two factors are for finding all the occurrences and the third one is for sorting the occurrences. The idea behind reducing the complexity to that claimed in Theorem 3.1 is to consider at most *nocc* occurrences (instead of *occ* occurrences) initially.

We break down the *occ* occurrences into maximal disjoint chains of occurrences – each successive occurrence in a chain are regularly separated, say by $x < |P|$ positions, but the first and last occurrence of two successive chains are separated by at least $|P|$ positions. We repeat the following steps for each chain. Start from the rightmost occurrence in a chain, and report it. Use x and $|P|$ to find the closest occurrence which is at least $|P|$ characters to the left of the latest reported one. Now, we report this previous occurrence and repeat until the entire chain is consumed.

It is not too hard to see that if we consider $O(nocc)$ chains initially and spend $O(\mathsf{t_{SA}})$ time in reporting each non-overlapping occurrence in a chain, we are done. This constitutes

our strategy – find the chains and then query each chain; Section 3.3 contains the details. We begin with a few basic ingredients in Section 3.2.

3.2 Definitions

In what follows, we use CSA to denote a full-text index of T (not necessarily a compressed index). Using CSA, let $\mathsf{search}(P) = \Omega(|P|)$ be the time in which we can compute the suffix range of P, and $\mathsf{t_{SA}} = \Omega(1)$ be the time in which we can compute a suffix array or inverse suffix array value. We assume that $\mathsf{search}(P)$ is proportional to $|P|$.

Lemma 3.1. *Given the suffix range $[sp, ep]$ of pattern P, using CSA, we can verify in time $O(\mathsf{t_{SA}})$ whether P occurs at a text-position t, or not.*

Proof. The lexicographic position ℓ of the suffix $T[t, n]$ (i.e., $\mathsf{SA}^{-1}[t]$) can be found in $O(\mathsf{t_{SA}})$ time. The lemma follows by observing that P occurs at t iff $sp \leq \ell \leq ep$. ■

Definition 3.1 (Period of a Pattern)**.** *The period of a pattern P is its shortest non-empty prefix Q, such that P can be written as the concatenation of several (say $\alpha > 0$) number of copies of Q and a (possibly empty) prefix Q' of Q. Specifically, $P = Q^{\alpha}Q'$.*

The period of P can be computed $O(|P|)$ time using the failure function of the KMP algorithm [Gus97, KJP77].

For example, if $P = abcabcab$, then $Q = abc$, $\alpha = 2$, and $Q' = ab$. If $P = aaa$, then $Q = a$, $\alpha = 3$, and Q' is empty. If $P = abc$, then $Q = abc$, $\alpha = 1$, and Q' is empty.

The following is an important observation related to periodicity of strings.

Observation 3.1. *Two occurrences of a pattern $P = Q^{\alpha}Q'$ are separated by at least $|Q|$ characters, i.e., if t is an occurrence, then the closest occurrence can be at $t \pm |Q|$.*

3.3 The Querying Process

In this section, we present our solution to Problem 3.1. Moving forward, assume that P has been decomposed as $Q^{\alpha}Q'$, which can be computed in $O(|P|)$ time using the failure function of the KMP algorithm [Gus97, KJP77]. We also assume that the suffix range of P has been computed in time $\mathsf{search}(P) = \Omega(|P|)$. We consider the following two cases.

3.3.1 Aperiodic Patterns

If P does not occur in T, Problem 3.1 can be trivially answered using CSA in $\mathsf{search}(P)$ time. Also, observe that P can overlap itself iff there is a proper suffix of P which is also its (proper) prefix; in this case, Q is a proper prefix of $P = Q^{\alpha}Q'$. If this condition does not hold (i.e., if $Q = P$), which can be verified in $O(|P|)$ time using the KMP algorithm [Gus97, KJP77], then the desired non-overlapping occurrences are simply all the occurrences of P (i.e., $nocc = occ$), which can be found in time $O(\mathsf{search}(P) + nocc \cdot \mathsf{t_{SA}})$.

Now, we consider the case when $\alpha = 1$ (i.e., $P = QQ'$) and Q' is not empty. In other words, P has a proper prefix which is also its proper suffix. In this case, note that $occ \leq 2nocc$. Therefore, we can simply find all occurrences using CSA in $O(\mathsf{search}(P) + occ \cdot \mathsf{t_{SA}})$ time. Then sort the occurrences and use the naive algorithm at the beginning of Section 3.1 to find the a set containing the desired $nocc$ number of non-overlapping occurrences in $O(\mathsf{search}(P) + nocc \cdot (\mathsf{t_{SA}} + \log nocc))$ time. Summarizing, we get the following lemma

Lemma 3.2. *Given a pattern P, we can check if it has any occurrence in T in $O(\mathsf{search}(P))$ time. If there is no proper suffix of P which is also its proper prefix, then we can find all the non-overlapping occurrences in $O(\mathsf{search}(P) + nocc \cdot \mathsf{t_{SA}})$ time. Lastly, if $P = QQ'$ and Q' is not empty, then we can find all the non-overlapping occurrences in $O(\mathsf{search}(P) + nocc \cdot (\mathsf{t_{SA}} + \log nocc))$ time.*

3.3.2 Periodic Patterns

In this case, $P = Q^{\alpha}Q'$, where $\alpha \geq 2$. We begin with the following definitions.

Definition 3.2 (Critical Occurrence)**.** *A position t_c in T is called a critical occurrence of $P = Q^{\alpha}Q'$, where $\alpha \geq 2$, iff t_c is an occurrence of P but the position $t_c + |Q|$ is not.*

Definition 3.3 (Range of a Critical Occurrence)**.** *Let t_c be a critical occurrence of P in T. Let $t' \leq t_c$ be the maximal position such that $t', t' + |Q|, t' + 2|Q|, \ldots, t_c$ are occurrences of P but the position $t' - |Q|$ is not. The range of t_c is $\mathsf{range}(t_c) = [t', t_c + |P| - 1]$.*

For example, let the text $T[1, 18]$ be $xyzabcabcabcabxyx\$$. Then $t_c = 7$ is a critical occurrence of $P = abcabcab$, but $t_c = 4$ is not. Also, $\mathsf{range}(7) = [4, 14]$.

Following are some crucial observations.

Observation 3.2. *Let t_c be a critical occurrence of P. Then, t_c is the rightmost occurrence of P in $\mathsf{range}(t_c)$. Furthermore, the ranges of two critical occurrences are disjoint.*

Observation 3.3. *A critical occurrence of P in T corresponds to at least one non-overlapping occurrence, i.e., nocc is at least the number of critical occurrences of P in T.*

It follows from Observations 3.2 and 3.3 that to find the desired non-overlapping occurrences of P in T, it suffices to find the maximum number of non-overlapping occurrences of P in the range of every critical occurrence. Clearly the following two components suffice – (i) an algorithm that finds the maximum number of non-overlapping occurrences of P in the range of a single critical occurrence, and (ii) an algorithm that can find all critical occurrences of P. The first component is met by Lemma 3.3 and the second by Lemma 3.4.

Lemma 3.3. *Given a pattern $P = Q^{\alpha}Q'$ in the form $\langle |Q|, \alpha, |Q'| \rangle$ for some $\alpha \geq 2$, and the suffix range of P, we can find a set of the maximum number of non-overlapping occurrences of P in $\mathsf{range}(t_c)$ in time $O(nocc' \cdot \mathsf{t_{SA}})$, where $nocc'$ is the size of the set and t_c is a critical occurrence of P.*

Proof. The proof is immediate from the following steps. See Figure 3.1 for an illustration.

1. Report t_c as a non-overlapping occurrence of P.

2. Let $t = t_c - \alpha|Q|$. If $t \leq 0$ or if P does not appear at t (which can be verified in $\mathsf{t_{SA}}$ time using Lemma 3.1), then terminate. Otherwise, t belongs to $\mathsf{range}(t_c)$. If Q' is empty, then let $t' = t - |Q|$, else $t' = t$.

3. If $t' \leq 0$ or if P does not appear at t', then terminate. Otherwise t' belongs to $\mathsf{range}(t_c)$, and is the closest occurrence (in $\mathsf{range}(t_c)$) to t_c such that t' and t_c are at least $|P|$ characters apart. Report t' as a non-overlapping occurrence.

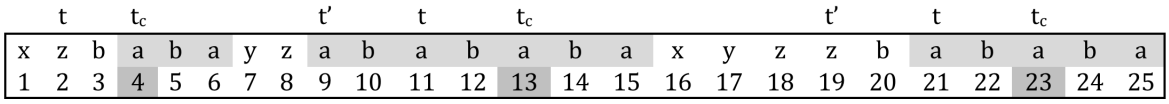

Figure 3.1: Illustration of Lemma 3.3. Top row shows the text, and bottom row shows the corresponding text position. Shaded text positions mark the critical occurrences of the pattern $P = aba$ for which $Q = ab$ and $Q' = a$. Shaded text region shows the range of the critical occurrences t_c; t and t' have the same meaning as in Lemma 3.3.

4. By letting $t_c = t'$, repeat the process starting from Step 2.

Clearly, at the end of the process described above, the desired $nocc'$ occurrences of P in $\mathsf{range}(t_c)$ are reported in $O(nocc' \cdot \mathsf{t_{SA}})$ time. ■

Our next task is to find all critical occurrences of P in T. The following lemma shows how to achieve this.

Lemma 3.4. *Given a pattern $P = Q^{\alpha}Q'$, we can find all critical occurrences of P in T in time bounded by $O(\mathsf{search}(P) + nocc \cdot \mathsf{t_{SA}})$.*

Proof. The proof relies on the following observation: *a critical occurrence of a pattern P is the same as the text position of a leaf which belongs to the suffix range of P, but not of QP.* If this is not true, then there is a critical occurrence of P, say at position t_c, such that $\mathsf{SA}^{-1}[t_c]$ lies in the suffix range of $QP = Q^{\alpha+1}Q'$. But then there is an occurrence of P at the position $t = t_c + |Q|$, a contradiction.

Since Q' is a prefix of Q, note that the suffix range of QP is contained within that of P; see Figure 3.2. Therefore, our objective translates to locating the suffix range of P, say $[sp, ep]$, and of QP, say $[sp', ep']$. This can be achieved in time $\mathsf{search}(QP)$, which can be bounded by $O(\mathsf{search}(P))$. (Recall that $\mathsf{search}(P)$ is proportional to $|P|$.)

For each leaf ℓ lying in $[sp, sp' - 1] \cup [ep' + 1, ep]$, the text position $\mathsf{SA}[\ell]$ is a (distinct) critical occurrence of P. Thus, the total number of these leaves is same as the number of critical occurrences of P in T. By Observation 3.3, the number of critical occurrences is at most the output size $nocc$. For every leaf ℓ, we can find the corresponding critical

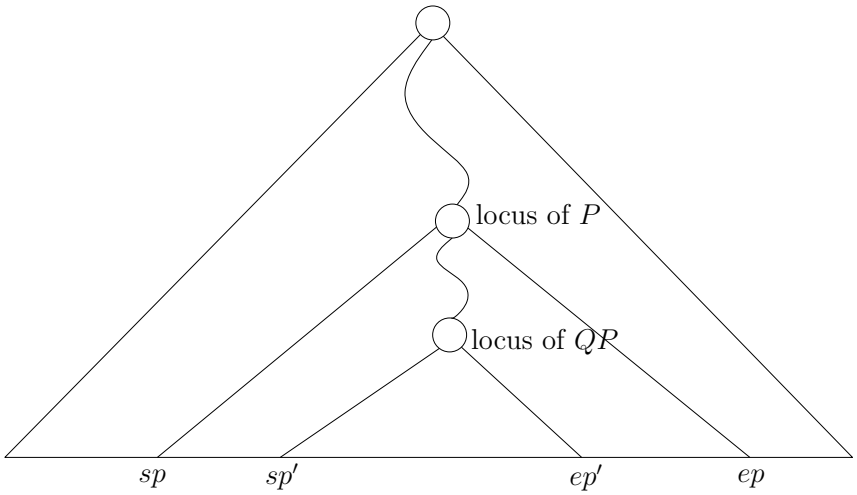

Figure 3.2: Illustration of Lemma 3.4. Since, Q' is a prefix of Q, the locus of $P = Q^{\alpha}Q'$ lies on the path from root to the locus of QP. For each leaf ℓ in $[sp, sp' - 1] \cup [ep' + 1, ep]$, the text position $\mathsf{SA}[\ell]$ is a critical occurrence of P.

occurrence (i.e., its text position) in time $\mathsf{t_{SA}}$ using $\mathsf{SA}[\ell]$. Therefore, once the suffix ranges of P and QP are located, all the critical occurrences are found in time $O(nocc \cdot \mathsf{t_{SA}})$. ■

From Lemma 3.3, we conclude that given the suffix range of P (which can be found in $\mathsf{search}(P)$ time) and every critical occurrence of P in T, we can find the desired $nocc$ non-overlapping occurrences of P in time $O(\mathsf{search}(P) + nocc \cdot \mathsf{t_{SA}})$. Every critical occurrence of P can be found using Lemma 3.4 in $O(\mathsf{search}(P) + nocc \cdot \mathsf{t_{SA}})$ time. By combining these lemmas with Lemma 3.2, we obtain Theorem 3.1.

Chapter 4
Succinct Index for Parameterized Pattern Matching

We begin with the definition of the parameterized matching. Here, the alphabet Σ is the union of two disjoint sets: Σ_s, the set of σ_s static characters (s-characters), and Σ_p, the set of σ_p parameterized characters (p-characters). Thus, $\sigma = |\Sigma| = \sigma_s + \sigma_p$.

Definition 4.1 (Parameterized Matching [Bak93])**.** *Two equal-length strings S and S' over Σ are a* parameterized match *(p-match) iff*

- $S[i] \in \Sigma_s \iff S'[i] \in \Sigma_s$,
- $S[i] = S'[i]$ *when* $S[i] \in \Sigma_s$, *and*
- *there exists a one-to-one matching-function f that renames the p-characters in S to the p-characters in S', i.e., $S'[i] = f(S[i])$ when $S[i] \in \Sigma_p$.*

For example, let $\Sigma_s = \{A, B, C, \$\}$ and $\Sigma_p = \{w, x, y, z\}$. Then, $S = AxByCx$ p-matches $S' = AzBwCz$, where the matching function f is: $f(x) = z$ and $f(y) = w$. Also, S p-matches $S' = AyBxCy$ with $f(x) = y$ and $f(y) = x$. However, S is not a p-match with $S' = AwBxCz$ as x in S would have to match with both w and z in S'.

The *Parameterized Text Indexing* problem is a generalization of the standard text-indexing problem, with the match between two strings replaced by a p-match. Specifically,

Problem 4.1 (Parameterized Text Indexing [Bak93])**.** *Let T be a text of length n over $\Sigma = \Sigma_s \cup \Sigma_p$. We assume that T terminates in an s-character \$ which appears only once. Index T, such that for a pattern P (also over Σ), we can report all the p-occurrences of P, i.e., all the starting positions of the substrings of T that are a p-match with P.*

For the above problem, Baker [Bak93] presented a linear-space index, known as the *Parameterized Suffix Tree*. It occupies $\Theta(n \log n)$-bits of space (i.e., $\Theta(n)$ words), and can

find all the *p-occurrences of* P in $O(|P| \log \sigma + occ)$ time, where occ is the number of p-occurrences of P. We present the following new result, which occupies (much) lesser space for a slightly higher query time.

Theorem 4.1. *By using an* $n \log \sigma + O(n)$*-bit index of* T*, the p-occurrences of* P *can be found in* $O(|P| \log \sigma + occ \cdot \log n \log \sigma)$ *time, where occ is the number of such p-occurrences.*

If we are allowed slightly higher space, then we can obtain the following result as an immediate consequence of our techniques for proving Theorem 4.1 above.

Theorem 4.2. *By using an* $O(n \log \sigma)$*-bit index of* T*, the p-occurrences of* P *can be found in* $O(|P| \log \sigma + occ \cdot \log n)$ *time, where occ is the number of such p-occurrences.*

4.1 Overview of Techniques

The key idea to obtain the linear-space index of Baker [Bak93] for Problem 4.1 is an encoding scheme, such that two strings are a p-match iff their encoded strings are a match in the traditional sense. The encoding scheme was introduced by Baker [Bak93], which led to the *parameterized suffix tree* (p-suffix tree) – encode each suffix of T and then create a compact trie of these encoded suffixes. Reporting the occurrences of a pattern is now trivial using the encoded P, the p-suffix tree, and the techniques introduced in Section 2.2. We first find the highest node u in the p-suffix tree such that the string obtained by concatenating the edge labels from root to u is prefixed by the encoded pattern. Then, we report the starting positions of the encoded suffixes corresponding to the leaves in the subtree of u. Although this uses lesser time than Theorem 4.1, the space required by the p-suffix tree is much higher than the text itself. We present the details in Section 4.2.

At this point, one may be tempted to think that we can apply the techniques of the FM-Index [FM05] to get a succinct equivalent of the p-suffix tree; see Section 2.3. The FM-Index (as does the Compressed Suffix Array [GV05]) relies on a crucial property of suffixes: any two suffixes which have the same preceding character c (in text order) will retain their relative lexicographic rank when they are prepended by c. Unfortunately, in

case of Baker's encoded suffixes, this property no longer holds. Consequently, FM-Index and CSA no longer work. The reason why encoded suffixes do not follow this property is that on prepending the preceding character, the encoding of the original suffix changes. Fortunately, this change happens at exactly one position. The key idea is to identify this position of change. However, we cannot explicitly store this position of change as it needs $\approx \log n$ bits per suffix. Instead, we store the number of distinct p-characters upto this position (from the start of the suffix). This information, which can be stored in $\approx \log \sigma$ bits per suffix, forms the backbone of our index. We call it the *Parameterized Burrows-Wheeler Transform* (pBWT); the details are in Section 4.3.

The main ingredient after obtaining pBWT is to implement an analogous version of the LF mapping of the FM-Index (see Section 2.3.1), which we call the *Parameterized LF mapping* (pLF mapping); see Section 4.4 for details. Recall that using LF mapping, we can simulate the suffix array without explicitly storing it; see Section 2.3.2. Similarly, using pLF mapping, we can simulate the parameterized suffix array, which stores the starting positions of the lexicographically arranged encoded suffixes; the techniques are standard and Theorem 4.3 presents a formal description.

Summarizing our discussions this far, we can see that the key is to compute pLF mapping. To this end, we use the pBWT and the topology of the p-suffix tree; the crucial insight is provided in Lemma 4.1. Based on this lemma, we implement pLF mapping in Section 4.5; space and time complexities are described in Theorem 4.4.

The last piece of the puzzle is to compute the suffix range of the encoded pattern (i.e., find the range of leaves under the node u defined at the beginning of this section). We again use pLF mapping, the tree topology, and pBWT to implement an analogous version of the backward search procedure of the FM Index (see Section 2.3.3). The details of the backward search procedure for p-matching is in Section 4.6.

4.2 Parameterized Suffix Tree

We now present the following encoding scheme introduced by Baker [Bak93].

Definition 4.2 (Baker's Encoding)**.** *We encode any string S over Σ into a string $\mathsf{prev}(S)$ of length $|S|$ as follows:*

$$\mathsf{prev}(S)[i] = \begin{cases} S[i] & \text{if } S[i] \text{ is an s-character,} \\ 0 & \text{else if } i \text{ is the first occurrence of } S[i] \text{ in } S, \\ i-j & \text{otherwise, where } j \text{ is the last occurrence of } S[i] \text{ before } i \text{ in } S. \end{cases}$$

In other words, $\mathsf{prev}(S)$ is obtained by replacing the first occurrence of every p-character in S by 0 and any other occurrence by the difference in text position from its previous occurrence. For example, $\mathsf{prev}(AxByCx) = A0B0C4$, where $A, B \in \Sigma_s$ and $x, y \in \Sigma_p$. The time required to compute $\mathsf{prev}(S)$ is $O(|S| \log \sigma)$[1].

Fact 4.1 ([Bak93])**.** *Two strings S and S' are a p-match iff $\mathsf{prev}(S) = \mathsf{prev}(S')$. Also S is a p-match with a prefix of S' iff $\mathsf{prev}(S)$ is a prefix of $\mathsf{prev}(S')$.*

Note that $\mathsf{prev}(S)$ is a string over an alphabet set $\Sigma_s \cup \{0, 1, \ldots, |S| - 1\}$. Moving forward, we follow the convention below.

Convention 4.1. *The integer characters (corresponding to p-characters) are lexicographically smaller than s-characters. An integer character i comes before another integer character j iff $i < j$. Also, \$ is lexicographically larger than all other characters.*

Parameterized Suffix Tree (pST) is the compacted trie of all strings in $\{\mathsf{prev}(T[k, n]) \mid 1 \leq k \leq n\}$. Clearly, pST consists of n leaves and at most $n-1$ internal nodes. Each edge is labeled with a sequence of characters from $\Sigma' = \Sigma_s \cup \{0, 1, \ldots, n-1\}$. See Figure 4.1 for an illustration. We use $\mathsf{path}(u)$ to denote the concatenation of edge labels on the path from

[1] Maintain a balanced binary search tree $\mathcal{T}$, which is initially empty. Scan S from left to right, and suppose $x \in \Sigma_p$ appears in position k. If $x \notin \mathcal{T}$, then assign $\mathsf{prev}(S)[k] = 0$, and insert x into $\mathcal{T}$. Otherwise, assign $\mathsf{prev}(S)[k] = k - k'$, where k' is the value associated with x in $\mathcal{T}$. Associate the value k with x in $\mathcal{T}$.

root to node u, and $\mathsf{strDepth}(u) = |\mathsf{path}(u)|$. The path of each leaf node corresponds to the encoding (using Definition 4.2) of a unique suffix of T, and leaves are ordered according to the lexicographic rank of the corresponding encoded suffixes. We use ℓ_i to denote the leaf corresponding to the ith lexicographically smallest prev-encoded suffix, i.e., the ith leftmost leaf in pST. Thus, $\mathsf{path}(\ell_i) = \mathsf{prev}(T[\mathsf{pSA}[i], n])$, where $\mathsf{pSA}[1, n]$ is the *Parameterized Suffix Array*, which maintains the lexicographic arrangement of all the encoded suffixes of T. In particular, $\mathsf{pSA}[i] = j$ and $\mathsf{pSA}^{-1}[j] = i$ iff $\mathsf{prev}(T[j, n])$ is the ith lexicographically smallest string in $\{\mathsf{prev}(T[k, n]) \mid 1 \leq k \leq n\}$.

Using pST, searching for all occurrences of P in T is straight-forward as follows. Simply traverse pST from root by following the edges labels and find the highest node u (called *locus*) with its path prefixed by $\mathsf{prev}(P)$. Then find the range $[sp, ep]$ (called *suffix range*) of leaves in the sub-tree of u and report $\{\mathsf{pSA}[i] \mid sp \leq i \leq ep\}$ as the output.

The space occupied by pST is $\Theta(n)$ words (or equivalently, $\Theta(n \log n)$ bits), and the query time is $O(|P| \log \sigma + occ)$, assuming perfect hashing [FKS84] at each node.

4.3 Parameterized Burrows-Wheeler Transform

We introduce a reversible transformation similar to that of the Burrows-Wheeler Transform [BW94]. We call this the Parameterized Burrows-Wheeler Transform (p-BWT). To obtain the p-BWT of T, we first create a conceptual matrix M, where each row corresponds to a unique circular suffix of T. Then, we sort all the rows lexicographically according to the $\mathsf{prev}(\cdot)$ encoding of the corresponding unique circular suffix and obtain the last column L of the sorted matrix M. Clearly, the ith row is equal to $T_{\mathsf{pSA}[i]}$. Moving forward, denote by f_i, the first occurrence of $L[i] = T_{\mathsf{pSA}[i]}[n]$ in $T_{\mathsf{pSA}[i]}$. (Note that f_i is defined.)

The p-BWT of T, denoted by $\mathsf{pBWT}[1, n]$, is defined as follows:

$$\mathsf{pBWT}[i] = \begin{cases} L[i], \text{ if } L[i] \text{ is an s-character,} \\ \text{number of distinct p-characters in } T_{\mathsf{pSA}[i]}[1, f_i], \quad \text{otherwise.} \end{cases}$$

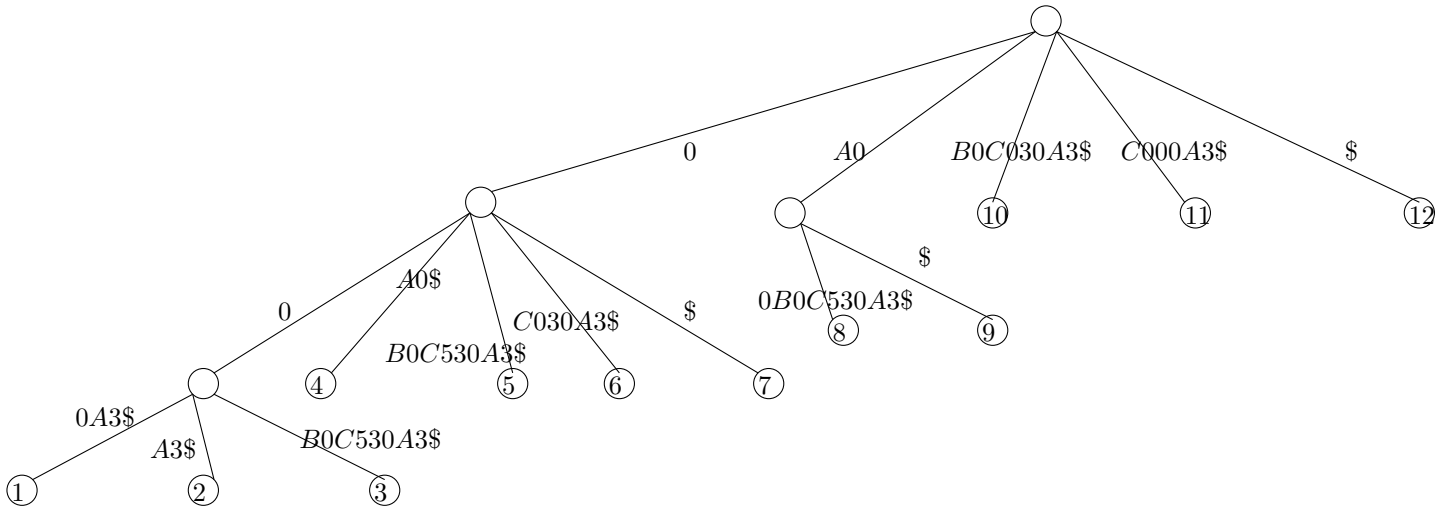

Figure 4.1: Parameterized Suffix Tree for text $AxyBzCxzwAz\$$, where $\Sigma_s = \{A, B, C, \$\}$ and $\Sigma_p = \{w, x, y, z\}$. We assume $0 \prec 1 \prec \cdots \prec n-1 \prec A \prec B \prec C \prec \$$, where $\prec$ denotes the total lexicographic order on the set $\{0, 1, 2, \ldots, n-1\} \cup \{A, B, C, \$\}$.

Table 4.1: Here the text is $T[1,12] = AxyBzCxzwAz\$$, where $\Sigma_s = \{A, B, C, \$\}$ and $\Sigma_p = \{w, x, y, z\}$

i	T_i	$\mathsf{prev}(T_i)$	$\mathsf{prev}(T_{\mathsf{pSA}[i]})$	$T_{\mathsf{pSA}[i]}$	$\mathsf{pSA}[i]$	$L[i]$	f_i	$\mathsf{pBWT}[i]$	$\mathsf{pLF}(i)$
1	AxyBzCxzwAz$	A00B0C530A3$	000A3$A70B6C	xzwAz$AxyBzC	7	C		C	11
2	xyBzCxzwAz$A	00B0C530A3$A	00A3$A00B6C5	zwAz$AxyBzCx	8	x	7	3	1
3	yBzCxzwAz$Ax	0B0C030A3$A7	00B0C530A3$A	xyBzCxzwAz$A	2	A		A	8
4	BzCxzwAz$Axy	B0C030A3$A70	0A0$A00B6C53	wAz$AxyBzCxz	9	z	3	2	2
5	zCxzwAz$AxyB	0C030A3$A70B	0B0C030A3$A7	yBzCxzwAz$Ax	3	x	5	3	3
6	CxzwAz$AxyBz	C000A3$A70B6	0C030A3$A70B	zCxzwAz$AxyB	5	B		B	10
7	xzwAz$AxyBzC	000A3$A70B6C	0$A00B5C530A	z$AxyBzCxzwA	11	A		A	9
8	zwAz$AxyBzCx	00A3$A00B6C5	A00B0C530A3$	AxyBzCxzwAz$	1	$		$	12
9	wAz$AxyBzCxz	0A0$A00B6C53	A0$A00B5C530	Az$AxyBzCxzw	10	w	12	4	4
10	Az$AxyBzCxzw	A0$A00B5C530	B0C030A3$A70	BzCxzwAz$Axy	4	y	12	4	5
11	z$AxyBzCxzwA	0$A00B5C530A	C000A3$A70B6	CxzwAz$AxyBz	6	z	3	2	6
12	$AxyBzCxzwAz	$A00B0C530A3	$A00B0C530A3	$AxyBzCxzwAz	12	z	6	3	7

In other words, when $L[i] \in \Sigma_s$, $\mathsf{pBWT}[i] = T[\mathsf{pSA}[i] - 1]$ (define $T[0] = T[n] = \$$ and $T_0 = T_n$) and when $L[i] \in \Sigma_p$, $\mathsf{pBWT}[i]$ is the number of 0's in the f_i-long prefix of $\mathsf{prev}(T_{\mathsf{pSA}[i]})$. Thus, pBWT is a sequence of n characters over the alphabet set $\Sigma'' = \Sigma_s \cup \{1, 2, \ldots, \sigma_p\}$ of size $\sigma_s + \sigma_p = \sigma$. See Table 4.1 for an illustration of pSA and pBWT.

In order to represent pBWT succinctly, we map each s-character in Σ'' to a unique integer in $[\sigma_p + 1, \sigma]$. Specifically, the ith smallest s-character will be denoted by $(i + \sigma_p)$. Moving forward, $\mathsf{pBWT}[i] \in [1, \sigma_p]$ iff $L[i]$ is a p-character and $\mathsf{pBWT}[i] \in [\sigma_p + 1, \sigma]$ iff $L[i]$ is a s-character. We summarize the relation between $\mathsf{prev}(T_{\mathsf{pSA}[i]})$ and $\mathsf{prev}(T_{\mathsf{pSA}[i]-1})$ in Observation 4.1.

Observation 4.1. *For any* $1 \leq i \leq n$,

$$\mathsf{prev}(T_{\mathsf{pSA}[i]-1}) = \begin{cases} \mathsf{pBWT}[i] \circ \mathsf{prev}(T_{\mathsf{pSA}[i]})[1, n-1], & \text{if } \mathsf{pBWT}[i] > \sigma_p, \\ 0 \circ \mathsf{prev}(T_{\mathsf{pSA}[i]})[1, f_i - 1] \circ f_i \circ \mathsf{prev}(T_{\mathsf{pSA}[i]})[f_i + 1, n-1], & \text{otherwise.} \end{cases}$$

4.4 Parameterized LF Mapping

Based on our conceptual matrix M, the parameterized last-to-first column (pLF) mapping of i is the position at which the character at $L[i]$ lies in the first column of M. Specifically, $\mathsf{pLF}(i) = \mathsf{pSA}^{-1}[\mathsf{pSA}[i] - 1]$, where $\mathsf{pSA}^{-1}[0] = \mathsf{pSA}^{-1}[n]$; see Table 4.1. The significance of pLF mapping is summarized in the following theorem.

Theorem 4.3. *Assume* $\mathsf{pLF}(i)$ *for any* $i \in [1, n]$ *is computed in* t_{pLF} *time. For any parameter* Δ*, by maintaining an additional* $O((n/\Delta) \log n)$*-bit data structure, we can compute* $\mathsf{pSA}[j]$ *for any* $j \in [1, n]$ *in* $O(\Delta \cdot t_{\mathsf{pLF}})$ *time.*

Proof. Define, $\mathsf{pLF}^0(i) = i$ and $\mathsf{pLF}^k(i) = \mathsf{pLF}(\mathsf{pLF}^{k-1}(i)) = \mathsf{pSA}^{-1}[\mathsf{pSA}[i] - k]$ for any integer $k > 0$. We employ perfect hashing [FKS84] to store the $\langle j, \mathsf{pSA}[j] \rangle$ key-value pairs for all j such that $\mathsf{pSA}[j]$ belongs to $\{1, 1 + \Delta, 1 + 2\Delta, 1 + 3\Delta, \ldots, n\}$. Using this, given a j, one can check if $\mathsf{pSA}[j]$ has been stored (and also retrieve the value) in $O(1)$ time. The total space occupied is $O((n/\Delta) \log n)$ bits. To find $\mathsf{pSA}[i]$, repeatedly apply the $\mathsf{pLF}(\cdot)$

operation (starting from i) until we obtain a j such that $\mathsf{pSA}[j]$ has been explicitly stored. If the number of $\mathsf{pLF}(\cdot)$ operations invoked is k, then $j = \mathsf{pLF}^k(i) = \mathsf{pSA}^{-1}[\mathsf{pSA}[i] - k]$, which gives $\mathsf{pSA}[i] = \mathsf{pSA}[j] + k$. Since $k \leq \Delta$, $\mathsf{pSA}[i]$ is computed in $O(\Delta \cdot t_{\mathsf{pLF}})$ time. ■

To aid the reader's intuition for computing pLF mapping, we present Lemma 4.1, which shows how to compare the (new) lexicographic rank of two encoded suffixes when prepended by their respective previous characters. This key concept is then implemented in Section 4.4 to arrive at Theorem 4.4.

Lemma 4.1. *Consider two suffixes i and j corresponding to the leaves ℓ_i and ℓ_j in pST. Then, $\mathsf{pLF}(i)$ and $\mathsf{pLF}(j)$ are related as follows:*

(a) *If $L[i] \in \Sigma_p$ and $L[j] \in \Sigma_s$, then $\mathsf{pLF}(i) < \mathsf{pLF}(j)$*

(b) *If both $L[i], L[j] \in \Sigma_s$, then $\mathsf{pLF}(i) < \mathsf{pLF}(j)$ iff one of the following holds:*

- $\mathsf{pBWT}[i] < \mathsf{pBWT}[j]$
- $\mathsf{pBWT}[i] = \mathsf{pBWT}[j]$ *and* $i < j$

(c) *Assume both $L[i], L[j] \in \Sigma_p$ and $i < j$. Let u be the lowest common ancestor of ℓ_i and ℓ_j in pST, and z be the number of 0's in the string $\mathsf{path}(u)$.*

(1) *If $\mathsf{pBWT}[i], \mathsf{pBWT}[j] \leq z$, then $\mathsf{pLF}(i) < \mathsf{pLF}(j)$ iff $\mathsf{pBWT}[i] \geq \mathsf{pBWT}[j]$*

(2) *If $\mathsf{pBWT}[i] \leq z < \mathsf{pBWT}[j]$, then $\mathsf{pLF}(i) > \mathsf{pLF}(j)$*

(3) *If $\mathsf{pBWT}[i] > z \geq \mathsf{pBWT}[j]$, then $\mathsf{pLF}(i) < \mathsf{pLF}(j)$*

(4) *If $\mathsf{pBWT}[i], \mathsf{pBWT}[j] > z$, then $\mathsf{pLF}(i) > \mathsf{pLF}(j)$ iff all of the following are true:*

- $\mathsf{pBWT}[i] = z + 1$,
- *the leading character on the u to ℓ_i path is 0, and*
- *the leading character on the u to ℓ_j path is not an s-character.*

Proof. (a) and (b): Follows immediately from Convention 4.1 and Observation 4.1.
(c) Recall that f_i and f_j are the first occurrences of the characters $L[i]$ and $L[j]$ in the circular suffixes $T_{\mathsf{pSA}[i]}$ and $T_{\mathsf{pSA}[j]}$ respectively. Let $d = \mathsf{strDepth}(u)$. Clearly, the conditions

in (1)–(4) can be written as: (1) Both $f_i, f_j \leq d$, (2) $f_i \leq d$ and $f_j > d$, (3) $f_i > d$ and $f_j \leq d$, and (4) Both $f_i, f_j > d$. The claims in (1)–(4) also follow from Convention 4.1 and Observation 4.1. The first three claims are trivial. To prove (4), observe that the suffixes i and j swap order iff $f_i = d + 1$, and the leading character on the u to ℓ_j path is not an s-character. ■

Theorem 4.4. *We can compute* $\mathsf{pLF}(i)$ *in* $O(\log \sigma)$ *time by using an* $n \log \sigma + O(n)$*-bit data structure.*

4.5 Implementing Parameterized LF Mapping

In this section, we prove Theorem 4.4. We begin with the following key components of the data structure.

- **Wavelet Tree over** pBWT**:** The pBWT is an array of length n over the alphabet $\{1, 2, \ldots, \sigma_p, \sigma_p + 1, \ldots, \sigma\}$; recall that the highest σ_s numbers in this alphabet correspond to the s-characters, and the ith highest s-character is given by $\sigma_p + i$. We maintain the Wavelet-Tree of Fact 2.3 in $n \log \sigma + o(n)$ bits to support the listed operations for the pBWT.

- **Succinct Representation of** pST**:** We maintain the data structure of Fact 2.4 in total $4n + o(n)$ bits to support the listed operations for the parameterized suffix tree.

4.5.1 ZeroDepth and ZeroNode

For a node u, $\mathsf{zeroDepth}(u)$ is the number of 0's in $\mathsf{path}(u)$. For a leaf ℓ_i with $\mathsf{pBWT}[i] \in [1, \sigma_p]$, we define $\mathsf{zeroNode}(\ell_i)$ as follows. If $f_i \leq n - \mathsf{pSA}[i]$, then $\mathsf{zeroNode}(\ell_i)$ is the highest node z on the root to ℓ_i path such that $\mathsf{zeroDepth}(z) \geq \mathsf{pBWT}[i]$. Thus, z is the locus of $\mathsf{path}(\ell_i)[1, f_i]$ if $f_i \leq n - \mathsf{pSA}[i]$; note that z exists as $\mathsf{zeroDepth}(\ell_i) \geq \mathsf{pBWT}[i]$. If $f_i > n - \mathsf{pSA}[i]$, then $\mathsf{zeroNode}(\ell_i)$ is not defined.

To distinguish between the two cases, we maintain a bit-vector $\mathsf{falseZero}[1, n]$, where $\mathsf{falseZero}[i] = 1$ iff $\mathsf{pBWT}[i] \leq \sigma_p$ and $f_i > n - \mathsf{pSA}[i]$. Moving forward, whenever we refer

to $\mathsf{zeroNode}(\ell_i)$, we assume $\mathsf{pBWT}[i] \leq \sigma_p$ and $\mathsf{falseZero}(i) = 0$. To find $\mathsf{zeroNode}(\ell_i)$, we use the following lemma (proof deferred to Section 4.5.5).

Lemma 4.2. *By using the wavelet tree over* pBWT *and an additional* $O(n)$*-bit data structure, we can find* $\mathsf{zeroNode}(\ell_i)$ *in* $O(\log \sigma)$ *time.*

We remark that the following components will be defined later: $\mathsf{leafLeadChar}(\cdot)$, $\mathsf{fSum}(\cdot)$ and $\mathsf{pCount}(\cdot)$; each can be computed in $O(1)$ time using an $O(n)$-bit data structure.

4.5.2 Computing $\mathsf{pLF}(i)$ when $\mathsf{pBWT}[i] \in [\sigma_p + 1, \sigma]$

Using Lemma 4.1, we conclude that $\mathsf{pLF}(i) > \mathsf{pLF}(j)$ iff either $j \in [1, n]$ and $\mathsf{pBWT}[j] < \mathsf{pBWT}[i]$, or $j \in [1, i-1]$ and $\mathsf{pBWT}[i] = \mathsf{pBWT}[j]$. Then,

$$\mathsf{pLF}(i) = 1 + \mathsf{count}(1, n, 1, \mathsf{pBWT}[i] - 1) + \mathsf{count}(1, i-1, \mathsf{pBWT}[i], \mathsf{pBWT}[i])$$

4.5.3 Computing $\mathsf{pLF}(i)$ when $\mathsf{pBWT}[i] \in [1, \sigma_p]$ and $\mathsf{falseZero}[i] = 0$

Let $z = \mathsf{zeroNode}(\ell_i)$ and $v = \mathsf{parent}(z)$. Then, $f_i = (\mathsf{strDepth}(v) + 1)$ if the leading character on the edge from v to z is 0 and $\mathsf{pBWT}[i] = (\mathsf{zeroDepth}(v) + 1)$; otherwise, $f_i > (\mathsf{strDepth}(v) + 1)$. For a leaf ℓ_j, $\mathsf{leafLeadChar}(j)$ is a boolean variable, which is 0 iff $f_j = (\mathsf{strDepth}(\mathsf{parent}(\mathsf{zeroNode}(\ell_j))) + 1)$. Using this information, in $O(1)$ time, we can determine which of the following two cases is satisfied (see Figure 4.2).

Sub-case 1 ***($f_i = \mathsf{strDepth}(v) + 1$).*** Note that the leading character on the edge from v to z is 0; hence, applying Convention 4.1, z is the leftmost child of v (because leaves are lexicographically ordered). Let w be the parent of v. We partition the leaves into four sets:

(a) $\mathcal{S}_1$: leaves to the left of the subtree of v.

(b) $\mathcal{S}_2$: leaves in the subtree of z.

(c) $\mathcal{S}_3$: leaves to the right of the subtree of v.

(d) $\mathcal{S}_4$: leaves in the subtree of v but not of z.

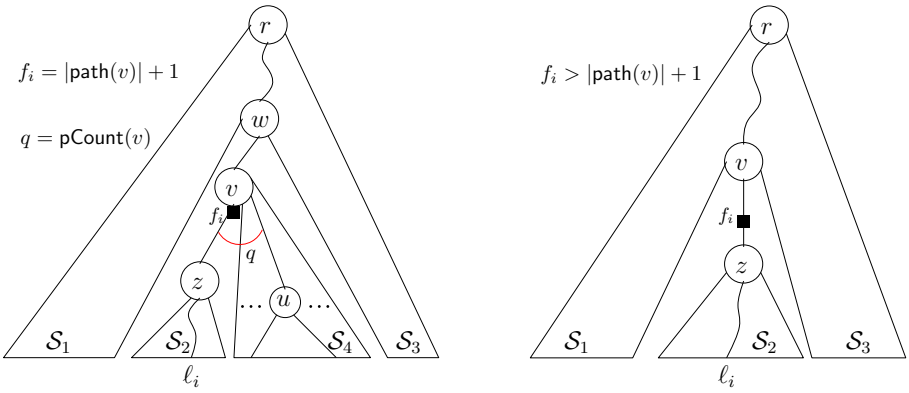

Figure 4.2: Various suffix ranges when $\mathsf{pBWT}[i] \leq \sigma_p$ and $\mathsf{falseZero}(i) = 0$

In case, v is the root node r, we take $w = r$; consequently, $\mathcal{S}_1 = \mathcal{S}_3 = \varnothing$.

Sub-case 2 ***($f_i > \mathsf{strDepth}(v) + 1$).*** We partition the leaves into three sets:

(a) $\mathcal{S}_1$: leaves to the left of the subtree of z.

(b) $\mathcal{S}_2$: leaves in the subtree of z.

(c) $\mathcal{S}_3$: leaves to the right of the subtree of z.

We first compute $z = \mathsf{zeroNode}(\ell_i)$ using Lemma 4.2, and then locate $v = \mathsf{parent}(z)$. Using $\mathsf{leafLeadChar}(i)$ and the $\mathsf{lmostLeaf}(\cdot)/\mathsf{rmostLeaf}(\cdot)$ tree operations, we find the desired ranges. Let $[L_x, R_x]$ denote the range of leaves in the subtree of any node x, i.e., $L_x = \mathsf{lmostLeaf}(x)$ and $R_x = \mathsf{rmostLeaf}(x)$. In order to compute $\mathsf{pLF}(i)$, we first compute N_1, N_2, and N_3, which are respectively the number of leaves ℓ_j in the ranges $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_3$ such that $\mathsf{pLF}(j) \leq \mathsf{pLF}(i)$. Likewise, we compute N_4 (w.r.t $\mathcal{S}_4$) if we are in the first case. Then, $\mathsf{pLF}(i)$ is computed as $\mathsf{pLF}(i) = N_1 + N_2 + N_3 + N_4$. Now, we undertake the task of computing N_1 through N_4. Computing these are simple applications of Lemma 4.1.

Computing N_1. For any leaf $\ell_j \in \mathcal{S}_1$, $\mathsf{pLF}(j) < \mathsf{pLF}(i)$ iff $f_j > 1 + \mathsf{strDepth}(\mathsf{lca}(z, \ell_j))$ and $\mathsf{pBWT}[j] \leq \sigma_p$. Thus, N_1 is the number of leaves ℓ_j such that $\mathsf{pBWT}[j] \leq \sigma_p$, $\mathsf{pre\text{-}order}(\ell_j) < \mathsf{pre\text{-}order}(z)$, and $f_j > 1 + \mathsf{strDepth}(\mathsf{lca}(z, \ell_j))$. Define, $\mathsf{fCount}(x)$ of a node x as the number of leaves ℓ_j in x's subtree such that $\mathsf{strDepth}(\mathsf{parent}(x)) + 2 \leq f_j \leq \mathsf{strDepth}(x) + 1$. If x is

the root node, then $\mathsf{fCount}(x) = 0$. Define $\mathsf{fSum}(x)$ of a node x as $\sum \mathsf{fCount}(y)$ of all nodes y which come before x in pre-order and are not ancestors of x. Then,

$$N_1 = \mathsf{rank}_{\mathsf{falseZero}}(L_z - 1, 1) + \mathsf{fSum}(z),$$

where $\mathsf{fSum}(z)$ is computed as follows.

Lemma 4.3. *By maintaining an $O(n)$-bit structure, we can compute $\mathsf{fSum}(x)$ in $O(1)$ time.*

Proof. Traverse the pST in DFS order. Append $\mathsf{fCount}(v)$ in unary[2] to an initially empty binary string B when exiting the subtree rooted at node v in the traversal, i.e., $\mathsf{fCount}(v)$ is associated with $\mathsf{post\text{-}order}(v)$. Maintain the $\mathsf{rank\text{-}select}$ structure of Fact 2.2 on B. Since $\sum_v \mathsf{fCount}(v) \leq n$, we have $|B| \leq 3n$. Thus, the space occupied is $3n + o(n)$ bits. Note that $\mathsf{fSum}(x)$ is same as the number of 1s in B up to the position corresponding to a node y, where the node y is conceptually found as follows. Traverse from x to root until we get a node y' which has a child to the left of the path. Then y is the rightmost child of y' that lies to the left of the path. If $L_x = 1$, then y is not defined and $\mathsf{fSum}(x) = 0$. Otherwise, we use Facts 2.2 and 2.4 to compute $y = \mathsf{levelAncestor}(\ell_{L_x-1}, \mathsf{nodeDepth}(\mathsf{lca}(\ell_{L_x}, \ell_{L_x-1})) + 1)$ and $\mathsf{fSum}(x) = \mathsf{rank}_B(\mathsf{select}_B(\mathsf{post\text{-}order}(y), 0), 1)$, both in $O(1)$ time. ■

Computing N_2. Note that for any leaf $\ell_j \in \mathcal{S}_2$, $\mathsf{pLF}(j) \leq \mathsf{pLF}(i)$ iff $\mathsf{pBWT}[j] \leq \sigma_p$ and either $f_j > f_i$ or $f_j = f_i$ and $j \leq i$. Therefore, N_2 is the number of leaves ℓ_j in $\mathcal{S}_2$ which satisfy either (a) $\mathsf{pBWT}[i] < \mathsf{pBWT}[j] \leq \sigma_p$, or (b) $\mathsf{pBWT}[i] = \mathsf{pBWT}[j]$ and $j \leq i$. Then,

$$N_2 = \mathsf{count}(L_z, R_z, \mathsf{pBWT}[i] + 1, \sigma_p) + \mathsf{count}(L_z, i, \mathsf{pBWT}[i], \mathsf{pBWT}[i])$$

Computing N_3. For any leaf $\ell_j \in \mathcal{S}_3$, $\mathsf{pLF}(j) > \mathsf{pLF}(i)$. Thus, $N_3 = 0$.

Computing N_4. Note that $\mathsf{pBWT}[i] = \mathsf{zeroDepth}(v) + 1$. Consider a leaf $\ell_j \in \mathcal{S}_4$ with $\mathsf{pBWT}[j] \leq \sigma_p$. Since the suffix j deviates from the suffix i at the node v, we have $f_j \neq f_i$.

[2]Unary encoding of a number x is a string containing x number of 1s followed by a 0.

Therefore, $\mathsf{pLF}(j) < \mathsf{pLF}(i)$ iff $f_j > f_i$, and the leading character on the path from v to ℓ_j is not an s-character. For a node x, $\mathsf{pCount}(x)$ is the number of children y of x such that the leading character from x to y is not an s-character. Note that $\sum_x \mathsf{pCount}(x) = O(n)$. Therefore, we encode $\mathsf{pCount}(\cdot)$ of all nodes in $O(n)$ bits using unary encoding, such that $\mathsf{pCount}(x)$ can be retrieved in constant time[3]. Let u be the $\mathsf{pCount}(v)$th child of v. Then, N_4 is the number of leaves ℓ_j in $\mathcal{S}_4$ such that $j \leq R_u$ and $\sigma_p \geq \mathsf{pBWT}[j] \geq \mathsf{pBWT}[i]$ i.e.,

$$N_4 = \mathsf{count}(R_z + 1, R_u, \mathsf{pBWT}[i], \sigma_p)$$

4.5.4 Computing $\mathsf{pLF}(i)$ when $\mathsf{pBWT}[i] \in [1, \sigma_p]$ and $\mathsf{falseZero}[i] = 1$

For any leaf ℓ_j, where $j < i$, we have $\mathsf{pLF}(j) < \mathsf{pLF}(i)$ iff $f_j > 1 + \mathsf{strDepth}(\mathsf{lca}(\ell_i, \ell_j))$ and $\mathsf{pBWT}[j] \leq \sigma_p$. For any leaf ℓ_j, where $j > i$, we have $\mathsf{pLF}(j) > \mathsf{pLF}(i)$. Therefore, $\mathsf{pLF}(i) = 1+N$, where N is the number of leaves ℓ_j, such that $\mathsf{pBWT}[j] \leq \sigma_p$, $\mathsf{pre\text{-}order}(\ell_j) < \mathsf{pre\text{-}order}(\ell_i)$, and $f_j > 1 + \mathsf{strDepth}(\mathsf{lca}(\ell_i, \ell_j))$. In other words,

$$\mathsf{pLF}(i) = 1 + \mathsf{rank}_{\mathsf{falseZero}}(i - 1, 1) + \mathsf{fSum}(\ell_i)$$

We summarize the LF mapping procedure in Algorithm 2. The data structure occupies $n \log \sigma + O(n)$ bits. If $\mathsf{pBWT}[i] > \sigma_p$, then $\mathsf{pLF}(i)$ is computed in $O(1 + \log \sigma / \log \log n)$ time. Otherwise, if $\mathsf{falseZero}(i) = 0$, we first compute $\mathsf{zeroNode}(\ell_i)$ in $O(\log \sigma)$ time using Lemma 4.2. Now, we find the desired ranges $\mathcal{S}_1$, $\mathcal{S}_2$, $\mathcal{S}_3$, and if required $\mathcal{S}_4$, all in $O(1)$ time. Then, N_1 is computed in $O(1)$ time, and both N_2 and N_4 are computed in $O(1 + \log \sigma / \log \log n)$ time. Lastly, if $\mathsf{pBWT}[i] \leq \sigma_p$ and $\mathsf{falseZero}(i) = 1$, we compute $\mathsf{pLF}(i)$ in $O(1)$ time. Thus, $\mathsf{pLF}(i)$ is computed in $O(\log \sigma)$ time when $\mathsf{pBWT}[i] \in [1, \sigma_p]$. This concludes the proof of Theorem 4.4.

[3] Create a binary string S as follows. For each node u in the pre-order traversal of pST, append to S a 0 followed by $\mathsf{pCount}(u)$ number of 1s. Append a 0 at the end. Maintain a rank-select structure (refer to Fact 2.2) over S. Then $\mathsf{pCount}(u)$ for a node u, having pre-order rank k, is the number of 1s between the kth 0 and the $(k+1)$th 0. The value is given by $\mathsf{rank}(\mathsf{select}(k+1, 0), 1) - \mathsf{rank}(\mathsf{select}(k, 0), 1)$ in $O(1)$ time.

Algorithm 2 computes $\mathsf{pLF}(i)$

1: $c \leftarrow \mathsf{pBWT}[i]$
2: **if** $(c > \sigma_p)$ **then**
3: $\quad \mathsf{pLF}(i) \leftarrow 1 + \mathsf{count}(1, n, 1, c-1) + \mathsf{count}(1, i-1, c, c)$
4: **else**
5: $\quad$ **if** $(\mathsf{falseZero}[i] = 1)$ **then**
6: $\quad\quad \mathsf{pLF}(i) \leftarrow 1 + \mathsf{rank}_{\mathsf{falseZero}}(i-1, 1) + \mathsf{fSum}(\ell_i)$
7: $\quad$ **else**
8: $\quad\quad z \leftarrow \mathsf{zeroNode}(\ell_i)$, $v \leftarrow \mathsf{parent}(z)$, $L_z \leftarrow \mathsf{lmostLeaf}(z)$, $R_z \leftarrow \mathsf{rmostLeaf}(z)$
9: $\quad\quad N_1 \leftarrow \mathsf{rank}_{\mathsf{falseZero}}(L_z - 1, 1) + \mathsf{fSum}(z)$
10: $\quad\quad N_2 \leftarrow \mathsf{count}(L_z, R_z, c+1, \sigma_p) + \mathsf{count}(L_z, i, c, c)$
11: $\quad\quad$ **if** $(\mathsf{leafLeadChar}(i)$ is $0)$ **then**
12: $\quad\quad\quad u \leftarrow \mathsf{child}(v, \mathsf{pCount}(v))$
13: $\quad\quad\quad N_4 \leftarrow \mathsf{count}(R_z + 1, \mathsf{rmostLeaf}(u), c, \sigma_p)$
14: $\quad\quad \mathsf{pLF}(i) \leftarrow N_1 + N_2 + N_4$

4.5.5 Finding ZeroNode

We prove Lemma 4.2 in this section. For a node x, let $L_x = \mathsf{lmostLeaf}(x)$ and $R_x = \mathsf{rmostLeaf}(x)$. For a node x on the root to ℓ_i path π, let $\alpha(x) =$ the number of leaves ℓ_j, $j \in [L_x, R_x]$ such that $L[j] \in \Sigma_p$ and $f_j \leq \mathsf{strDepth}(x)$, and $\beta(x) = \mathsf{count}(L_x, R_x, 1, \mathsf{pBWT}[i])$. Consider a node u_k on π. Observe that $\mathsf{zeroNode}(\ell_i)$ is below u_k iff $\beta(u_k) > \alpha(u_k)$. Thus, $\mathsf{zeroNode}(\ell_i)$ is the shallowest node $u_{k'}$ on this path that satisfies $\beta(u_{k'}) \leq \alpha(u_{k'})$. Equipped with this knowledge, we can binary search on π (using $\mathsf{nodeDepth}$ and $\mathsf{levelAncestor}$ operations) to find the exact location. The first task is to compute $\alpha(x)$, which is handled by Lemma 4.4. A normal binary search will have to consider n nodes on the path in the worst case. Lemma 4.5 shows how to reduce this to $\lceil \log \sigma \rceil$. Thus, the binary search has at most $\lceil \log \log \sigma \rceil$ steps, and the total time is $\log \log \sigma \times \lceil \frac{\log \sigma}{\log \log n} \rceil = O(\log \sigma)$, as required.

Lemma 4.4. *By maintaining an $O(n)$-bit data structure, we can find $\alpha(x)$ in $O(1)$ time.*

Proof. Let $A[1, n]$ be a bit-vector such that $A[i] = 1$ iff $\mathsf{pBWT}[i] \leq \sigma_p$. Define $\gamma(v)$ as the number of leaves $\ell_j \in [L_v, R_v]$ that satisfy $\mathsf{pBWT}[j] \leq \sigma_p$ and $\mathsf{strDepth}(\mathsf{parent}(v)) < f_j \leq \mathsf{strDepth}(v)$. Traverse pST in DFS order. Append $\gamma(v)$ in unary to an initially empty binary string B when entering the subtree of node v in the traversal, i.e., $\gamma(v)$ is associated with $\mathsf{pre\text{-}order}(v)$. Since $\sum_v \gamma(v) \leq n$, we have $|B| \leq 3n$. Maintain the $\mathsf{rank\text{-}select}$ structure

of Fact 2.2 on the bit-vectors A and B. The total space needed is $O(n)$ bits. Let $\alpha'(x)$ be the number of leaves $\ell_j \in [L_x, R_x]$ such that $\mathsf{pBWT}[j] \leq \sigma_p$ and $f_j > \mathsf{strDepth}(x)$. Then,

$$l = \mathsf{select}_B(\mathsf{pre\text{-}order}(x), 0) \text{ and } r = \mathsf{select}_B(\mathsf{pre\text{-}order}(\ell_{R_x}), 0)$$
$$\alpha'(x) = \mathsf{rank}_B(l, r, 1) + \mathsf{rank}_{\mathsf{falseZero}}(L_x, R_x, 1)$$
$$\alpha(x) = \mathsf{rank}_A(L_x, R_x, 1) - \alpha'(x)$$

Clearly, the space-and-time bounds are met. ■

Lemma 4.5. *Using the wavelet tree over* pBWT *and an additional $O(n)$-bit data structure, in $O(\log \sigma)$ time, we can find an ancestor w_i of ℓ_i such that* $\mathsf{zeroDepth}(w_i) < \mathsf{pBWT}[i]$ *and w_i is at most $\lceil \log \sigma \rceil$ nodes above* $\mathsf{zeroNode}(\ell_i)$.

Proof. Let $g = \lceil \log \sigma \rceil$. We mark the nodes v in the pST such that $\mathsf{nodeDepth}(v)$ is a multiple of g and the subtree of v has at least g nodes. Also, mark the root node. It is easy to see that (i) between any two closest marked nodes (or a lowest marked node and a leaf in its subtree) there are at most g nodes, and (ii) the number of marked nodes is $O(n/g)$. Maintain a bit-vector B such that $B[k] = 1$ iff the node with pre-order rank k is a marked node. Also, maintain the rank-select structure of Fact 2.2 on B. The space needed is $O(n)$ bits. Let D be array, where $D[k]$ equals the zeroDepth of the marked node corresponding to the kth 1-bit in B. Given a marked node with pre-order rank k', its corresponding position in D is given by $\mathsf{rank}_B(k', 1)$. We maintain the wavelet tree of Fact 2.3 over D. The space needed is $O(\frac{n}{g} \log \sigma) = O(n)$ bits.

To find w_i, we locate the lowest marked ancestor u of ℓ_i by traversing pST upwards using the parent operation. We can check if a node is marked using B in $O(1)$ time. The time required to find u is $O(\log \sigma)$. Let $j = \mathsf{rank}_B(\mathsf{pre\text{-}order}(u), 1)$ be the position corresponding to u in D. If $\mathsf{zeroDepth}(u) = D[j] < \mathsf{pBWT}[i]$, assign $w_i = u$. Otherwise, we locate the rightmost position $j' < j$ in D such that $D[j'] < \mathsf{pBWT}[i]$ in $O(\log \sigma)$ time using the query $j' = \mathsf{predecessor}_D(j, \mathsf{pBWT}[i]-1)$. (Since the root r is marked and $\mathsf{zeroDepth}(r) = 0$, the position

j' exists.) Obtain the marked node v corresponding to the j'th 1-bit in B via $\mathsf{select}_B(j', 1)$. Assign $w_i = \mathsf{lca}(u, v)$. The time required is $O(\log \sigma)$. To see the correctness, observe that $\mathsf{lca}(u, v)$ is an ancestor of ℓ_i. For a node x, $\mathsf{zeroDepth}(x) \geq \mathsf{zeroDepth}(\mathsf{parent}(x))$. Thus, $\mathsf{zeroDepth}(\mathsf{lca}(u, v)) \leq \mathsf{zeroDepth}(v) < \mathsf{pBWT}[i]$. If w_i has been incorrectly chosen, then $\mathsf{lca}(u, v)$ is more than $\lceil \log \sigma \rceil$ nodes above $\mathsf{zeroNode}(\ell_i)$. Hence, $\mathsf{lca}(u, v)$ has a marked descendant $u' \neq u$ on the path to u such that $\mathsf{zeroDepth}(u') < \mathsf{pBWT}[i]$. But u' appears after v and before u in pre-order, a contradiction. ■

4.6 Finding Suffix Range via Backward Search

Since pLF mapping is computed in $O(\log \sigma)$ time (see Theorem 4.4), by using Theorem 4.3 with an appropriate choice of Δ, we can compute the parameterized suffix array values. To complete pattern matching, we now need to compute the suffix range of $\mathsf{prev}(P)$. To this end, we modify the backward search algorithm in the FM-index [FM05]. In particular, given a proper suffix Q of P, assume that we know the suffix range $[sp_1, ep_1]$ of $\mathsf{prev}(Q)$. Our task is to find the suffix range $[sp_2, ep_2]$ of $\mathsf{prev}(c \circ Q)$, where c is the character previous to Q in P. If we can carry out this operation, then by repeating the process until P is entirely read, we can find the suffix range of $\mathsf{prev}(P)$. We note that any suffix $i \notin [sp_1, ep_1]$ satisfies $\mathsf{pLF}(i) \notin [sp_2, ep_2]$. This is because Q is not a p-match with $T_{\mathsf{pSA}[i]}[1, |Q|]$; hence, $c \circ Q$ cannot be a p-match with $T_{\mathsf{pSA}[\mathsf{pLF}(i)]}[1, 1 + |Q|]$. Therefore, only suffixes from $[sp_1, ep_1]$ can belong to $[sp_2, ep_2]$. We consider the following two cases.

4.6.1 c is static

In this case, $\mathsf{prev}(c \circ Q) = c \circ \mathsf{prev}(Q)$. Thus, $\mathsf{pLF}(i) \in [sp_2, ep_2]$ iff $i \in [sp_1, ep_1]$ and $\mathsf{pBWT}[i] = c$. The backward search in this case is similar to that in FM-index. Specifically,

$$sp_2 = 1 + \mathsf{count}(1, n, 1, c - 1) + \mathsf{count}(1, sp_1 - 1, c, c)$$

$$ep_2 = \mathsf{count}(1, n, 1, c - 1) + \mathsf{count}(1, ep_1, c, c)$$

4.6.2 c is parameterized

We first identify all positions j, such that $P[j] \in \Sigma_p$ and $P[j]$ does not appear in $P[j{+}1, |P|]$. This pre-processing requires $O(|P| \log \sigma)$ time[4]. We have the following two cases.

Case 1 (c does not appear in Q). Note that $\mathsf{pLF}(i) \in [sp_2, ep_2]$ iff $i \in [sp_1, ep_1]$, $\mathsf{pBWT}[i] \leq \sigma_p$ and $f_i > |Q|$. This holds iff $i \in [sp_1, ep_1]$ and $\mathsf{pBWT}[i] \in [d+1, \sigma_p]$. Here, d is the number of distinct p-characters in Q, which can be obtained in $O(1)$ time by initially pre-processing P in $O(|P| \log \sigma)$ time[5]. Thus,

$$(ep_2 - sp_2 + 1) = \mathsf{count}(sp_1, ep_1, d+1, \sigma_p)$$

Additionally, $\mathsf{pLF}(i) > ep_2$ for any $i > ep_1$ or for any $i \in [sp_1, ep_1]$ and $\mathsf{pBWT}[i] \leq d$. In other words, $\mathsf{pLF}(i) < sp_2$ iff $i < sp_1$, $\mathsf{pBWT}[i] \leq \sigma_p$, and $f_i > 1 + \mathsf{strDepth}(\mathsf{lca}(u, \ell_i))$, where $u = \mathsf{lca}(\ell_{sp_1}, \ell_{ep_1})$. Thus,

$$sp_2 = 1 + \mathsf{rank}_{\mathsf{falseZero}}(sp_1 - 1, 1) + \mathsf{fSum}(u),$$

which is computed in $O(1)$ time using Fact 2.2 and Lemma 4.3.

Case 2 (c appears in Q). Note that $\mathsf{pLF}(i) \in [sp_2, ep_2]$ iff $i \in [sp_1, ep_1]$, $\mathsf{pBWT}[i] \leq \sigma_p$, and f_i equals the first occurrence of c in Q. This holds iff $i \in [sp_1, ep_1]$ and $\mathsf{pBWT}[i] = d$, where d is the number of distinct p-characters in Q until (and including) the first occurrence of c. We can compute d in $O(1)$ time by pre-processing P in $O(|P| \log \sigma)$ time[6].

[4] Maintain a balanced binary search tree $\mathcal{T}$ for the symbols in Σ_p. Initially $\mathcal{T}$ is empty. If $P[x] \in \Sigma_p$, then $P[x]$ appears in $P[x+1, |P|]$ iff $P[x]$ is present in $\mathcal{T}$. If $P[x]$ is not in $\mathcal{T}$, insert it. Continue the process. Since $\mathcal{T}$ has $O(\sigma_p)$ nodes, search and insertion takes $O(\log \sigma)$ time.

[5] Maintain a balanced binary search tree $\mathcal{T}$ for the symbols in Σ_p. Initially $\mathcal{T}$ is empty. Also, maintain a counter C, initialized to 0. Scan P from right to left. If a p-character c_p is encountered at position x, the required count at x is C. If c_p does not belong to $\mathcal{T}$, then increment C by 1, and insert c_p into $\mathcal{T}$. Continue the process. Since $\mathcal{T}$ has $O(\sigma_p)$ nodes, search and insertion takes $O(\log \sigma)$ time.

[6] Assume Σ_p is the integer alphabet $\{1, 2, \ldots, \sigma_p\}$. Maintain an array $F[1, \sigma_p]$, all values initialized to 0, and a balanced binary search tree (BST) $\mathcal{T}$ (initially empty). Scan P from right to left. If a p-character

Consider $i, j \in [sp_1, ep_1]$ such that $i < j$ and $\mathsf{pLF}(i), \mathsf{pLF}(j) \in [sp_2, ep_2]$. Now, both f_i and f_j equals the first occurrence of c in Q. Based on Observation 4.1, we conclude that $\mathsf{pLF}(i) < \mathsf{pLF}(j)$. Therefore, where

$$sp_2 = \mathsf{pLF}(i_{min}) \text{ and } ep_2 = \mathsf{pLF}(i_{max}), \text{ where}$$
$$i_{min} = \min\{j \mid j \in [sp_1, ep_1] \text{ and } \mathsf{pBWT}[j] = d\} = \mathsf{select}(\mathsf{rank}(sp_1 - 1, d) + 1, d)$$
$$i_{max} = \max\{j \mid j \in [sp_1, ep_1] \text{ and } \mathsf{pBWT}[j] = d\} = \mathsf{select}(\mathsf{rank}(ep_1, d), d)$$

The backward search is summarized in Algorithm 3. Each wavelet tree operation (refer to Fact 2.3) takes $O(1+\log\sigma/\log\log n)$ time. The pLF operation takes $O(\log\sigma)$ time using Theorem 4.4. All other operations need $O(1)$ time (refer to Fact 2.4 and Lemma 4.3). The initial pre-processing of P takes $O(|P|\log\sigma)$ time. Therefore, we find the suffix range of $\mathsf{prev}(P)$ in $O(|P|\log\sigma)$ time. By choosing $\Delta = \lceil \log n \rceil$ in Theorem 4.3, each parameterized suffix array value can be computed in $O(\log\sigma \log n)$ by using an $O(n)$-bit data structure, in addition to the data structure in Section 4.4 for implementing Theorem 4.4. Thus, all parameterized matches of a pattern P in the text T are found in $O(|P|\log\sigma + occ \cdot \log\sigma \log n)$ time using an $n\log\sigma + O(n)$-bit data structure, which concludes the proof of Theorem 4.1. To prove Theorem 4.2, simply chose $\Delta = \lceil \log_\sigma n \rceil$ in Theorem 4.3.

---

c_p is encountered at a position x, check $F[c_p]$. If $F[c_p] = 0$, insert c_p in $\mathcal{T}$ keyed by x, else the count at x is the number of nodes in $\mathcal{T}$ with key at most $F[c_p]$. Update $F[c_p]$ and the key of c_p to x. Since $\mathcal{T}$ has $O(\sigma_p)$ nodes, search and update time is $O(\log\sigma)$.

If Σ_p is not the integer set, then the array F can be simulated via another balanced binary search tree, where we store the p-characters associated with their latest occurrence. Since this BST will have at most $O(\sigma_p)$ nodes, search and update requires $O(\log\sigma)$ time.

Algorithm 3 computes Suffix Range of $\mathsf{prev}(P[1,p])$

$c \leftarrow P[p]$, $i \leftarrow p$,
if $(c > \sigma_p)$ **then**
    $sp \leftarrow 1 + \mathsf{count}(1, n, 1, c-1)$
    $ep \leftarrow \mathsf{count}(1, n, 1, c)$
else
    $sp \leftarrow 1$
    $ep \leftarrow 1 + \mathsf{count}(1, n, 1, \sigma_p)$
while $(sp \leq ep$ and $i \geq 2)$ **do**
    $c \leftarrow P[i-1]$
    if $(c > \sigma_p)$ **then**
        $sp \leftarrow 1 + \mathsf{count}(1, n, 1, c-1) + \mathsf{count}(1, sp-1, c, c)$
        $ep \leftarrow \mathsf{count}(1, n, 1, c-1) + \mathsf{count}(1, ep, c, c)$
    else if $(c \notin P[i,p])$ **then**
        $d \leftarrow$ number of distinct p-characters in $P[i,p]$
        $sp' \leftarrow sp$
        $sp \leftarrow 1 + \mathsf{rank}_{\mathsf{falseZero}}(sp_1 - 1, 1) + \mathsf{fSum}(\mathsf{lca}(\ell_{sp}, \ell_{ep}))$
        $ep \leftarrow sp + \mathsf{count}(sp', ep, d+1, \sigma_p) - 1$
    else
        $f \leftarrow$ first occurrence of c in $P[i,p]$
        $d \leftarrow$ number of distinct p-characters in $P[i,f]$
        $i_{min} \leftarrow \mathsf{select}(\mathsf{rank}(sp-1, d) + 1, d)$
        $i_{max} \leftarrow \mathsf{select}(\mathsf{rank}(ep_1, d), d)$
        $sp \leftarrow \mathsf{pLF}(i_{min})$
        $ep \leftarrow \mathsf{pLF}(i_{max})$
if $(sp < ep)$ **then** "no match found" **else return** $[sp, ep]$

Chapter 5
Succinct Index for Parameterized Dictionary Matching

Recall the parameterized matching of two strings as outlined in Definition 4.1. We consider the *Parameterized Dictionary Indexing* problem of Idury and Schäffer [IS94].

Problem 5.1 (Parameterized Dictionary Indexing[IS94])**.** *Let $\mathcal{D}$ be a collection of d patterns $\{P_1, P_2, \ldots, P_d\}$ of total length n; each P_i contains characters from Σ. The task is to index $\mathcal{D}$, such that given a text T (also over Σ), we can report all pairs $\langle j, P_i \rangle$, such that there exists a pattern $P_i \in \mathcal{D}$ which is a parameterized match with $T[j - |P_i| + 1, j]$.*

Largely based on the Aho-Corasick (AC) automaton [AC75], Idury and Schäffer [IS94] presented an automaton which occupies $\Theta(m \log m)$ bits and reports all *occ* pairs in time $O(|T| \log \sigma + occ)$; here, $m \leq n + 1$ is the number of states in the automaton of Idury and Schäffer. Ganguly et al. [GHS$^+$16a] presented an $O(n \log \sigma + d \log n)$-bit index with $O(|T|(\log \sigma + \log_\sigma n) + occ)$ query time (see [GHS16b] for its dynamic version). We improve these results in the following theorem.

Theorem 5.1. *By using an $m \log \sigma + O(m + d \log \frac{m}{d})$-bit index of $\mathcal{D}$, all pairs $\langle j, P_i \rangle$, such that there exists a pattern $P_i \in \mathcal{D}$ which is a parameterized match with $T[j - |P_i| + 1, j]$, can be found in $O(|T| \log \sigma + occ)$ time, where occ is the number of such pairs and $m \leq n + 1$ is the number of states in the automaton of Idury and Schäffer [IS94].*

5.1 Overview of Techniques

We begin with the linear-space index of Idury and Schäffer [IS94], which is an extension of the classical AC automaton [AC75]. The idea is to first encode each pattern using Baker's encoding [Bak93] in Definition 4.2, and then create a trie $\mathcal{T}$ of these encode patterns. Each node in the trie is associated with three components: a next link that says what is the next node in the trie where we should go in case of a match, a failure link that says what is the

next node in the trie where we should go in case of a mismatch, and a report link that says what are the patterns which have an occurrence ending at the current position of the text. Section 5.2 contains the detailed description of the trie and the query process.

The first question is how to represent the nodes succinctly, in the sense that given a node, we can quickly find out if there is a pattern which has the same encoding as the path from root to this node; these nodes are termed final nodes. The other importance of node representation is to quickly simulate the three transitions above. Given a node v in the trie, let $\overleftarrow{\mathsf{prev}}(v) = \mathsf{prev}(\overleftarrow{P_i[1, \delta_v]})$, where P_i is a pattern corresponding to a final node in the subtree of v, δ_v is the node-depth of v, and $\overleftarrow{S}$ denotes the reverse of a string S. We conceptually label a node u with the lexicographic rank of $\overleftarrow{\mathsf{prev}}(u)$ in the set $\{\overleftarrow{\mathsf{prev}}(v) \mid v \text{ is a node in the trie}\}$. Labels of final nodes are explicitly stored, enabling us to find the pattern corresponding to any final node quickly. These ideas are borrowed (and extended) from Belazzougui [Bel10]. Section 5.3 contains the details.

The next task (and arguably the most difficult one) is to simulate the next transition. Here, given the label of u, we are seeking the label of a child v of u, where we should proceed in case of a match. We make the following observations. If the edge (u, v) is labeled by a p-character c, then $\overleftarrow{\mathsf{prev}}(v)$ is obtained from $\overleftarrow{\mathsf{prev}}(u)$ by prepending a 0 and appropriately modifying (if required) $\overleftarrow{\mathsf{prev}}(u)$ at the position that corresponds to the first occurrence of c. Otherwise, if c is an s-character, then $\overleftarrow{\mathsf{prev}}(v) = c \circ \overleftarrow{\mathsf{prev}}(u)$. This is extremely similar to Observation 4.1, and immediately paves way to a pBWT-like structure which can be used to carry out this kind of transition. The crucial insights are presented in Observations 5.1 and 5.2. More specifically, we create another trie $\overleftarrow{\mathcal{T}}$ of the strings $\overleftarrow{\mathsf{prev}}(w)$ for each node w in $\mathcal{T}$. Now each node u in $\mathcal{T}$ can be mapped to a leaf ℓ_u in $\overleftarrow{\mathcal{T}}$, and the label of u can be found by applying a pLF mapping-like transform on ℓ_u. Lemma 5.2 outlines the basic idea on how to implement the pLF mapping-like transform. The details are in Section 5.4.

The final piece is to simulate the failure and report transitions. However, this is rather simple and is immediate from Belazzougui [Bel10]; the details are in Section 5.5. Once,

we have all the above components, the final query procedure is largely a straightforward extension of the query procedure of Idury and Schäffer; Section 5.6 contains the details.

5.2 Idury and Schäffer's Linear Index

We recall the prev-encoding scheme of Baker [Bak93]: $\mathsf{prev}(S)$ for a string S is obtained by replacing the first occurrence of every p-character in S by 0 and any other occurrence of a p-character by the difference in text position from its previous occurrence; see Definition 4.2.

Two strings S and S' are a p-match iff $\mathsf{prev}(S) = \mathsf{prev}(S')$, and S is a p-match with a prefix of S' iff $\mathsf{prev}(S)$ is a prefix of $\mathsf{prev}(S')$; see Fact 4.1.

Note that $\mathsf{prev}(S)$ is a string over an alphabet set $\Sigma_s \cup \{0, 1, \ldots, |S| - 1\}$. We follow Convention 4.1 when comparing characters in prev-encoded strings.

5.2.1 The 3 Main Components of the Linear Index

We create a trie $\mathcal{T}$ of the strings $\{\mathsf{prev}(P_i) \mid P_i \in \mathcal{D}\}$. Let the number of nodes in the trie be m, where $m \leq n + 1$. Each edge in the trie is labeled by a single encoded character. For a node u in the trie, let $\mathsf{path}(u)$ be the string formed by concatenating the edge labels from root to u; let $\mathsf{strDepth}(u) = |\mathsf{path}(u)|$. Mark a node u in the trie as *final* iff $\mathsf{path}(u) = \mathsf{prev}(P_i)$ for some P_i in $\mathcal{D}$. Thus, the number of final nodes is d.

For any prev-encoded string $\mathsf{prev}(S)$ of a string S and an integer $j \in [1, |S|]$, we let $\zeta(\mathsf{prev}(S), j) = \mathsf{prev}(S[j, |S|])$. Note that given $\mathsf{prev}(S)$, we can obtain $\zeta(\mathsf{prev}(S), j)$ simply by assigning $\mathsf{prev}(S)[i] = 0$ for any $i \geq j$, where $\mathsf{prev}(S)[i] > i - j$ and $S[i] \in \Sigma_p$.

Each node u in the trie $\mathcal{T}$ is associated with 3 links as defined below:

- $\mathsf{next}(u, c) = v$ iff the label on the edge from the node u to v is labeled by the character c. This transition can be carried out in $O(1)$ time using a perfect hash function.

- $\mathsf{failure}(u) = v$ iff $\mathsf{path}(v) = \zeta(\mathsf{path}(u), j)$, where $j > 1$ is the smallest integer for which such a node v exists. If no such j exists, then $\mathsf{failure}(u)$ points to the root node. Conceptually, this is the smallest shift to be performed in T in case of a mismatch.

- $\mathsf{report}(u) = v$ iff v is a final node and $\mathsf{path}(v) = \zeta(\mathsf{path}(u), j)$, where $j > 1$ is the small-

est integer for which such a node v exists. If no such j exists, then $\mathsf{report}(u)$ points to the root node. Conceptually, this represents a pattern which has an occurrence ending at the current symbol of the text.

Summarizing, the total space needed by the index is $\Theta(m \log m)$ bits. Moving forward, we use the terms node and state interchangeably. Likewise, for links and transitions.

5.2.2 The Querying Algorithm

To find the occurrence pairs, we follow the steps below:

(a) First obtain $T' = \mathsf{prev}(T)$.

(b) Suppose, we are at the position j in T' (initially, $j = 1$), and we are at a node u, i.e., we have matched $T[j, j + \mathsf{path}(u) - 1]$ in the trie $\mathcal{T}$.

(c) Repeatedly follow report-links starting from u until the root node is reached. Effectively, we report all patterns with a p-match ending at $(j + \mathsf{path}(u) - 1)$.

(d) Now, look at the character $c = T'[j + \mathsf{path}(u)]$ to match. If $T'[j + \mathsf{path}(u)] \geq \mathsf{path}(u)$ and $T[j + \mathsf{path}(u)] \notin \Sigma_s$, then take $c = 0$.

(e) If $\mathsf{next}(u, c)$ is defined, then follow $\mathsf{next}(u, c)$ to a node v, and repeat the process starting from Step (b), by letting $j = j + \mathsf{strDepth}(u) - \mathsf{strDepth}(v)$.

(f) If $\mathsf{next}(u, c)$ is not defined or $j + \mathsf{path}(u) > |T|$, follow $\mathsf{failure}(u)$ to a node w, assign $j = j + \mathsf{strDepth}(u) - \mathsf{strDepth}(w)$, and repeat the process starting from Step (b).

The time required to report all occurrence pairs is $O(|T| \log \sigma + occ)$[1].

5.3 Representing States Succinctly

Belazzougui [Bel10] obtained a succinct representation of the AC automaton for the classical dictionary indexing problem [AC75]. Among other techniques, the main idea is to use a Succinctly Indexable Dictionary (SID); see Fact 2.5. The edges in the trie $\mathcal{T}$ are labeled from an alphabet of size $\Theta(m)$ in the worst case. This proves to be the primary bottleneck

[1] Initially T' is obtained in $O(|T| \log \sigma)$ time. On following a report link, either we report an occurrence, or we reach the root. Following this, either we take a next transition or we follow a $\mathsf{failure}$ link; the number of such operations combined is at most $2|T|$. Since each transition takes $O(1)$ time, the total time required is $O(|T| \log \sigma + occ)$.

that prevents us from directly applying the technique of Belazzougui [Bel10] to obtain a succinct representation of Idury and Schäffer's linear space index. More specifically, the SID in this case will need $m \log m + O(m)$ bits of space, which is not desirable.

To alleviate this problem, we first modify the label in the trie $\mathcal{T}$. Assign a bit to every edge, which is set to 1 iff the labeling on this edge corresponds to a p-character. This can be easily achieved while building $\mathcal{T}$. Accordingly, we categorize an edge as a *p-edge* or an *s-edge*. Every p-edge is in one of two states: *visited* or *new*. Initally all p-edges are new. Initialize a counter $C = 1$. Traverse from the root node to the leftmost leaf (the ordering of the leaves can be arbitrary), and modify the labeling as follows:

- Suppose, we are at a node u (initially the root), and we will traverse to the node v.
- For an s-edge (u, v), store the value of C at v, and move to the next edge on the path.
- If a p-edge (u, v) is labeled by 0, then label it by the Cth character in Σ_p (characters in Σ_p may be arbitrarily ordered). Store the value of C at v, and then increment C by 1. Move to the next edge on the path.
- If a p-edge (u, v) is labeled by $c > 0$, then assign it the same label (from Σ_p) as the $(c + 1)$th edge on the path from v to root. Store the value of C at v, and move to the next edge on the path.

After the ith leftmost leaf ℓ_i is reached, find $x = \mathsf{lca}(\ell_i, \ell_{i+1})$, and use the stored value of C at x to label the edges on the path from x to ℓ_{i+1}. By pre-processing the trie with the data structure of Fact 2.4, the entire process can be carried out in $O(m)$ time.

Observe that each edge is now labeled by a character from Σ. Let $\overleftarrow{\mathsf{path}}_\Sigma(u)$ (resp. $\mathsf{path}_\Sigma(u)$) denote the concatenation of the new edge labels on the path from u to root (resp. from root to u). With slight abuse of notation, let $\overleftarrow{\mathsf{prev}}(u) = \mathsf{prev}(\overleftarrow{\mathsf{path}}_\Sigma(u))$. Each state u is *conceptually labeled* by the lexicographic rank of $\overleftarrow{\mathsf{prev}}(u)$ in the set $\{\overleftarrow{\mathsf{prev}}(v) \mid v \text{ is a node in } \mathcal{T}\}$. Thus, each state is labeled by a number in $[1, m]$, where the root node is labeled by 1. To distinguish the final states, we use the following convention.

Convention 5.1. *For any two patterns $P_i[1, p_i]$ and $P_j[1, p_j]$ in the dictionary $\mathcal{D}$, we let $i < j$ iff $\mathsf{prev}(P_i[p_i] \circ P_i[p_i - 1] \circ \cdots \circ 1)$ is lexicographically smaller than $\mathsf{prev}(P_j[p_j] \circ P_j[p_j - 1] \circ \cdots \circ 1)$. We follow the lexicographic order of Convention 4.1 when comparing characters.*

We *explicitly store* the labels of the final states using Fact 2.5. Since there are d final nodes, the space required is $d \log(m/d) + O(d)$ bits. Now, given the label of a final state, we first find its rank among all the final states using Fact 2.5. If the rank is r, then the final state corresponds to the pattern P_r by Convention 5.1. Thus, given the label of a final state, we can find the corresponding pattern in $O(1)$ time, leading to the following lemma:

Lemma 5.1. *Given the label of a final state, we can find the corresponding pattern in $O(1)$ time by using an $d \log(m/d) + O(d)$-bit data structure.*

Lastly, we maintain a bit-vector $\mathsf{leaf}[1, m]$ such that $\mathsf{leaf}[j] = 1$ iff the state with label j is a leaf in $\mathcal{T}$. The total space for representing the states is $m + d \log(m/d) + O(d)$ bits.

5.4 Handling next Transitions Succinctly

We begin with a few notations. For any p-edge $e = (w, x)$ in $\mathcal{T}$, $w = \mathsf{parent}(x)$, we define $\mathsf{Z}(x) =$ the number of 0's in $\overleftarrow{\mathsf{prev}}(w)[1, f_x]$, where f_x is the first occurrence of the p-character labeling e in the string $\overleftarrow{\mathsf{path}}_\Sigma(w)$. If f_x is not defined, $\mathsf{Z}(x) = -z$, where z is the number of 0's in $\overleftarrow{\mathsf{prev}}(x)$. For any s-edge $e = (w, x)$ in $\mathcal{T}$, $w = \mathsf{parent}(x)$, we define $\mathsf{Z}(x) =$ the s-character labeling the edge e. We map the s-characters to the interval $[\sigma_p + 1, \sigma]$, where the ith smallest s-character has value $(\sigma_p + i)$.

Observation 5.1. *Let x_i and x_j be any two children of a node x in $\mathcal{T}$. Then, $\mathsf{Z}(x_i) \neq \mathsf{Z}(x_j)$. Also, $\overleftarrow{\mathsf{prev}}(x_i)$ is lexicographically smaller than $\overleftarrow{\mathsf{prev}}(x_j)$ iff one of the following holds:*

- $\sigma_p < \mathsf{Z}(x_i) < \mathsf{Z}(x_j)$
- $\mathsf{Z}(x_i) \leq \sigma_p < \mathsf{Z}(x_j)$
- $0 < \mathsf{Z}(x_j) < \mathsf{Z}(x_i) \leq \sigma_p$ *or* $\mathsf{Z}(x_i) < 0 < \mathsf{Z}(x_j) \leq \sigma_p$

Observation 5.2. *Suppose we are at a node u and have matched the substring $T[j, j + \mathsf{path}(u) - 1]$ in $\mathcal{T}$. Now, we want to match $c = T[j + \mathsf{path}(u)]$ and select the correct edge*

(if any). If $c \in \Sigma_s$, then clearly we need to select the s-edge (u, v) such that $\mathsf{Z}(v) = c$. Now, assume that $c \in \Sigma_p$. Let j' be the last occurrence of c in $T[1, j + \mathsf{path}(u) - 1]$; if j' is not defined, then $j' = 1$. Let z be the number of distinct p-characters in $T[j', j + \mathsf{path}(u) - 1]$. Then, we select the edge (u, v) iff one of the following is satisfied:

- $\mathsf{Z}(v) = z$; *in this case,* $j \leq j' \leq j + \mathsf{path}(u) - 1$,
- $-z \leq \mathsf{Z}(v) < 0$; *in this case,* $j' < j$

We create a compressed trie $\overleftarrow{\mathcal{T}}$ as follows. Initially $\overleftarrow{\mathcal{T}}$ is empty. For each non-leaf node u in $\mathcal{T}$ and each child u_i of u, we add the string $\overleftarrow{\mathsf{prev}}(u) \circ \$_{u,i}$ to $\overleftarrow{\mathcal{T}}$. Clearly, the string corresponds to a unique leaf, say $\ell_{u,i}$, in $\overleftarrow{\mathcal{T}}$. For any node $u \in \overleftarrow{\mathcal{T}}$, we define $\mathsf{path}(u)$ as the concatenation of the prev-encoded edge labels from the root to u.

Let $\ell_{u,i}$ denote the ith leftmost leaf in $\overleftarrow{\mathcal{T}}$, where u_i is the non-root node in $\mathcal{T}$ that corresponds to $\ell_{u,i}$, i.e., $\mathsf{path}(\ell_{u,i}) = \overleftarrow{\mathsf{prev}}(u) \circ \$_{u,i}$, and $u = \mathsf{parent}(u_i)$. We order the leaves in $\overleftarrow{\mathcal{T}}$ as follows. Consider the leaves $\ell_{u,i}$ and $\ell_{u,j}$ in $\overleftarrow{\mathcal{T}}$ such that the corresponding nodes u_i and u_j in $\mathcal{T}$ share the same parent u. (Note that $\ell_{u,i}$ and $\ell_{u,j}$ also share the same parent in $\overleftarrow{\mathcal{T}}$.) Then, the leaf $\ell_{u,i}$ lies to the left of $\ell_{u,j}$ in $\overleftarrow{\mathcal{T}}$ (i.e., $i < j$) if $\overleftarrow{\mathsf{prev}}(u_i)$ is lexicographically smaller than $\overleftarrow{\mathsf{prev}}(u_j)$. Now, consider two leaves $\ell_{u,i}$ and $\ell_{v,j}$ in $\overleftarrow{\mathcal{T}}$ with the corresponding nodes u_i and v_j in $\mathcal{T}$ having different parents u and v. Then, $\ell_{u,i}$ lies to the left of $\ell_{v,j}$ (i.e., $i < j$) if $\overleftarrow{\mathsf{prev}}(u)$ is lexicographically smaller than $\overleftarrow{\mathsf{prev}}(v)$.

Note that the number leaves in $\overleftarrow{\mathcal{T}}$ is same as the number of edges in $\mathcal{T}$. Therefore, it has $(m-1)$ leaves and at most $(m-2)$ internal nodes. Maintain the array Z of Observation 5.1, corresponding to the leaves in $\overleftarrow{\mathcal{T}}$, as a wavelet tree of Fact 2.3. Specifically, $\mathsf{Z}[i] = \mathsf{Z}(u_i)$, where u_i is the node in $\mathcal{T}$ corresponding to the ith leftmost leaf $\ell_{u,i}$ in $\overleftarrow{\mathcal{T}}$. Also, maintain a succinct representation of $\overleftarrow{\mathcal{T}}$; see Fact 2.4. The space required is $m \log \sigma + O(m)$ bits.

Define $\mathsf{leafRank}(i)$ as the rank of the string $\overleftarrow{\mathsf{prev}}(u_i)$ among the strings $\{\overleftarrow{\mathsf{prev}}(x) \mid x \in \mathcal{T} \text{ is a non-root node}\}$. Since the label of the root node is 1, the label of a node $u_i \in \mathcal{T}$ is given by $1 + \mathsf{leafRank}(i)$; recall that u_i corresponds to the ith leftmost leaf in $\overleftarrow{\mathcal{T}}$. Therefore, the main task is to compute $\mathsf{leafRank}(i)$. To facilitate this, we begin with Lemma 5.2.

Lemma 5.2. *Let u and v be two nodes (not necessarily distinct) in $\mathcal{T}$. Let u_i and v_j be their respective children (necessarily distinct) in $\mathcal{T}$, which correspond respectively to the ith and the jth leftmost leaves $\ell_{u,i}$ and $\ell_{v,j}$ in $\overleftarrow{\mathcal{T}}$.*

(a) *If* $\mathsf{Z}[i] \leq \sigma_p < \mathsf{Z}[j]$, *then* $\mathsf{leafRank}(i) < \mathsf{leafRank}(j)$.

(b) *If both* $\mathsf{Z}[i], \mathsf{Z}[j] > \sigma_p$, *then* $\mathsf{leafRank}(i) < \mathsf{leafRank}(j)$ *iff one of the following holds:*

- $\mathsf{Z}[i] < \mathsf{Z}[j]$
- $\mathsf{Z}[i] = \mathsf{Z}[j]$ *and* $i < j$. *(Note that in this case* $u \neq v$.*)*

(c) *Assume both* $\mathsf{Z}[i], \mathsf{Z}[j] \leq \sigma_p$ *and* $i < j$. *Let* $x = \mathsf{lca}(\ell_{u,i}, \ell_{v,j})$ *and* z *be the number of* 0*'s in* $\mathsf{path}(x)$.

(1) *If* $\mathsf{Z}[i] < 0$, *then* $\mathsf{leafRank}(i) < \mathsf{leafRank}(j)$

(2) *If* $0 < \mathsf{Z}[i], \mathsf{Z}[j] \leq z$, *then* $\mathsf{leafRank}(i) < \mathsf{leafRank}(j)$ *iff* $\mathsf{Z}[i] \geq \mathsf{Z}[j]$

(3) *If* $0 < \mathsf{Z}[i] \leq z < \mathsf{Z}[j]$, *then* $\mathsf{leafRank}(i) > \mathsf{leafRank}(j)$

(4) *If* $0 < \mathsf{Z}[j] \leq z < \mathsf{Z}[i]$, *then* $\mathsf{leafRank}(i) < \mathsf{leafRank}(j)$

(5) *If* $\mathsf{Z}[i], \mathsf{Z}[j] > z$, *then* $\mathsf{leafRank}(i) > \mathsf{leafRank}(j)$ *iff all of the following are true:*

- $\mathsf{Z}[i] = z + 1$,
- *the leading character on the path from* x *to* $\ell_{u,i}$ *is* 0, *and*
- *the leading character on the path from* x *to* $\ell_{v,j}$ *is not an s-character*

(6) *If* $\mathsf{Z}[j] < 0$, *then* $\mathsf{leafRank}(i) > \mathsf{leafRank}(j)$ *iff either* $0 < \mathsf{Z}[i] \leq z$ *or if all of the following are true:*

- $\mathsf{Z}[i] = z + 1$,
- *the leading character on the path from* x *to* $\ell_{u,i}$ *is* 0, *and*
- *the leading character on the path from* x *to* $\ell_{v,j}$ *is not an s-character*

Proof. The lemma is a direct consequence of how the leaves are arranged in $\overleftarrow{\mathcal{T}}$ in conjunction with Observation 5.1 and Convention 4.1. ■

Crucially, Lemma 5.2 is similar to Lemma 4.1 with the following changes:

- pBWT replaced by Z
- pLF mapping replaced by $\mathsf{leafRank}$, and
- pST replaced by $\overleftarrow{\mathcal{T}}$.

Computing $\mathsf{leafRank}(i)$ is handled by the following lemma (proof is similar to that for computing pLF-mapping and is deferred to Section 5.4.1):

Lemma 5.3. *By using the wavelet tree over* Z *and an additional* $O(m)$*-bit data structure, we can compute* $\mathsf{leafRank}(i)$ *for the* i*th leftmost leaf in* $\overleftarrow{\mathcal{T}}$ *in* $O(\log \sigma)$ *time.*

Suppose, we are at a node u in $\mathcal{T}$. Given a $c \in [1, \sigma]$, our next task is to select (if exists) the leaf $\ell_{u,i}$ in $\overleftarrow{\mathcal{T}}$ that corresponds to the child u_i of u, where $\mathsf{Z}(u_i) = c$. This is handled by the following lemma:

Lemma 5.4. *Let* u *be a non-leaf node in* $\mathcal{T}$ *having label* k*. By using the wavelet tree over* Z *and an additional* $O(m)$*-bit data structure, we can support the following in* $O(\log \sigma)$ *time:*

- *find the node* v *in* $\overleftarrow{\mathcal{T}}$ *such that* $\mathsf{path}(v) = \overleftarrow{\mathsf{prev}}(u)$
- *given* $z \in [1, \sigma]$*, we can detect if* u *has a child* u_i *such that* $\mathsf{Z}(u_i) = z$*. We can also find the leaf* $\ell_{u,i} \in \overleftarrow{\mathcal{T}}$ *corresponding to* u_i*.*

Proof. We create a binary string B as follows. Traverse the leaves from left to right in $\overleftarrow{\mathcal{T}}$, and append a 1-bit for each leaf, followed by a 0-bit whenever we switch from a leaf to the next which has a different parent. Append a 0 both at the beginning and at the end, and maintain the $\mathsf{rank\text{-}select}$ structure of Fact 2.2. The space needed is $O(m)$ bits.

Consider the nodes $\mathcal{V} = \{v_1, v_2, \ldots, v_t\}$ in $\overleftarrow{\mathcal{T}}$ in increasing order of their pre-order rank, where each v_x is the parent of a leaf in $\overleftarrow{\mathcal{T}}$. Let u_x be the node in $\mathcal{T}$ corresponding to v_x, i.e., $\mathsf{path}(v_x) = \overleftarrow{\mathsf{prev}}(u_x)$. Then, the label of u_x is smaller than the label of u_y iff $1 \leq x < y \leq t$. Therefore, the desired node v is the k'th node in $\mathcal{V}$, where k' is the number of non-leaf nodes in $\mathcal{T}$ with label at most k, i.e., $k' = \mathsf{rank}_{\mathsf{leaf}}(k, 0) > 0$ (follows from the way leaves

are arranged in $\overleftarrow{\mathcal{T}}$). Also, u_i exists iff $\mathsf{Z}[x] = z$ for some $x \in [\mathsf{lmostLeaf}(v), \mathsf{rmostLeaf}(v)]$. If u_i exists then $\ell_{u,i}$ is the xth leftmost leaf in $\overleftarrow{\mathcal{T}}$.

We locate the leaves ℓ_{sp} and ℓ_{ep} in $\overleftarrow{\mathcal{T}}$, where $sp = \mathsf{rank}_B(\mathsf{select}_B(k', 0), 1) + 1$, and $ep = \mathsf{rank}_B(\mathsf{select}_B(k' + 1, 0), 1)$. Then $v = \mathsf{parent}(\ell_{sp})$. Obtain $q = \mathsf{select}_{\mathsf{Z}}(1 + \mathsf{rank}_{\mathsf{Z}}(sp - 1, z), z)$ in $O(\log \sigma)$ time. If $q > ep$, then u_i does not exist. Otherwise, the desired leaf $\ell_{u,i}$ is given by $\mathsf{child}(\mathsf{parent}(\ell_{sp}), q - sp + 1)$. ■

We now have sufficient arsenal to prove the following lemma.

Lemma 5.5. *Let $T_0 = T[j_0, |T|]$, $1 \le j_0 \le |T|$. Suppose we are at a node u in $\mathcal{T}$ and have matched $T_0[1, j]$. Given the label k of u, we can find the label (if any) of $\mathsf{next}(u, \mathsf{prev}(T_0)[j + 1])$ in $O(\log \sigma)$ amortized time by using an $m \log \sigma + O(m)$-bit data structure.*

Proof. If u is a leaf (i.e., $\mathsf{leaf}[k] = 1$), then u does not have a child and the lemma trivially follows. So, assume otherwise. Let $c = T_0[j + 1]$. If c is a p-character, then let z be the number of distinct p-characters in $T[j', j_0 + j]$, where $j' = 1$ if $(j_0 + j)$ is the first occurrence of c in T, else j' is the maximum position where c occurs in $T[1, j_0 + j - 1]$. Note that z can be computed in $O(\log \sigma)$ time per character[2]. If c is an s-character, then let $z = c$.

Now, we apply Observation 5.2 to find the label k' of $u' = \mathsf{next}(u, \mathsf{prev}(T_0)[j + 1])$, if exists. Using Lemma 5.4, we first detect if u has a child u_i such that $\mathsf{Z}(u_i) = z$. If u_i exists, then $u' = u_i$. To compute the label of $u' = u_i$, we first find the corresponding leaf $\ell_{u,i}$ using Lemma 5.4, and then compute $k' = 1 + \mathsf{leafRank}(i)$ using Lemma 5.3. If u_i does not exist and $c \in \Sigma_p$, we find the node v in $\overleftarrow{\mathcal{T}}$ corresponding to u by using Lemma 5.4. Now, u' exists iff $-z \le \mathsf{Z}(u') < 0$. Also, based on how leaves are arranged in $\overleftarrow{\mathcal{T}}$, if u' exists, then it

[2] Assume Σ_p is the integer alphabet $\{1, 2, \ldots, \sigma_p\}$. We maintain an array $A[1, \sigma_p]$, where $A[c]$ is the position of the last occurrence of the p-character $c \in \Sigma_p$. Initially, $A[c] = -1$ for every $c \in [1, \sigma_p]$. Maintain a balanced binary search tree $\mathcal{T}_{bin}$ containing only the p-characters c indexed by $A[c]$. Note that the size of $\mathcal{T}_{bin}$ is $O(\sigma_p)$, which implies update and search operation requires $O(\log \sigma_p)$ time.

Suppose we are looking at a p-character $c = T[k]$. If $A[c] = -1$, we add c to $\mathcal{T}_{bin}$, and find the number of characters in $\mathcal{T}_{bin}$, which gives us the desired encoding. Otherwise, find the number of entries in $\mathcal{T}_{bin}$ with key at least $A[c]$, which gives us the desired encoding. Finally, update $A[c] = k$ and proceed.

If Σ_p is not the integer set, then the array A can be simulated via another balanced binary search tree, where we store the p-characters associated with their latest occurrence. Since this BST will have at most $O(\sigma_p)$ nodes, search and update requires $O(\log \sigma)$ time.

corresponds to the leftmost leaf in v's subtree. Let $l = \mathsf{lmostLeaf}(v)$. If $-z \leq \mathsf{Z}[l] < 0$, then $k' = 1 + \mathsf{leafRank}(l)$, else u' does not exist. The time required is amortized $O(\log \sigma)$. ■

5.4.1 Computing leafRank

Recall that in the data structure to compute pLF mapping, we only use the property that pST is a compressed trie (i.e., marking scheme, and the structure for computing fSum and zeroNode only use this property). Hence, our techniques for computing leafRank remains largely similar to that for computing pLF mapping. We include the details for completion.

We define f_j as the first occurrence (if any) of the p-character labeling the p-edge $(u_j, \mathsf{parent}(u_j))$ on the path from $\mathsf{parent}(u_j)$ to the root in $\mathcal{T}$. For any node $u \in \overleftarrow{\mathcal{T}}$, we define $\mathsf{zeroDepth}(u)$ as the number of 0's in $\mathsf{path}(u)$.

For a leaf $\ell_{u,i} \in \overleftarrow{\mathcal{T}}$, let $\mathsf{zeroNode}(\ell_{u,i})$ be the highest node v on the root to $\ell_{u,i}$ path that satisfies $\mathsf{zeroDepth}(v) \geq \mathsf{Z}[i]$. Note that $\mathsf{zeroNode}(\ell_{u,i})$ is the root when $\mathsf{Z}[i] < 0$. We shall need the following (proof deferred to Section 5.4.2):

Lemma 5.6. *We can find* $\mathsf{zeroNode}(\ell_{u,i})$ *in* $O(\log \sigma)$ *time by using a wavelet tree over the array* Z *and an additional* $O(m)$*-bit data structure.*

We will define the following additional functionality later: $\mathsf{leafLeadChar}(\cdot)$, $\mathsf{fSum}(\cdot)$ and $\mathsf{pCount}(\cdot)$. Each of these can be computed in $O(1)$ time using an $O(n)$-bit data structure.

Computing $\mathsf{leafRank}(i)$ ***when*** $\mathsf{Z}[i] \in [\sigma_p + 1, \sigma]$**.** Using Lemma 5.2, we conclude that $\mathsf{leafRank}(i) > \mathsf{leafRank}(j)$ iff either $j \in [1, n]$ and $\mathsf{Z}[j] < \mathsf{Z}[i]$, or $j \in [1, i-1]$ and $\mathsf{Z}[i] = \mathsf{Z}[j]$. Then,

$$\mathsf{leafRank}(i) = 1 + \mathsf{count}(1, n, 1, \mathsf{Z}[i] - 1) + \mathsf{count}(1, i-1, \mathsf{Z}[i], \mathsf{Z}[i])$$

Computing $\mathsf{leafRank}(i)$ ***when*** $\mathsf{Z}[i] \in [1, \sigma_p]$**.** Let $z = \mathsf{zeroNode}(\ell_{u,i})$ and $v = \mathsf{parent}(z)$. Then, $f_i = (\mathsf{strDepth}(v) + 1)$ if the leading character on the edge from v to z is 0 and $\mathsf{Z}[i] = (\mathsf{zeroDepth}(v) + 1)$; otherwise, $f_i > (\mathsf{strDepth}(v) + 1)$. For a leaf $\ell_{u',j}$ in $\overleftarrow{\mathcal{T}}$, $\mathsf{leafLeadChar}(j)$

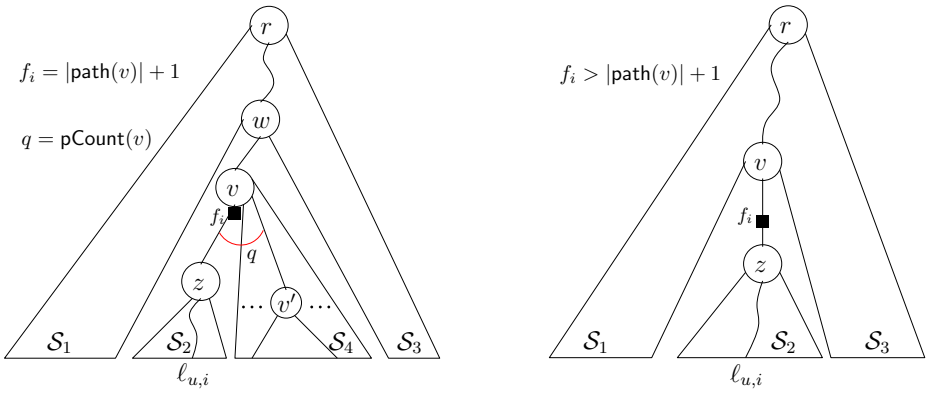

Figure 5.1: Computing $\mathsf{leafRank}(i)$ when $0 < \mathsf{Z}[i] \leq \sigma_p$

is a boolean variable, which is 0 iff $f_j = (\mathsf{strDepth}(\mathsf{parent}(\mathsf{zeroNode}(\ell_{u',j}))) + 1)$. Now, we can determine which of the following two cases is satisfied in $O(1)$ time (see Figure 5.1):

Sub-case 1 ($f_i = \mathsf{strDepth}(v) + 1$). In this case, z is the leftmost child of v. Let w be the parent of v. We partition the leaves into four sets:

(a) $\mathcal{S}_1$: leaves to the left of the subtree of v.

(b) $\mathcal{S}_2$: leaves in the subtree of z.

(c) $\mathcal{S}_3$: leaves to the right of the subtree of v.

(d) $\mathcal{S}_4$: leaves in the subtree of v but not of z.

In case, v is the root node r, we take $w = r$; consequently, $\mathcal{S}_1 = \mathcal{S}_3 = \varnothing$.

Sub-case 2 ($f_i > \mathsf{strDepth}(v) + 1$). We partition the leaves into three sets:

(a) $\mathcal{S}_1$ (resp. $\mathcal{S}_3$): leaves to the left (resp. right) of the subtree of z.

(b) $\mathcal{S}_2$: leaves in the subtree of z.

We first compute $z = \mathsf{zeroNode}(\ell_{u,i})$ using Lemma 5.6 and then locate $v = \mathsf{parent}(z)$. Using $\mathsf{leafLeadChar}(i)$ and the $\mathsf{lmostLeaf}(\cdot)/\mathsf{rmostLeaf}(\cdot)$ tree operations, we find the desired ranges. Let $[L_x, R_x]$ denote the range of leaves in the subtree of any node x, i.e., $L_x = \mathsf{lmostLeaf}(x)$ and $R_x = \mathsf{rmostLeaf}(x)$. In order to compute $\mathsf{leafRank}(i)$, we first compute

N_1, N_2, and N_3, which are respectively the number of leaves ℓ_j in the ranges $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_3$ such that $\mathsf{leafRank}(j) \leq \mathsf{leafRank}(i)$. Likewise, we compute N_4 (w.r.t $\mathcal{S}_4$) if we are in the first case. Then, $\mathsf{leafRank}(i)$ is computed using $\mathsf{leafRank}(i) = N_1 + N_2 + N_3 + N_4$.

Now, we undertake the task of computing N_1 through N_4. In each case, the claims are direct consequences of Lemma 5.2.

Computing N_1. For any leaf $\ell_{u',j} \in \mathcal{S}_1$, $\mathsf{leafRank}(j) < \mathsf{leafRank}(i)$ if either (1) $f_j > 1 + \mathsf{strDepth}(\mathsf{lca}(z, \ell_j))$ and $0 < \mathsf{Z}[j] \leq \sigma_p$, or (2) $\mathsf{Z}[j] < 0$. Therefore, N_1 is the number of leaves $\ell_{u',j}$, such that either (1) $0 < \mathsf{Z}[j] \leq \sigma_p$ and $\ell_{u',j}$ comes before z in pre-order with $f_j > 1 + \mathsf{strDepth}(\mathsf{lca}(z, \ell_j))$, or (2) $\mathsf{Z}[j] < 0$.

Define, $\mathsf{fCount}(x)$ of a node x as the number of leaves $\ell_{u',j}$ in x's subtree such that $\mathsf{strDepth}(y) + 2 \leq f_j \leq \mathsf{strDepth}(x) + 1$, where $y = \mathsf{parent}(x)$ and $0 < \mathsf{Z}[j] \leq \sigma_p$. If x is the root node, then $\mathsf{fCount}(x) = 0$. Define $\mathsf{fSum}(x)$ of a node x as $\sum \mathsf{fCount}(y)$ of all nodes y which come before x in pre-order and are not ancestors of x. Then,

$$N_1 = \mathsf{fSum}(z) + \mathsf{count}(1, L_z - 1, -\sigma_p, -1),$$

where $\mathsf{fSum}(z)$ is computed as follows.

Lemma 5.7. *By maintaining an $O(n)$-bit structure, we can compute $\mathsf{fSum}(x)$ in $O(1)$ time.*

Proof. Traverse $\overleftarrow{\mathcal{T}}$ in DFS order. Append $\mathsf{fCount}(v)$ in unary[3] to an initially empty binary string B when exiting the subtree rooted at node v in the traversal, i.e., $\mathsf{fCount}(v)$ is associated with $\mathsf{post\text{-}order}(v)$. Maintain the rank-select structure of Fact 2.2 on B. Since $\sum_v \mathsf{fCount}(v) \leq n$, we have $|B| \leq 3n$. Thus, the space needed is $3n + o(n)$ bits. Note that $\mathsf{fSum}(x)$ is same as the number of 1s in B up to the position corresponding to a node y, where the node y is conceptually found as follows. Traverse upwards from x to the root node until we reach a node y' which has a child to the left of the path. Then y is the rightmost child of y'

[3]Unary encoding of a number x is a string containing x number of 1s followed by a 0.

that lies to the left of the path. If $L_x = 1$, then y is undefined and $\mathsf{fSum}(x) = 0$. Otherwise, use Facts 2.2 and 2.4 to compute $y = \mathsf{levelAncestor}(\ell_{L_x-1}, \mathsf{nodeDepth}(\mathsf{lca}(\ell_{L_x}, \ell_{L_x-1})) + 1)$ and $\mathsf{fSum}(x) = \mathsf{rank}_B(\mathsf{select}_B(\mathsf{post\text{-}order}(y), 0), 1)$ in $O(1)$ time. ■

Computing N_2. For any leaf $\ell_{u',j} \in \mathcal{S}_2$, $\mathsf{leafRank}(j) \leq \mathsf{leafRank}(i)$ iff one of the following is satisfied: (1) $0 < \mathsf{Z}[j] \leq \sigma_p$ and $f_j > f_i$, or (2) $0 < \mathsf{Z}[j] \leq \sigma_p$, $f_j = f_i$, and $j \leq i$, or (3) $\mathsf{Z}[j] < 0$. Therefore, N_2 is the number of leaves $\ell_{u',j}$ in $\mathcal{S}_2$ which satisfy one of the following conditions: (1) $\mathsf{Z}[i] < \mathsf{Z}[j] \leq \sigma_p$, or (2) $\mathsf{Z}[i] = \mathsf{Z}[j]$ and $j \leq i$, or (3) $\mathsf{Z}[j] < 0$. Then,

$$N_2 = \mathsf{count}(L_z, R_z, \mathsf{Z}[i] + 1, \sigma_p) + \mathsf{count}(L_z, i, \mathsf{Z}[i], \mathsf{Z}[i]) + \mathsf{count}(L_z, R_z, -\sigma_p, -1)$$

Computing N_3. For any leaf $\ell_{u',j} \in \mathcal{S}_3$, $\mathsf{leafRank}(j) > \mathsf{leafRank}(i)$. Thus, $N_3 = 0$.

Computing N_4. Note that $\mathsf{Z}[i] = \mathsf{zeroDepth}(v)+1$. Consider a leaf $\ell_{u',j} \in \mathcal{S}_4$ with $\mathsf{Z}[j] \leq \sigma_p$. Since the root to $\ell_{u',j}$ path deviates from the root to $\ell_{u,i}$ path at the node v, we have $f_j \neq f_i$. Therefore, $\mathsf{leafRank}(j) < \mathsf{leafRank}(i)$ iff the leading character on the path from v to ℓ_j is not an s-character and either (1) $f_j > f_i$, or (2) $\mathsf{Z}[j] < 0$. For a node x, $\mathsf{pCount}(x)$ is the number of children y of x such that the leading character from x to y is not an s-character. Note that $\sum_x \mathsf{pCount}(x) = O(n)$. Therefore, we encode $\mathsf{pCount}(\cdot)$ of all nodes in $O(n)$ bits using unary encoding, such that $\mathsf{pCount}(x)$ can be retrieved in constant time[4]. Let u be the $\mathsf{pCount}(v)$th child of v. Then, N_4 is the number of leaves ℓ_j in $\mathcal{S}_4$ such that $j \leq R_u$ and either (1) $\sigma_p \geq \mathsf{Z}[j] \geq \mathsf{Z}[i]$, or (2) $\mathsf{Z}[j] < 0$. Thus,

$$N_4 = \mathsf{count}(R_z + 1, R_u, \mathsf{Z}[i], \sigma_p) + \mathsf{count}(R_z + 1, R_u, -\sigma_p, -1)$$

[4] Create a binary string S as follows. For each node u in the pre-order traversal of pST, append to S a 0 followed by $\mathsf{pCount}(u)$ number of 1s. Append a 0 at the end. Maintain a $\mathsf{rank\text{-}select}$ structure (refer to Fact 2.2) over S. Then $\mathsf{pCount}(u)$ for a node u, having pre-order rank k, is the number of 1s between the kth 0 and the $(k+1)$th 0. The value is given by $\mathsf{rank}(\mathsf{select}(k+1, 0), 1) - \mathsf{rank}(\mathsf{select}(k, 0), 1)$ in $O(1)$ time.

Algorithm 4 computes $\mathsf{leafRank}(i)$

1: $c \leftarrow \mathsf{Z}[i]$
2: **if** $(c > \sigma_p)$ **then**
3: $\quad \mathsf{leafRank}(i) \leftarrow 1 + \mathsf{count}(1, n, 1, c-1) + \mathsf{count}(1, i-1, c, c)$
4: **else if** $c < 0$ **then**
5: $\quad \mathsf{leafRank}(i) \leftarrow 1 + \mathsf{fSum}(\ell_{u,i}) + \mathsf{count}(1, i-1, -\sigma_p, -1)$
6: **else**
7: $\quad z \leftarrow \mathsf{zeroNode}(\ell_i)$, $v \leftarrow \mathsf{parent}(z)$, $L_z \leftarrow \mathsf{lmostLeaf}(z)$, $R_z \leftarrow \mathsf{rmostLeaf}(z)$
8: $\quad N_1 \leftarrow \mathsf{fSum}(z) + \mathsf{count}(1, L_z - 1, -\sigma_p, -1)$
9: $\quad N_2 \leftarrow \mathsf{count}(L_z, R_z, c+1, \sigma_p) + \mathsf{count}(L_z, R_z, -\sigma_p, -1) + \mathsf{count}(L_z, i, c, c)$
10: $\quad$ **if** ($\mathsf{leafLeadChar}(i)$ is 0) **then**
11: $\quad\quad u \leftarrow \mathsf{child}(v, \mathsf{pCount}(v))$, $R_u \leftarrow \mathsf{rmostLeaf}(u)$
12: $\quad\quad N_4 \leftarrow \mathsf{count}(R_z + 1, R_u, c, \sigma_p) + \mathsf{count}(R_z + 1, R_u, -\sigma_p, -1)$
13: $\quad \mathsf{leafRank}(i) \leftarrow N_1 + N_2 + N_4$

Computing $\mathsf{leafRank}(i)$ ***when*** $\mathsf{Z}[i] < 0$**.** Consider a leaf $\ell_{u',j}$. If $j < i$, then $\mathsf{leafRank}(j) < \mathsf{leafRank}(i)$ iff either (1) $f_j > 1 + \mathsf{strDepth}(\mathsf{lca}(\ell_{u',j}, \ell_{u,i}))$, or (2) $\mathsf{Z}[j] < 0$. If $j > i$, then $\mathsf{leafRank}(j) > \mathsf{leafRank}(i)$. Thus,

$$\mathsf{leafRank}(i) = 1 + \mathsf{fSum}(\ell_{u,i}) + \mathsf{count}(1, i-1, -\sigma_p, -1)$$

We summarize the $\mathsf{leafRank}$ computation procedure in Algorithm 4. The data structure occupies $m \log \sigma + O(n)$ bits. If $\mathsf{Z}[i] > \sigma_p$ or $\mathsf{Z}[i] < 0$, then $\mathsf{leafRank}(i)$ is computed in $O(1 + \log \sigma / \log\log m)$ time. Otherwise, we first compute $\mathsf{zeroNode}(\ell_{u,i})$ in $O(\log \sigma)$ time using Lemma 5.6. Now, we find the desired ranges $\mathcal{S}_1$, $\mathcal{S}_2$, $\mathcal{S}_3$, and if required $\mathcal{S}_4$, all in $O(1)$ time. Then, N_1 is computed in $O(1)$ time, and both N_2 and N_4 are computed in $O(1 + \log \sigma / \log\log m)$ time. Thus, $\mathsf{leafRank}(i)$ is computed in $O(\log \sigma)$ time when $\mathsf{Z}[i] \in [1, \sigma_p]$. This concludes the proof of Lemma 5.3.

5.4.2 Finding ZeroNode

We prove Lemma 5.6 in this section. For any node x, let $L_x = \mathsf{lmostLeaf}(x)$ and $R_x = \mathsf{rmostLeaf}(x)$. For any node x on the root to $\ell_{u,i}$ path π, define $\alpha(x)$ = the number of leaves $\ell_{v,j}$, $j \in [L_x, R_x]$ such that $\mathsf{Z}[j] \leq \sigma_p$ and $f_j \leq \mathsf{strDepth}(x)$, and $\beta(x) = \mathsf{count}(L_x, R_x, 1, \mathsf{Z}[i])$. Consider a node w on π. Observe that $\mathsf{zeroNode}(\ell_{u,i})$ is below w

iff $\beta(w) > \alpha(w)$. Therefore, $\mathsf{zeroNode}(\ell_{u,i})$ is the shallowest node w' on this path that satisfies $\beta(w') \leq \alpha(w')$. Equipped with this knowledge, now we can binary search on π (using $\mathsf{nodeDepth}$ and $\mathsf{levelAncestor}$ operations) to find the exact location. The first question is to compute $\alpha(x)$, which is handled by Lemma 5.8. A normal binary search will have to consider m nodes on the path in the worst case. Lemma 5.9 shows how to reduce this to $\lceil \log \sigma \rceil$. Note that $m \geq \sigma$. Thus, the binary search has at most $\lceil \log \log \sigma \rceil$ steps, and the total time is $\log \log \sigma \times \lceil \frac{\log \sigma}{\log \log m} \rceil = O(\log \sigma)$, as required.

Lemma 5.8. *By maintaining the wavelet tree over* Z *and an additional* $O(m)$*-bit data structure, we can find* $\alpha(x)$ *in* $O(1 + \frac{\log \sigma}{\log \log m})$ *time.*

Proof. Let $A[1, m]$ be a bit-vector such that $A[i] = 1$ iff $0 < \mathsf{Z}[i] \leq \sigma_p$. Define $\gamma(v)$ as the number of leaves ℓ_j, $j \in [L_v, R_v]$ that satisfy $0 < \mathsf{Z}[j] \leq \sigma_p$ and $\mathsf{strDepth}(\mathsf{parent}(v)) < f_j \leq \mathsf{strDepth}(v)$. Traverse $\overleftarrow{\mathcal{T}}$ in DFS order. Append $\gamma(v)$ in unary to an initially empty binary string B when entering the subtree of node v in the traversal, i.e., $\gamma(v)$ is associated with $\mathsf{pre\text{-}order}(v)$. Since $\sum_v \gamma(v) \leq m$, we have $|B| \leq 3m$. Maintain the $\mathsf{rank\text{-}select}$ structure of Fact 2.2 on the bit-vectors A and B. The total space needed is $4m + o(m)$ bits. Let $\alpha'(x)$ be the number of leaves $\ell_j \in [L_x, R_x]$ such that $0 < \mathsf{Z}[j] \leq \sigma_p$ and $f_j > \mathsf{strDepth}(x)$. Then,

$$l = \mathsf{select}_B(\mathsf{pre\text{-}order}(x), 0) \text{ and } r = \mathsf{select}_B(\mathsf{pre\text{-}order}(\ell_{R_x}), 0)$$
$$\alpha'(x) = \mathsf{rank}_B(l, r, 1) + \mathsf{count}(L_x, R_x, -\sigma_p, -1)$$
$$\alpha(x) = \mathsf{rank}_A(L_x, R_x, 1) - \alpha'(x)$$

Clearly, the space-and-time bounds are met. ■

Lemma 5.9. *Using the wavelet tree over* Z *and an additional* $O(m)$*-bit data structure, in* $O(\log \sigma)$ *time, we can find an ancestor* w_i *of* $\ell_{u,i}$ *such that* $\mathsf{zeroDepth}(w_i) < \mathsf{Z}[i]$ *and* w_i *is at most* $\lceil \log \sigma \rceil$ *nodes above* $\mathsf{zeroNode}(\ell_{u,i})$*, where* $\sigma_p \geq \mathsf{Z}[i] > 0$*.*

Proof. Let $g = \lceil \log \sigma \rceil$. We mark the nodes v in the $\overleftarrow{\mathcal{T}}$ such that $\mathsf{nodeDepth}(v)$ is a multiple of g and the subtree of v has at least g nodes. Also, mark the root node. It is easy to

see that (i) between any two closest marked nodes (or a lowest marked node and a leaf in its subtree) there are at most g nodes, and (ii) the number of marked nodes is $O(m/g)$. Maintain a bit-vector B such that $B[k] = 1$ iff the node with pre-order rank k is a marked node. Also, maintain the data structure of Fact 2.2 on B. The space needed is $O(m)$ bits. Let D be an array, such that $D[k]$ equals the $\mathsf{zeroDepth}$ of the marked node corresponding to the kth 1-bit in B. Given a marked node with pre-order rank k', its corresponding position in D is given by $\mathsf{rank}_B(k', 1)$. We maintain the wavelet tree of Fact 2.3 over D. The space needed is $O(\frac{m}{g} \log \sigma) = O(m)$ bits.

To find the desired node w_i, we locate the lowest marked ancestor u of ℓ_i by traversing the tree upwards via the parent operation. We can check if a node is marked using B in $O(1)$ time. The time to find u is $O(\log \sigma)$. Let j be the position corresponding to u in the array D, i.e., $j = \mathsf{rank}_B(\mathsf{pre\text{-}order}(u), 1)$. If $\mathsf{zeroDepth}(u) = D[j] < \mathsf{Z}[i]$, assign $w_i = u$. Otherwise, we locate the rightmost position $j' < j$ in D such that $D[j'] < \mathsf{Z}[i]$ in $O(\log \sigma)$ time using the query $j' = \mathsf{predecessor}_D(j, \mathsf{Z}[i] - 1)$. (Since the root r is marked and $\mathsf{zeroDepth}(r) = 0$, the position j' exists.) Obtain the marked node v corresponding to the j'th 1-bit in B via a $\mathsf{select}_B(j', 1)$ operation. Assign $w_i = \mathsf{lca}(u, v)$. The time required is $O(\log \sigma)$. To see the correctness, observe that $\mathsf{lca}(u, v)$ is an ancestor of ℓ_i. For a node x, $\mathsf{zeroDepth}(x) \geq \mathsf{zeroDepth}(\mathsf{parent}(x))$. Thus, $\mathsf{zeroDepth}(\mathsf{lca}(u, v)) \leq \mathsf{zeroDepth}(v) < \mathsf{Z}[i]$. If w_i has been incorrectly chosen, then $\mathsf{lca}(u, v)$ has a marked descendant $u' \neq u$ on the path to u such that $\mathsf{zeroDepth}(u') < \mathsf{Z}[i]$. But u' appears after v and before u in pre-order, which is a contradiction. ■

5.5 Handling failure and report Transitions Succinctly

Note that for any two nodes u and v, if $\mathsf{failure}(u) = v$, then it $\overleftarrow{\mathsf{prev}}(v)$ is the longest prefix of $\overleftarrow{\mathsf{prev}}(u)$ that appears in $\mathcal{T}$. Similar remarks hold for $\mathsf{report}(u) = v$, where v is a final node. Therefore, these transitions behave exactly in the same manner as in the case of traditional pattern matching, and we can simply use the idea of Belazzougui to perform these transitions (see Sections 3.3 and 3.4 in [Bel10]). We get the following lemma.

Lemma 5.10. *By using an $O(m + d\log(m/d))$-bit data structure, given the label of a node u, we can find the label of* $\mathsf{failure}(u)$ *or* $\mathsf{report}(u)$ *in $O(1)$ time.*

5.6 The Final Query Procedure

Suppose we are at a node u in $\mathcal{T}$ with label k. (Initially we are at the root node with label 1.) Suppose, we have matched $T[j', j-1]$. First repeatedly apply Lemma 5.10 to find the label of the final states reached by the report links. The occurrences of all the patterns ending at the position $j-1$ corresponding to these final states are found using Lemma 5.1. Now, use Lemma 5.5 to find the label k_v of the node (if any) v such that $v = \mathsf{next}(u, T'[j-j'+1])$, where $T' = \mathsf{prev}(T[j', |T|])$. If v exists, then repeat the process with node v and label k_v. If v does not exist, then find the label k_w of the node (if any) w such that $w = \mathsf{failure}(u)$, and repeat the process with node w and label k_w. The total time required is $O(|T|\log\sigma + occ)$[5]. The total space needed to implement Lemmas 5.1, 5.5, and 5.10 is $m\log\sigma + O(m + d\log\frac{m}{d})$ bits. This concludes the proof of Theorem 5.1.

[5] When we follow a failure link we remove at least one character from the beginning of the text. When we follow a next link we advance one character at the beginning of the text. The number of such operations is at most $2|T|$. On following each report link, we report an occurrence. Each failure and report transition needs $O(1)$ time, and each next transition needs $O(\log\sigma)$ time.

Chapter 6
Compact Index for Order-preserving Pattern Matching

We begin with the definition of order-preserving matching. For the sake of simplicity, we assume $\Sigma = \{1, 2, \ldots, \sigma\}$.

Definition 6.1 (Order-preserving Matching [KEF+14]). *Two equal-length strings S and S' over Σ are an order-preserving match (o-match) iff for any two positions $i, j \in [1, |S|]$,*

- $S[i] < S[j] \iff S'[i] < S'[j]$
- $S[i] = S[j] \iff S'[i] = S'[j]$
- $S[i] > S[j] \iff S'[i] > S'[j]$

Thus, the strings $S = 12231$ and $S' = 37793$ are order-preserving. However, $S = 12231$ and $S' = 34452$ are not order-preserving (because $S[1] = S[5] = 1$ and $S'[1] > S'[5]$).

Introduced by Crochemore et al. [CIK+13], we consider the following problem:

Problem 6.1 (Order-preserving Text Indexing [CIK+13]). *Let T be a text of length n over $\Sigma = \{1, 2, \ldots, \sigma\}$. Index T, such that given a pattern P (also over Σ), we can report all the o-occurrences of P, i.e., all the starting positions of the substrings of T that are an o-match with P.*

Crochemore et al. [CIK+13] presented a $\Theta(n \log n)$-bit index for this problem with a query time of $O(|P| \log \sigma + occ)$. We present the following compact index.

Theorem 6.1. *By using an $O(n \log \sigma)$-bit index of T, all the o-occurrences of P can be found in $O(|P| \log \sigma \log \log n + (\log \log n + occ) \log n \log_\sigma n)$ time, where occ is the number of such o-occurrences.*

6.1 Overview of Techniques

For the parameterized text indexing problem, we have seen that the main component of the linear-space index is an encoding scheme such that two strings are a p-match iff the

corresponding encoded strings are a match in the traditional sense. Using the encoding scheme, we can design a suffix tree variant (p-suffix tree), which allows us to answer a pattern matching query efficiently. Likewise, for Problem 6.1, a similar encoding scheme is not only desirable, but is in some sense necessary, as without such an encoding scheme, it is almost inconceivable that we can even get a suffix-tree like solution, let alone compressed indexes. Luckily, for order-preserving matching, there exists two encoding schemes, one by Kim et al. [KEF+14] and the other by Crochemore et al. [CIK+13]. Needless to say, both the schemes can be computed efficiently for any input string. Therefore, a suffix-tree like solution is immediate. First encode each suffix with either of the encoding schemes, and then create a compacted trie (called the *order-preserving suffix tree*, or o-suffix tree) with this encoded suffixes. To find the occurrences, search the o-suffix tree with the encoded pattern to first find the highest node u in the o-suffix tree such that the string obtained by concatenating the edge labels from root to u is prefixed by the encoded pattern. Then, we report the starting positions of the encoded suffixes corresponding to the leaves in the subtree of u; this is obtained using the order-isomorphic suffix array (o-suffix array) which stores the starting positions of the lexicographically arranged encoded suffixes. A linear-space index using the encoding of Crochemore et al. [CIK+13] is presented in Section 6.2.

Recall that LF mapping of a leaf i in the suffix tree is a leaf j, i.e., $\mathsf{LF}(i) = j$ iff $\mathsf{SA}[j] = \mathsf{SA}[i]-1$; see Section 2.3.1 for more details. The key component behind compressing the p-suffix tree is a modified form of LF mapping. One may think that to compress the o-suffix tree, we again need some version of LF mapping, and that is indeed the case. The main importance of LF mapping lies in the fact that it allows us to simulate the suffix array, which is crucial for text-indexing data structures. In fact, once we can simulate the o-suffix array, we can find the suffix range of the encoded pattern, essentially by employing a binary-search based approach (see Theorem 6.4). Therefore, the holy grail is to implement LF mapping. For order-preserving text indexing, we introduce a new way of implementing LF mapping, which we call *LF Successor*: if the LF successor of a leaf i in the o-suffix tree

is the leaf j, then $\mathsf{LF}(j) = \mathsf{LF}(i) + 1$. A sampling technique, similar to the one used for LF mapping to suffix array transition in Theorem 4.3, is employed to make transition from LF successor to LF mapping. Therefore, we have reduced the problem to that of computing LF successor. See Theorem 6.3 for details on how to simulate o-suffix array from LF successor. Theorem 6.1 is now a simple corollary. Section 6.3 contains the details.

At this point, one might wonder why we need an alternate way of implementing LF mapping, and why will a pBWT-like approach not work. We provide some insight into this. Recall that in the encoding scheme of Baker [Bak93], when we prepend a character to a suffix, the encoding of at most one position in the suffix changes. Unfortunately, for the o-matching problem, such an encoding scheme does not exist, and in fact, does not even seem likely. In retrospect, the encoding scheme for p-matching problem is possible because of the following phenomenon. Suppose, two strings S and S' are a p-match and we want to check whether cS and $c'S'$ are a p-match for some p-character c. Then, all we have to do is make sure that the first occurrence (if any) of c in S is the same as that of c' in S'. Hence, an encoding scheme which changes only one position on prepending a character seems possible, and indeed such an encoding scheme exists for p-matching. For o-matching, when we prepend the characters c and c' to the already o-matched strings S and S', we have to ensure that for any position i in S, $S[i]$ has the same order w.r.t c as $S'[i]$ has w.r.t to c'. Therefore, existence of a scheme that changes only one position on prepending seems unlikely.

Luckily, for o-matching, an encoding scheme is conceivable, where the change on prepending only occurs on the first occurrence of a distinct character. We provide the insight behind this. Suppose, we are prepending c to S and c' to S'. Consider two positions i and j, such that $i < j$ and $S[i] = S[j]$. Due to the order-preserving property, we have $S'[i] = S'[j]$. Clearly, to verify if cS is an o-match with $c'S'$, it suffices to verify that the relative order between c and the first occurrence of each distinct character in S is same as that between c' and the first occurrence of the corresponding distinct character

in S'. Indeed the encoding scheme of Crochemore et al. [CIK+13] guarantees that changes only occur on the first occurrence of a character. This crucial insight, along with other important definitions are presented in Section 6.4.

To implement LF successor, we first make an important observation. Assume j is the LF successor of i, i.e., $\mathsf{LF}(j) = \mathsf{LF}(i) + 1$. Then, $\mathsf{LF}(j)$ is the suffix that shares the longest prefix with $\mathsf{LF}(i)$ and lies to the right of $\mathsf{LF}(i)$. In other words, we can say that j is the suffix that agrees the most with i when prepended by their respective previous characters in the text, i.e., the suffixes i and j have similar points of change when prepended. Thus in some sense, it becomes important to identify the first point of disagreement between i and j, i.e., the first position where one suffix changes and the other one does not. We divide the LF successor pairs i and j into two categories based on whether the first point of disagreement between them is before or after their longest common prefix length. These two cases are implemented in Sections 6.5 and 6.6. Lemmas 6.3, 6.11, and 6.12 are the most important takeaways from these sections.

At this point, we have gathered enough information to identify j, given i and a suitable range of leaves in the o-suffix tree. The question is how to identify j efficiently. We store enough information, still only $O(\log \sigma)$ bits per suffix, so that locating j becomes rather an arguably straight-forward exercise in (succinct) data structures. However, not all the data structures that we use come in a box. Hence, we design a few in Section 6.7. Finally, in Section 6.8, we combine these data structures, along with Lemmas 6.3, 6.11, 6.12, to obtain a compact index for implementing LF successor.

6.2 Order-preserving Indexing

6.2.1 Encoding Scheme

We convert a string S to a string $\mathsf{order}(S)$ as follows. Let $i \in [1, |S|]$ and p_- (resp. p_+) be the highest value (resp. lowest value) in $S[1, i-1]$ that is at most $S[i]$ (resp. at least $S[i]$). Let j_- (resp. j_+) be the rightmost occurrence of p_- (resp. p_+) in $[1, i)$. If p_- (resp. p_+) does not exist, then assign $j_- = i$ (resp. $j_+ = i$). Assign, $\mathsf{order}(S)[i] = \langle i - j_-, i - j_+ \rangle$.

Thus, $\mathsf{order}(43665) = \langle 0,0\rangle\langle 0,1\rangle\langle 2,0\rangle\langle 1,1\rangle\langle 4,1\rangle$. Although we can compute $\mathsf{order}(S)$ in $O(|S|\log\sigma)$ time using a balanced binary search tree, we employ the more general method of Lemma 6.1, which shall be required later.

Lemma 6.1. *Given a string S over Σ, let $S' = S[j,k]$ be an arbitrary substring of S. We can compute $\mathsf{order}(S')[i]$, $1 \le i \le |S'|$, in $O(\log\sigma)$ time after an initial pre-processing of S in $O(|S|\log\sigma)$ time.*

Proof. We first construct the wavelet tree of Fact 2.3 over S. Using the algorithm of Grossi et al. [GGV03], this can be achieved in $O(|S|\log\sigma)$ time. Then, $\mathsf{order}(S')[i] = \langle i - \mathsf{prevVal}_S(j, j+i-1), i - \mathsf{nextVal}_S(j, j+i-1)\rangle$ is computed in $O(\log\sigma)$ time. ■

Two pairs $\langle x_i, y_i\rangle$ and $\langle x_j, y_j\rangle$ in this encoding scheme are the same iff $x_i = x_j$ and $y_i = y_j$. Lexicographic order is defined first by comparing x_i and x_j and then (if $x_i = x_j$) by comparing y_i and y_j. As a convention, we use the lexicographic order: $0 \prec 1 \prec \cdots \prec \$$.

Fact 6.1 ([CIK+13])**.** *Two strings X and Y are order-preserving iff $\mathsf{order}(X) = \mathsf{order}(Y)$. Also, X is order-preserving with a prefix of Y iff $\mathsf{order}(X)$ is a prefix of $\mathsf{order}(Y)$.*

6.2.2 Order-preserving Suffix Tree

Let $\mathcal{P} = \{\mathsf{order}(T[i,n])\langle \$, \$\rangle \mid 1 \le i \le n\}$, where $\$ \notin \Sigma$. The *order-preserving suffix tree* oST of T is the compacted trie of the strings in $\mathcal{P}$. The suffix tree occupies $\Theta(n\log n)$ bits, has one leaf per suffix, and each edge (u,v) is labeled by a substring of $\mathsf{order}(T[j,n])\langle \$, \$\rangle$, where $T[j,n]$ is a suffix corresponding to a leaf in the subtree of v. We assume that leaves are arranged from left to right in the lexicographic order of the corresponding order-encoded suffix. We use ℓ_i to denote the ith leftmost leaf. To find all *occ* occurrences of P, first compute $\mathsf{order}(P)$ in $O(|P|\log\sigma)$ time using Lemma 6.1. Now, trace the path to the highest node u (called *locus*) such that $\mathsf{order}(P)$ is a prefix of $\mathsf{path}(u)$, the concatenation of edge labels from the root to u. Each occurrence is given by $\mathsf{oSA}[i]$, $sp \le i \le ep$, where $[sp, ep]$ is the suffix range of P, i.e., the leaves under u. Here, oSA is the order-preserving suffix

array, i.e., $\mathsf{oSA}[i] = j$ and $i = \mathsf{oSA}^{-1}[j]$ iff i is the lexicographic rank of $\mathsf{order}(T[j, n])\langle\$, \$\rangle$ in $\mathcal{P}$. The time needed to report all the *occ* occurrences is $O(|P| \log \sigma + occ)$.

6.3 LF Successor

As discussed in the previous section, for supporting pattern matching, we need to **(a)** compute the suffix range of $\mathsf{order}(P)$ and **(b)** decode suffix array values. By using the wavelet tree over T (see Fact 2.3) and an additional $O(n)$-bit data structure, we can compute the suffix range efficiently, provided that we can decode suffix array values. Therefore, the main task is to simulate the suffix array. To this end, we store a *sampled suffix array*, i.e., we store $\mathsf{oSA}[i]$ if $\mathsf{oSA}[i] \in \{1, 1 + \Delta, 1 + 2\Delta, \ldots, n\}$. Now, the original suffix array can be simulated via order-preserving LF mapping. For any two leaves ℓ_i and ℓ_j, LF mapping is defined as:

$$j = \mathsf{oLF}(i) \text{ and } i = \mathsf{oLF}^{-1}(j) \text{ iff } j = \mathsf{oSA}^{-1}[\mathsf{oSA}[i] - 1], \text{ where } \mathsf{oSA}^{-1}[0] = \mathsf{oSA}^{-1}[n]$$

To find $\mathsf{oSA}[i]$, we repeatedly apply LF mapping starting from i until we reach j such that $\mathsf{oSA}[j]$ has been sampled; then, $\mathsf{oSA}[i] = \mathsf{oSA}[j] + k$, where k is the number of oLF operations applied. By choosing $\Delta = \log_\sigma n$, we can bound k by Δ and the sampled suffix array size by $O(n \log \sigma)$ bits. Thus, we have reduced the problem to that of computing $\mathsf{oLF}(i)$. We present a new way of computing $\mathsf{oLF}(i)$, called *LF successor*, defined as:

$$j = \mathsf{oLF_s}(i), \text{ i.e., } j \text{ is the order-preserving LF successor of } i \text{ iff}$$
$$j = \mathsf{oLF}^{-1}(\mathsf{oLF}(i) + 1), \text{ where } \mathsf{oLF}(i) < n$$

To compute LF mapping from LF successor, we again use a sampling technique as above. Thus, the problem has been reduced to computing LF successor. Our approach to compute LF successor is via Theorem 6.2; note that $\mathsf{lexRank}(i)$ closely resembles $\mathsf{oLF_s}(i)$ when $\mathcal{X}$ is the set of suffixes of T and x_i is the previous character of each suffix.

Theorem 6.2. *Let $\mathcal{X}$ be a collection of κ strings $\{X_1, X_2, \ldots, X_\kappa\}$ over Σ. Each X_i is associated with a character $x_i \in \Sigma$. We assume that $\mathcal{X}$ is prefix free, i.e., X_i is not an o-match with a prefix of any other X_j.*
Define $\mathsf{lexRank}(i)$ *to be the lexicographic rank of* $\mathsf{order}(x_iX_i)$ *in* $\{\mathsf{order}(x_kX_k) \mid 1 \leq k \leq \kappa\}$. *Define* $\mathsf{succ}(i) = j$ *iff* $\mathsf{lexRank}(j) = \mathsf{lexRank}(i) + 1$, *where* $\mathsf{lexRank}(i) < \kappa$. *By using an $O(\kappa \log \sigma)$-bit data structure, we can compute* $\mathsf{succ}(i)$ *in $O(\log \sigma)$ time.*

We defer the proof of the above theorem to Sections 6.4 through 6.8. Instead, we now focus on how to compute LF successor using Theorem 6.2. Following this, we decode suffix array values using the techniques discussed previously; Theorem 6.3 presents the details.

Theorem 6.3. *By using an $O(n \log \sigma)$-bit data structure, we can compute*

(a) $\mathsf{oLF_s}(i)$ *in $O(\log \sigma)$ time, and*

(b) $\mathsf{oSA}[i]$ *in $O(\log_\sigma n \log n)$ time.*

Proof. **(a)** We use Theorem 6.2 with $\mathcal{X} = \{T[i, n]\$ \mid 1 < i \leq n\}$, where $\$ \notin \Sigma$ is lexicographically larger than any integer. For any string S over Σ, we have $\mathsf{order}(S\$) = \mathsf{order}(S)\langle \$, \$ \rangle$. Note that $\mathcal{X}$ is prefix-free and does not contain the string $T[1, n]$. Let p be the leaf in the order-preserving suffix tree ST that corresponds to $T[1, n]$, i.e., $p = \mathsf{oSA}^{-1}[1]$. We number the strings in $\mathcal{X}$ according to the lexicographic order of the corresponding order-encoded suffix of T. Thus, the ith string in $\mathcal{X}$ corresponds to the jth leaf in oST as follows: $i = j$ if $j < p$ and $i = j - 1$ if $j > p$. Let $\gamma(i) = j$ and $i = \gamma^{-1}(j)$ if the ith string in $\mathcal{X}$ corresponds to the jth leaf in oST. Clearly, both γ and γ^{-1} operations can be implemented in $O(1)$ time. For every $i \in [1, n-1]$, we associate the character $x_i = T[\mathsf{oSA}[\gamma(i)] - 1]$ with the string $X_i = T[\mathsf{oSA}[\gamma(j)] - 1, n]\$ \in \mathcal{X}$. For the leaf $\ell_p \in \mathsf{oST}$, we explicitly store $\mathsf{oLF_s}(p)$. Also, we explicitly store q with $\mathsf{oLF_s}(q) = p$. The space needed is $O(\log n)$ bits. Therefore, given p or q, we can return $\mathsf{oLF_s}(p)$ or $\mathsf{oLF_s}(q)$ in $O(1)$ time. For any other i, we compute $\mathsf{oLF_s}(i) = \gamma^{-1}(\mathsf{succ}(\gamma(i)))$ via Theorem 6.2 in $O(\log \sigma)$ time. The total space usage is $O(n \log \sigma + \log n) = O(n \log \sigma)$ bits.

(b) We employ perfect hashing [FKS84] to store the $\langle j, \mathsf{oLF}(j)\rangle$ key-value pairs for all j such that $\mathsf{oLF}(j)$ belongs to $\{1, 1+\Delta, 1+2\Delta, 1+3\Delta, \ldots, n\}$, where $\Delta = \lceil \log_\sigma n \rceil$. Using this, given a j, one can check if $\mathsf{oLF}(j)$ has been stored (and also retrieve the value) in $O(1)$ time. The space needed is $O((n/\Delta)\log n)$ bits. Then $\mathsf{oLF}(i)$ is computed in $O(1)$ time if its value has been explicitly stored. Otherwise, we apply successive $\mathsf{oLF_s}$ operations starting from i until we reach an index j such that $\mathsf{oLF}(j)$ has been explicitly stored. Let the number of $\mathsf{oLF_s}$ operations be k, and let the indexes yielded be $i_1, i_2, \ldots, i_k = j$ in that order. We have the following telescoping series: $\mathsf{oLF}(i_1) = \mathsf{oLF}(i) + 1, \mathsf{oLF}(i_2) = \mathsf{oLF}(i_1) + 1, \ldots, \mathsf{oLF}(i_k) = \mathsf{oLF}(i_{k-1}) + 1$. Then, $\mathsf{oLF}(i_k) = \mathsf{oLF}(j) = \mathsf{oLF}(i) + k$. Clearly, $k \leq \Delta$ and the time required is $O(\Delta \cdot \log \sigma)$. To compute $\mathsf{oSA}[i]$, maintain a Δ-sampled suffix array as in the case of LF. If $\mathsf{oSA}[i]$ has not been explicitly stored, then starting from i, apply oLF operation k times until we hit an index j such that $\mathsf{oSA}[j]$ has been explicitly stored. We have $\mathsf{oSA}[j] = \mathsf{oSA}[i] - k$. Since $k \leq \Delta$, the time required is $O(\Delta^2 \cdot \log \sigma) = O(\log_\sigma n \log n)$. The total space occupied by the data structure is $O((n/\Delta)\log n) = O(n \log \sigma)$ bits. ■

At the beginning of this section, we claimed that if we can compute suffix array values, then we can find the suffix range of $\mathsf{order}(P)$. The technique is rather standard and the details are presented in Theorem 6.4.

Theorem 6.4. *Assume* $\mathsf{oSA}[i]$ *is computed in time* t_{oSA} *and for any substring* S *of* T *or* P, $\mathsf{order}(S)[i]$ *in time* t_{order}. *By using an additional* $O(n)$*-bit data structure, we can find all occ occurrences of* P *in* T *in time* $O(|P| t_{\mathsf{order}} \log\log n + (\log\log n + occ) \cdot t_{\mathsf{oSA}})$.

Proof. Given a range $R = [x, y]$ of leaves in oST, we can compute the maximal subrange of R such that for each ℓ_i in this subrange, $\mathsf{order}(P)$ is a prefix of $\mathsf{order}(T[\mathsf{oSA}[i], n])$. This is achieved via a binary search on R using oSA (see e.g., [Gus97] for more details on how this binary search is performed). The time needed is $O((|P| t_{\mathsf{order}} + t_{\mathsf{oSA}}) \cdot \log |R|)$. Thus, if the range is $R = [1, n]$, we can find the suffix range of P in time $O(|P| t_{\mathsf{order}} \log n + t_{\mathsf{oSA}} \cdot \log n)$. We show that by maintaining slightly more additional information, we can improve this. We

maintain a compacted trie oST^{Δ} of the strings $\mathsf{order}(T[\mathsf{oSA}[i], n])$, where $i \in \{1, 1+\Delta, 1+2\Delta, \ldots, n\}$ and $\Delta = \lceil \log n \rceil$. Given P, we match $\mathsf{order}(P)$ in oST^{Δ} until we reach a position on an edge (u, v), where either P is completely matched or a mismatch is found. Searching in oST^{Δ} can be performed in $O(|P| t_{\mathsf{order}})$ time if the internal nodes in oST^{Δ} are equipped with perfect hashing [FKS84] for selecting the correct outgoing edge (if any) in $O(1)$ time. Then, the desired suffix range in oST corresponds to the leaves under v (if P was completely matched), and at most 2Δ leaves on either side of the leaves under v. Clearly, the desired suffix range $[sp, ep]$ of $\mathsf{order}(P)$ is found in time $O(|P| t_{\mathsf{order}} + (|P| t_{\mathsf{order}} + t_{\mathsf{oSA}}) \cdot \log \Delta)$. Finally, each occurrence is given by $\mathsf{oSA}[i]$, $sp \leq i \leq ep$, and is reported in $O(t_{\mathsf{oSA}})$ time. The total space occupied is $O((n/\Delta) \log n) = O(n)$ bits. ■

6.3.1 Proof of Main Result

Now, we have the ingredients to prove Theorem 6.1.

The data structure comprises of a wavelet tree over T; see Fact 2.3. The space required is $n \log \sigma + o(n)$ bits. Applying Theorem 6.3, $\mathsf{oSA}[i]$ is computed in $t_{\mathsf{oSA}} = O(\log_\sigma n \log n)$ time via an $O(n \log \sigma)$-bit data structure. By invoking Lemma 6.1, for any substring S of T or P, $\mathsf{order}(S)[i]$ is computed in time $t_{\mathsf{order}} = O(\log \sigma)$. Thus, using Theorem 6.4, we can find the suffix range of $\mathsf{order}(P)$ in $O((|P| \log \sigma + \log_\sigma n \log n) \cdot \log \log n)$ time. Theorem 6.1 follows immediately.

The remaining challenge in this chapter is to prove Theorem 6.2. The proof is technical and requires numerous supporting lemmas and data structures. We split the proof over Sections 6.4 – 6.8.

6.4 Some Useful Definitions

Let $\mathcal{T}_{\mathcal{X}}$ be the compacted trie of the strings $\{\mathsf{order}(X_i) \mid 1 \leq i \leq \kappa\}$. Note that $\mathcal{T}_{\mathcal{X}}$ has κ leaves (since $\mathcal{X}$ is prefix free, each string in $\mathcal{X}$ corresponds to a unique leaf in $\mathcal{T}_{\mathcal{X}}$). Let the ith leftmost leaf correspond to X_i.

Definition 6.2. *Define $\alpha(S)$ to be the number of distinct symbols in a string S. Let ℓ_i denote the ith leftmost leaf in $\mathcal{T}_{\mathcal{X}}$, which is associated with the string X_i and the character x_i; we use "leaf i" to denote ℓ_i. Define $\mathsf{path}(u)$ to be the string formed by concatenating the edge-labels from the root to u, $\mathsf{strDepth}(u) = |\mathsf{path}(u)|$ and $\mathsf{zeroDepth}(u) = \alpha(X_i[1, \mathsf{strDepth}(u)])$, where ℓ_i lies in the subtree of u. Let $\mathsf{lcp}(i, j)$ be the length of the longest common prefix (LCP) of $\mathsf{order}(X_i)$ and $\mathsf{order}(X_j)$, i.e., $\mathsf{lcp}(i, j) = \mathsf{strDepth}(\mathsf{lca}(\ell_i, \ell_j))$.*

Definition 6.3. *d is a **point of change** of i iff $\mathsf{order}(X_i)[d] \neq \mathsf{order}(x_i X_i)[d+1]$. d is a **point of disagreement** of i and j iff $\mathsf{order}(x_i X_i)[d+1] \neq \mathsf{order}(x_j X_j)[d+1]$. Let $\delta(i, j)$ denote the **first point of disagreement** of i and j; then, we say i and j **agree upto** the position $\delta(i, j) - 1$.*

We present some important properties, which can be easily derived from the definition of order and the definitions above.

Observation 6.1. *Given a string S, let $|S| \geq q > p \geq 1$, $\langle p_-, p_+ \rangle = \mathsf{order}(S)[p]$, and $\langle q_-, q_+ \rangle = \mathsf{order}(S)[q]$. Define line segments $L_-(S, p) = (p, S[p]) \to (p'_-, S[p'_-])$ and $L_+(S, p) = (p, S[p]) \to (p'_+, S[p'_+])$, where $p'_- = i - p_-$ and $p'_+ = i - p_+$. Likewise, define $L_-(S, q)$ and $L_+(S, q)$. Let $L_{S,q}$ be the projection of $L_-(S, q)$ or $L_+(S, q)$ onto the y-axis. (The projection of a line from $(x_1, y_1) \to (x_2, y_2)$ onto the y-axis is the line $(0, y_1) \to (0, y_2)$.) Likewise, define $L_{S,p}$. Then, $L_{S,q}$ is either contained in $L_{S,p}$, or they are disjoint except (possibly) at the end points.*

Observation 6.2. *Let S and S' be two strings such that $\mathsf{order}(S) = \mathsf{order}(S')$. Consider the line segments $L_{S,q}$, $L_{S',q}$, $L_{S,p}$, and $L_{S',p}$, as defined in Observation 6.1, where $|S| \geq q > p \geq 1$. Then, the line segment $L_{S,q}$ is contained in $L_{S,p}$ iff $L_{S',q}$ is contained in $L_{S',p}$.*

Observation 6.3. *Let d be a point of change of i, and $\langle d_-, d_+ \rangle = \mathsf{order}(X_i)[d]$. Then the horizontal line through x_i intersects at least one of the lines $L_-(X_i, d)$ or $L_+(X_i, d)$. Also, x_i and $X_i[d]$ do not appear in $X_i[1, d-1]$; hence, $\alpha(X_i[1, d]) = \alpha(X_i[1, d-1]) + 1$.*

The main idea is to classify the LF-successor pairs $\langle i, j = \mathsf{succ}(i)\rangle$ based on their first point of disagreement $\delta(i,j)$ and $\mathsf{lcp}(i,j)$. The first case, when $\delta(i,j) \leq \mathsf{lcp}(i,j)$, follows directly from Observations 6.1 and 6.3; the details are in Section 6.5. The other case, when $\delta(i,j) > \mathsf{lcp}(i,j)$, needs much more machinery; the details are in Section 6.6. At this point, we have gathered enough information for identifying j. Now, we transform this into a data structure problem. By using existing data structures (Fact 2.3 and 2.4), along with standard encoding tricks, we prove Theorem 6.2; the details are in Sections 6.7 and 6.8.

6.5 Successor Pair First Disagree Before their LCP

Let $j = \mathsf{succ}(i)$, where $\delta(i,j) \leq \mathsf{lcp}(i,j)$. We note that $\delta(i,j)$ is a point of change of j (follows from lexicographic convention $0 \prec 1 \prec \cdots \prec \$$). Assume $\delta(i,j)$ lies on the edge from u to its parent. Using Lemmas 6.2 and 6.3, we show that there cannot be a leaf ℓ_l, such that $l \neq i$, $l \in [\mathsf{lmostLeaf}(u), \mathsf{rmostLeaf}(u)]$ and $\delta(i,j) = \delta(l, \mathsf{succ}(l))$. Thus, j is a leaf in the subtree of u, which first disagrees with i at $\delta(i,j)$. By storing $\alpha(X_i[1, \delta(i,j)])$ at both i and j, we can identify j.

Definition 6.4. *Let $j = \mathsf{succ}(i)$. Define two arrays $\alpha\mathsf{Disagree}$ and $\overline{\alpha\mathsf{Disagree}}$ as follows. If $\delta(i,j) \leq \mathsf{lcp}(i,j)$, $\alpha\mathsf{Disagree}[i] = \overline{\alpha\mathsf{Disagree}}[j] = \alpha(X_i[1, \delta(i,j)])$. Otherwise, $\alpha\mathsf{Disagree}[i] = \overline{\alpha\mathsf{Disagree}}[j] = 0$.*

Lemma 6.2. *If $q \leq \mathsf{lcp}(s,t)$ is a point of change of s and t, then $\delta(s,t) \geq q$.*

Proof. Since $\mathsf{lcp}(s,t) \geq q$, $\mathsf{order}(X_s[1,q]) = \mathsf{order}(X_t[1,q])$. Consider a position $p < q$. Let $\langle p_-, p_+\rangle = \mathsf{order}(X_s)[p] = \mathsf{order}(X_t)[p]$, $\langle p^s_-, p^s_+\rangle = \mathsf{order}(x_s X_s)[p+1]$ and $\langle p^t_-, p^t_+\rangle = \mathsf{order}(x_t X_t)[p+1]$. Let $L_{X_s,q}$ be the projection of the line $L_+(X_s, q)$ or $L_-(X_s, q)$ onto the y-axis (see Observation 6.1). Likewise, define $L_{X_t,q}$, $L_{X_s,p}$, and $L_{X_t,p}$. By Observation 6.2, $L_{X_s,q}$ is contained in $L_{X_s,p}$ iff $L_{X_t,q}$ is contained in $L_{X_s,p}$. Since q is a point of change for both s and t, applying Observation 6.3, we get $p^s_- \neq p_-$ iff $p^t_- \neq p_-$ and $p^s_+ \neq p_+$ iff $p^t_+ \neq p_+$. Hence, the lemma follows. ■

Lemma 6.3. *Let* $\alpha\mathsf{Disagree}[i] > 0$ *and* $j = \mathsf{succ}(i)$. *Let* u *be the highest node on the root to* ℓ_i *path such that* $\mathsf{zeroDepth}(u) \geq \alpha\mathsf{Disagree}[i]$. *There is exactly one* $k \in [\mathsf{lmostLeaf}(u), \mathsf{rmostLeaf}(u)]$ *such that* $\overline{\alpha\mathsf{Disagree}}[k] = \alpha\mathsf{Disagree}[i]$. *Moreover,* $j = k$.

Proof. Since $j \in [\mathsf{lmostLeaf}(u), \mathsf{rmostLeaf}(u)]$, it suffices to show there is exactly one $k \in [\mathsf{lmostLeaf}(u), \mathsf{rmostLeaf}(u)]$, where $\overline{\alpha\mathsf{Disagree}}[k] = \alpha\mathsf{Disagree}[i]$. Assume there exists $k, k' \in [\mathsf{lmostLeaf}(u), \mathsf{rmostLeaf}(u)]$, where $\alpha\mathsf{Disagree}[i] = \overline{\alpha\mathsf{Disagree}}[k] = \overline{\alpha\mathsf{Disagree}}[k']$. Then there exists $l, l' \in [\mathsf{lmostLeaf}(u), \mathsf{rmostLeaf}(u)]$ satisfying $k = \mathsf{succ}(l)$, $k' = \mathsf{succ}(l')$, and $\alpha\mathsf{Disagree}[l] = \alpha\mathsf{Disagree}[l'] = \overline{\alpha\mathsf{Disagree}}[k]$. Applying Observation 6.3, $\delta(k, l) = \delta(k', l')$; otherwise $\alpha(X_k[1, \delta(k, l)]) \neq \alpha(X_{k'}[1, \delta(k', l')])$, a contradiction. Since $\mathsf{lexRank}(k) > \mathsf{lexRank}(l)$ and $\mathsf{lexRank}(k') > \mathsf{lexRank}(l')$, $d = \delta(k, l)$ is a point of change for k and k'. Applying Lemma 6.2, $\delta(k, k') \geq d$. Consequently, l, l', k, and k' agree upto $d - 1$ characters. Hence, the strings $\mathsf{order}(x_k X_k)$ and $\mathsf{order}(x_{k'} X_{k'})$ are both lexicographically larger than the strings $\mathsf{order}(x_l X_l)$ and $\mathsf{order}(x_{l'} X_{l'})$, i.e., $\min\{\mathsf{lexRank}(k), \mathsf{lexRank}(k')\} > \max\{\mathsf{lexRank}(l), \mathsf{lexRank}(l')\}$. Consequently, $k = \mathsf{succ}(l)$ and $k' = \mathsf{succ}(l')$ do not simultaneously hold, a contradiction. ■

6.6 Successor Pair First Disagree After their LCP

Let $j = \mathsf{succ}(i)$, where $\delta(i, j) > \mathsf{lcp}(i, j)$. We consider two cases: $i < j$ and $i > j$. Under the respective cases, given i, we will first find a range of leaves such that j is the leftmost (or the only leaf) in this range that agrees with i upto their LCP. To identify j, we will record $\mathsf{zeroDepth}(\mathsf{lca}(i, j))$ with both i and j; this is handled in Sections 6.6.1 and 6.6.2 for the two cases respectively. The only remaining question is to filter out the leaves in this range that disagree with i before their LCP; Section 6.6.3 handles this.

Definition 6.5. *Let* $j = \mathsf{succ}(i)$. *If* $j > i$ *then let* $c = \mathsf{zeroDepth}(\mathsf{lca}(i, j))$, *else* $c = -\mathsf{zeroDepth}(\mathsf{lca}(i, j))$. *Define two arrays* $\alpha\mathsf{LCP}$ *and* $\overline{\alpha\mathsf{LCP}}$ *as follows. If* $\delta(i, j) > \mathsf{lcp}(i, j)$, $\alpha\mathsf{LCP}[i] = \overline{\alpha\mathsf{LCP}}[j] = c$, *else* $\alpha\mathsf{LCP}[i] = \overline{\alpha\mathsf{LCP}}[i] = 0$.

Lemma 6.4. *Consider two leaves* ℓ_s *and* ℓ_t, *and let* $d = \mathsf{lcp}(s, t)$. *Furthermore, let* $\langle p_-, p_+ \rangle = \mathsf{order}(X_s)[d + 1]$ *and* $\langle q_-, q_+ \rangle = \mathsf{order}(X_t)[d + 1]$. *Then,* $p_- \neq q_-$ *and* $p_+ \neq q_+$.

Proof. Since $d = \mathsf{lcp}(s,t)$, $p_- = q_-$ and $p_+ = q_+$ do not simultaneously hold. Let $p_- = q_-$ but $p_+ \neq q_+$. A simple analysis shows that this contradicts the definition of $\mathsf{order}(X_s)[d+1]$ or $\mathsf{order}(X_t)[d+1]$. The case when $p_+ = q_+$ but $p_- \neq q_-$ does not hold by symmetry. ■

Lemma 6.5. *Consider two leaves ℓ_s and ℓ_t, and let $d = \mathsf{lcp}(s,t)$. Furthermore, let $\langle p_-, p_+ \rangle = \mathsf{order}(X_s)[d+1]$, $\langle q_-, q_+ \rangle = \mathsf{order}(X_t)[d+1]$, $\langle p'_-, p'_+ \rangle = \mathsf{order}(x_s X_s)[d+2]$, and $\langle q'_-, q'_+ \rangle = \mathsf{order}(x_t X_t)[d+2]$. If $p_- \neq p'_-$ and $q_- \neq q'_-$ (or, $p_+ \neq p'_+$ and $q_+ \neq q'_+$), then $\delta(s,t) \leq d$.*

Proof. We invoke Lemma 6.4 to establish $p_- \neq q_-$ and $p_+ \neq q_+$. Now, consider the lines $L_-(X_s, d+1)$ and $L_-(X_t, d+1)$ in Observation 6.1. Let L_s/L_t be the horizontal lines through x_s/x_t. It can be shown via a simple (but tedious) case-by-case analysis that if L_s and L_t intersect $L_-(X_s, d+1)$ and $L_-(X_t, d+1)$ respectively, then there is a position $d' \leq d$ where L_s will intersect $L_+(X_s, d')$ (resp. $L_-(X_s, d')$) but L_t will not intersect $L_+(X_t, d')$ (resp. $L_-(X_t, d')$); now, we apply the first part of Observation 6.3 to establish our claim. Similar remarks hold if L_s and L_t intersect $L_+(X_s, d+1)$ and $L_+(X_t, d+1)$ respectively. ■

6.6.1 Case When $i < \mathsf{succ}(i)$

Let $i < j = \mathsf{succ}(i)$. We record $\alpha_i = \mathsf{zeroDepth}(\mathsf{lca}(i,j))$ for both i and j. Using Lemma 6.6, we find a range of leaves $[L, R]$ which contains the rightmost leaf ℓ_k, $i < k < j$, such that $\overline{\alpha\mathsf{LCP}}[k] = \alpha\mathsf{LCP}[i]$ and $\delta(k,i) > \mathsf{lcp}(i,j)$. Given this range, $j > R$ is the leftmost leaf that has α_i recorded with itself and agrees with i upto their LCP. We find j using Lemma 6.7.

Lemma 6.6. *Let $j = \mathsf{succ}(i)$ and $\alpha\mathsf{LCP}[i] > 0$. Let k be the rightmost leaf in $[i+1, j-1]$ satisfying $\overline{\alpha\mathsf{LCP}}[k] = \alpha\mathsf{LCP}[i]$ and $\delta(i,k) > \mathsf{lcp}(i,j)$. Then,* **(a)** *$\mathsf{lcp}(i,k) > \mathsf{lcp}(i,j)$, and* **(b)** *$1 + \mathsf{lcp}(i,j) < \delta(i,k) \leq \mathsf{lcp}(i,k) + 1$, and $\delta(i,k)$ is a point of change of i.*

Proof. Let $k = \mathsf{succ}(l)$. Since $\overline{\alpha\mathsf{LCP}}[k] > 0$, we have $l < k$ and $\delta(l,k) > \mathsf{lcp}(l,k)$.

(a) We show contradiction in the following cases, which leads to the claim.

- Assume $\mathsf{lcp}(i,k) = \mathsf{lcp}(k,j) = \mathsf{lcp}(i,j)$. Let, $d = \mathsf{lcp}(i,j) + 1$. Since $j = \mathsf{succ}(i)$, either $\mathsf{lexRank}(k) > \mathsf{lexRank}(j)$ or $\mathsf{lexRank}(k) < \mathsf{lexRank}(i)$. If d is not a point of change of

k, either $j = \mathsf{succ}(i)$ is violated or $\delta(i,j) > \mathsf{lcp}(i,j)$ is violated (applying Lemma 6.5). Hence, d is a point of change of k, implying $\mathsf{lexRank}(k) > \mathsf{lexRank}(j)$ and $\alpha(X_k[1,d]) = 1 + \mathsf{zeroDepth}(\mathsf{lca}(i,j))$. If $\mathsf{lcp}(l,k) < \mathsf{lcp}(i,j)$, then i, j, l, and k agree upto $\mathsf{lcp}(l,k)$, implying $k \neq \mathsf{succ}(l)$, a contradiction. If $\mathsf{lcp}(l,k) = \mathsf{lcp}(i,j)$, then either $k \neq \mathsf{succ}(l)$ or $\delta(l,k) < \mathsf{lcp}(i,j) = \mathsf{lcp}(l,k)$ applying Lemma 6.5, a contradiction. If $\mathsf{lcp}(l,k) > \mathsf{lcp}(i,j)$, then $\mathsf{zeroDepth}(\mathsf{lca}(k,l)) = \overline{\alpha\mathsf{LCP}}[k] \geq \alpha(X_k[1,d]) > \mathsf{zeroDepth}(\mathsf{lca}(i,j)) = \alpha\mathsf{LCP}[i]$, a contradiction. Therefore, our assumption is false.

- Assume $\mathsf{lcp}(k,j) > \mathsf{lcp}(i,j)$. Using arguments as in the previous case, we first conclude that $\delta(k,j) \leq \mathsf{lcp}(k,j) + 1$ is a point of change of k. Then we use this conclusion to contradict at least one of the following: $j = \mathsf{succ}(i)$, or $\delta(i,j) < \mathsf{lcp}(i,j)$, or $k = \mathsf{succ}(l)$, or $\delta(l,k) < \mathsf{lcp}(l,k)$, or $\overline{\alpha\mathsf{LCP}}[k] = \alpha\mathsf{LCP}[i]$.

(b) Note that $\mathsf{lcp}(i,j)+1$ is not a point of change of k or i; otherwise, applying arguments as above, at least one of the following is violated: $k = \mathsf{succ}(l)$, or $\overline{\alpha\mathsf{LCP}}[k] = \alpha\mathsf{LCP}[i]$, or $\delta(l,k) < \mathsf{lcp}(l,k)$. Since $j = \mathsf{succ}(i)$, the claim follows. ■

The following lemma is our main result in this subsection.

Lemma 6.7. *Let $\alpha\mathsf{LCP}[i] > 0$ and $j = \mathsf{succ}(i)$. Let k be the rightmost leaf in $[i+1, j-1]$ such that $\overline{\alpha\mathsf{LCP}}[k] = \alpha\mathsf{LCP}[i]$ and $\delta(i,k) > \mathsf{lcp}(i,j)$. If k does not exist, then assign $u = \ell_i$. Otherwise, let $d = \delta(i,k)$ lie on the edge (v,w), $v = \mathsf{parent}(w)$. If $d = \mathsf{strDepth}(v)+1$, then assign $u = v$, else assign $u = w$. Let k' be the leftmost leaf in $[\mathsf{rmostLeaf}(u)+1, n]$, where $\overline{\alpha\mathsf{LCP}}[k'] = \alpha\mathsf{LCP}[i]$ and $\delta(i,k') > \mathsf{lcp}(i,k')$. Then, we claim that $j = k'$.*

Proof. Assume k exists. Applying Lemma 6.6, d is a point of change of i. If $d = \mathsf{strDepth}(v)+1$, then $j \notin [\mathsf{lmostLeaf}(v), \mathsf{rmostLeaf}(v)]$; otherwise, $d = \mathsf{lcp}(i,j)+1$, a contradiction (applying Lemma 6.6). If $d > \mathsf{strDepth}(v)+1$, then j does not lie in the subtree of w as $\delta(i,k) \leq \mathsf{lcp}(i,j)$, a contradiction. Hence, j must be leaf to the right of the subtree of u. Assume $j \neq k'$. Then $j = k''$ for some $k'' \in [k'+1, n]$ that satisfies $\overline{\alpha\mathsf{LCP}}[k''] = \alpha\mathsf{LCP}[i]$ and $\delta(i,k'') > \mathsf{lcp}(i,k'')$. Note that $\mathsf{lcp}(i,k') \geq \mathsf{lcp}(i,k'')$. Thus, i, k and k'' all agree upto

$\mathsf{lcp}(i, k'')$, i.e., $\delta(i, k') > \mathsf{lcp}(i, k'') = \mathsf{lcp}(i, j)$. But this violates the definition of k and node u. A similar argument satisfies the claim when k does not exist. ■

6.6.2 Case When $i > \mathsf{succ}(i)$

Let $i > j = \mathsf{succ}(i)$. We record $\alpha_i = -\mathsf{zeroDepth}(\mathsf{lca}(i, j))$ for both i and j. Let u be the child of $\mathsf{lca}(j, i)$ on the path to ℓ_j. Using Lemma 6.8, the first position after $\mathsf{lcp}(i, j)$ must be a point of change of j, and u has only leaf (which is the desired leaf j) that has α_i recored with itself and agrees with i upto their LCP. Lemma 6.9 shows how to find j.

Observation 6.4. *Let $i > j = \mathsf{succ}(i)$ and $d = \mathsf{lcp}(j, i) < \delta(j, i)$. Then, d is a point of change of j. Also, $p_- \neq p'_-$, where $\langle p_-, p_+\rangle = \mathsf{order}(X_j)[d{+}1]$ and $\langle p'_-, p'_+\rangle = \mathsf{order}(x_j X_j)[d{+}2]$. Applying Lemma 6.5, since $\delta(j, i) > \mathsf{lcp}(j, i)$, we have $q_- = q'_-$, where $\langle q_-, q_+\rangle = \mathsf{order}(X_i)[d + 1]$ and $\langle q'_-, q'_+\rangle = \mathsf{order}(x_i X_i)[d + 2]$.*

Lemma 6.8. *Let $j = \mathsf{succ}(i)$ and $\alpha\mathsf{LCP}[i] < 0$. Let u be the child of $\mathsf{lca}(j, i)$ on the path to ℓ_j. Then, $\mathsf{lcp}(j, i) + 1$ is a point of change for j and there is no other $k \in [\mathsf{lmostLeaf}(u), \mathsf{rmostLeaf}(u)]$ that satisfies $\overline{\alpha\mathsf{LCP}}[k] = \alpha\mathsf{LCP}[i]$ and $\delta(k, i) > \mathsf{lcp}(k, i)$.*

Proof. Since $\delta(j, i) > \mathsf{lcp}(j, i)$ and $j = \mathsf{succ}(i)$, it follows that $\delta(j, i) = \mathsf{lcp}(j, i) + 1$ is a point of change of j. For contradiction, assume $k \neq j$ satisfies the above conditions, and let $k = \mathsf{succ}(l)$. Clearly, $\mathsf{lcp}(k, i) = \mathsf{lcp}(j, i)$; hence, i, j and k agree upto $\mathsf{lcp}(i, j)$. Note that $\mathsf{lcp}(k, l) \leq \mathsf{lcp}(k, i) = \mathsf{lcp}(j, i)$; otherwise, $\overline{\alpha\mathsf{LCP}}[k] = -\mathsf{zeroDepth}(\mathsf{lca}(k, l)) \leq -\alpha(X_j[1, 1 + \mathsf{lcp}(j, i)]) = -(1 + \mathsf{zeroDepth}(\mathsf{lca}(j, i))) < \alpha\mathsf{LCP}[i]$, a contradiction (refer to Observation 6.3). Since $\overline{\alpha\mathsf{LCP}}[k] < 0$, $l > k$ and $\mathsf{lcp}(k, l) + 1$ is a point of change of k. If $\mathsf{lcp}(k, l) < \mathsf{lcp}(j, i)$, then $\alpha\mathsf{LCP}[i] = -\mathsf{zeroDepth}(\mathsf{lca}(j, i)) \leq -(1 + \mathsf{zeroDepth}(\mathsf{lca}(k, l))) < \overline{\alpha\mathsf{LCP}}[k]$, a contradiction. If $\mathsf{lcp}(k, l) = \mathsf{lcp}(j, i)$, then applying Observation 6.4, at least one of $k = \mathsf{succ}(l)$ or $j = \mathsf{succ}(i)$ is violated. ■

Now, we proceed to the prove our main result in this subsection, summarized in the following lemma.

Lemma 6.9. *Let* $\alpha\mathsf{LCP}[i] < 0$ *and* $j = \mathsf{succ}(i)$. *Let* v *be the lowest ancestor of* ℓ_i *that has a child* u *such that* $\mathsf{pre\text{-}order}(u) < \mathsf{pre\text{-}order}(\ell_i)$, u *is not an ancestor of* ℓ_i, *and there is a* $k \in [\mathsf{lmostLeaf}(u), \mathsf{rmostLeaf}(u)]$ *with* **(a)** $\overline{\alpha\mathsf{LCP}}[k] = \alpha\mathsf{LCP}[i]$, **(b)** $\delta(k, i) > \mathsf{lcp}(k, i)$, *and* **(c)** $\mathsf{strDepth}(v) + 1$ *is a point of change of* k. *Then,* u *and* k *are both unique, and* $j = k$.

Proof. We first assume that u is unique, and we show that k is unique and $j = k$. In the second part of the proof, we show that u is indeed unique.

Applying Lemma 6.8, we conclude that if $j \in [\mathsf{lmostLeaf}(u), \mathsf{rmostLeaf}(u)]$ then k is unique and $j = k$. Therefore, it suffices to show that $j \in [\mathsf{lmostLeaf}(u), \mathsf{rmostLeaf}(u)]$. Consider the set $\mathcal{W}$ of all nodes w such that $\mathsf{parent}(w)$ is an ancestor of ℓ_i but w is not, $\mathsf{pre\text{-}order}(w) < \mathsf{pre\text{-}order}(\ell_i)$, and there exists a $k' \in [\mathsf{lmostLeaf}(w), \mathsf{rmostLeaf}(w)]$ with $\overline{\alpha\mathsf{LCP}}[k'] = \alpha\mathsf{LCP}[i]$, $\delta(k', i) > \mathsf{lcp}(k', i)$ and the leading character on the edge $(\mathsf{parent}(w), w)$ is a point of change of k'. Clearly, $j \in [\mathsf{lmostLeaf}(w'), \mathsf{rmostLeaf}(w')]$ for a node $w' \in \mathcal{W}$ and $u \in \mathcal{W}$. If $u \neq w'$, then $\mathsf{nodeDepth}(w') < \mathsf{nodeDepth}(u)$; otherwise, $v = \mathsf{parent}(u)$ has lower node-depth than $\mathsf{parent}(w')$. Let $k = \mathsf{succ}(l)$. Since $\mathsf{pre\text{-}order}(w') < \mathsf{pre\text{-}order}(u)$, we get $j < k$ and $\mathsf{lcp}(j, i) < \mathsf{lcp}(k, i)$. Now, i and j agree upto $\mathsf{lcp}(j, i)$, and $\mathsf{lcp}(j, i)$ is a point of change of j (applying Observation 6.4). Also, i and k agree upto $\mathsf{lcp}(k, i)$, and $\mathsf{lcp}(k, i)$ is a point of change of k. Applying Observation 6.4, we get $\mathsf{lexRank}(j) > \mathsf{lexRank}(k) > \mathsf{lexRank}(i)$, which contradicts $j = \mathsf{succ}(i)$. Hence, $j \in [\mathsf{lmostLeaf}(w'), \mathsf{rmostLeaf}(w')]$.

Suppose, v has two children u_1 and u_2, neither of which is an ancestor of ℓ_i. For $t \in \{1, 2\}$, assume $\mathsf{pre\text{-}order}(u_t) < \mathsf{pre\text{-}order}(\ell_i)$, there exists a $k_t \in [\mathsf{lmostLeaf}(u_t), \mathsf{rmostLeaf}(u_t)]$ with **(i)** $\overline{\alpha\mathsf{LCP}}[k_t] = \alpha\mathsf{LCP}[i]$, **(ii)** $\delta(k_t, i) > \mathsf{lcp}(k_t, i)$, and **(iii)** the leading character on the edge (v, u_t) is a point of change of k_t. Let $k_t = \mathsf{succ}(l_t)$. First observe that $\mathsf{lcp}(j, i) \leq \mathsf{lcp}(k_t, t)$; otherwise $\mathsf{nodeDepth}(\mathsf{lca}(j, i)) > \mathsf{nodeDepth}(v)$ which contradicts the definition of node v (since j will end up satisfying (a)-(c) in the lemma statement by replacing k with j). Now observe that $\mathsf{lcp}(k_t, l_t) \leq \mathsf{lcp}(k_t, i)$; otherwise, $\overline{\alpha\mathsf{LCP}}[k_t] = -\mathsf{zeroDepth}(\mathsf{lca}(k_t, l_t)) \leq -(1 + \mathsf{zeroDepth}(v)) \leq -(1 + \mathsf{zeroDepth}(\mathsf{lca}(j, i))) < \alpha\mathsf{LCP}[i]$, a contradiction (refer to Observation 6.3). Since k_1 and k_2 agree upto $\mathsf{strDepth}(v)$ and both have a point of change

at $\mathsf{strDepth}(v) + 1$, applying Observation 6.4, we will contradict either $k_1 = \mathsf{succ}(l_1)$ or $k_2 = \mathsf{succ}(l_2)$. Hence, there exists only such node u. ■

6.6.3 Eliminating Rogue Candidate Leaves

At this point, we are equipped with Lemmas 6.7 and 6.9 for finding $j = \mathsf{succ}(i)$ for a given i, where $\alpha\mathsf{LCP}[i] \neq 0$. Specifically, given some appropriate leaf range (determined by the node u in the two lemmas), we have to find the (leftmost) leaf k in the range that satisfies $\overline{\alpha\mathsf{LCP}}[k] = \alpha\mathsf{LCP}[i]$ and $\delta(k, i) > \mathsf{lcp}(k, i)$. The first task is to find the proper range (which we will handle later), and the other is to ignore leaves k' that satisfy $\delta(k', i) \leq \mathsf{lcp}(k', i)$. We now concentrate on the second and introduce the following definition.

Definition 6.6. *For any string $S[1, s]$ and a character c, we let $\chi(S, c) = \langle x, y\rangle$, where x (resp. y) is the number of distinct symbols in S that are smaller than c (resp. greater than c), i.e., $x = |\{S[i] \mid S[i] < c \text{ and } i \in [1, s]\}|$ and $y = |\{S[i] \mid S[i] > c \text{ and } i \in [1, s]\}|$.*

Lemma 6.10. *Consider two strings S and S', where $\mathsf{order}(S) = \mathsf{order}(S')$. Then, for any two characters $c, c' \in \Sigma$, $\mathsf{order}(cS) = \mathsf{order}(c'S')$ iff $\chi(S, c) = \chi(S', c')$.*

Proof. Consider the line segments $L_-(S, p)$ and $L_-(S', p)$ in Observation 6.1. Since $\mathsf{order}(S) = \mathsf{order}(S')$, for any p, $L_-(S, p)$ has the same x-ordinate at the two ends as that of $L_-(S', p)$. Since $\chi(S, c) = \chi(S', c')$, the line $L_-(S, p)$ is intersected by the horizontal line through c iff $L_-(S, p')$ is intersected by the horizontal line through c'. Similar remarks hold for the lines $L_+(S, p)$ and $L_+(S', p)$. Hence, the claim. ■

Consider the node u in Lemmas 6.7 and 6.9. Also, consider a leaf ℓ_k, where $k > \mathsf{rmostLeaf}(u)$ in Lemma 6.7 and $k \in [\mathsf{lmostLeaf}(u), \mathsf{rmostLeaf}(u)]$ in Lemma 6.9. In both the lemmas, let $k \neq j$ and $\overline{\alpha\mathsf{LCP}}[k] = \alpha\mathsf{LCP}[i]$. Then, $\delta(k, i) \leq \mathsf{lcp}(k, i)$ in both the lemmas. Let $k = \mathsf{succ}(l)$. Then, l and k agree upto $\mathsf{lcp}(l, k)$, but they disagree with i and j before $d = \min\{\mathsf{lcp}(i, j), \mathsf{lcp}(l, k)\}$. Applying Lemma 6.10, $\chi(X_l[1, d], x_l) \neq \chi(X_i[1, d], x_i)$. With this intuition,

Definition 6.7. *Define two arrays* Group *and* $\overline{\mathsf{Group}}$ *as follows. For any* i *with* $\alpha\mathsf{LCP}[i] \neq 0$ *and* $j = \mathsf{succ}(i)$, $\mathsf{Group}[i] = \overline{\mathsf{Group}}[j] = \chi(X_i[1, \mathsf{lcp}(i,j)], x_i) = \chi(X_j[1, \mathsf{lcp}(i,j)], x_j)$.

We rewrite Lemmas 6.7 and 6.9 as Lemmas 6.11 and 6.12 respectively.

Lemma 6.11. *Let* $\alpha\mathsf{LCP}[i] > 0$ *and* $j = \mathsf{succ}(i)$. *Let* k *be the rightmost leaf in* $[i+1, j-1]$ *such that* $\overline{\alpha\mathsf{LCP}}[k] = \alpha\mathsf{LCP}[i]$ *and* $\delta(i,k) > \mathsf{lcp}(i,j)$. *If* k *does not exist, then assign* $u = \ell_i$. *Otherwise, let* $d = \delta(i,k)$ *lie on the edge* (v,w), $v = \mathsf{parent}(w)$. *If* $d = \mathsf{strDepth}(v)+1$, *then assign* $u = v$, *else assign* $u = w$. *Let* k' *be the leftmost leaf in* $[\mathsf{rmostLeaf}(u)+1, n]$, *where* $\overline{\alpha\mathsf{LCP}}[k'] = \alpha\mathsf{LCP}[i]$ *and* $\overline{\mathsf{Group}}[k'] = \mathsf{Group}[i]$. *Then, we claim that* $j = k'$.

Proof. Assume there exists a leaf $k'' \in [\mathsf{rmostLeaf}(u)+1, j-1]$ satisfying $\overline{\alpha\mathsf{LCP}}[k''] = \alpha\mathsf{LCP}[i]$ and $\overline{\mathsf{Group}}[k''] = \mathsf{Group}[i]$. By Lemma 6.7, $\delta(k'', i) \leq \mathsf{lcp}(i,j)$. Let $k'' = \mathsf{succ}(l'')$. Let w be $\mathsf{lca}(i,j)$ or $\mathsf{lca}(l'', k'')$, whichever has lower node-depth, breaking ties arbitrarily. Since $\alpha\mathsf{LCP}[l''] = \alpha\mathsf{LCP}[i]$, it follows that $\mathsf{zeroDepth}(\mathsf{lca}(i,j)) = \mathsf{zeroDepth}(\mathsf{lca}(l'', k''))$. Therefore, no position on the path between $\mathsf{lca}(i,j)$ and $\mathsf{lca}(l'', k'')$ is a point of change (refer to Observation 6.3). Let $w = \mathsf{lca}(l'', k'')$. Then, $\chi(X_i[1, \mathsf{strDepth}(w)], x_i) = \chi(X_i[1, \mathsf{lcp}(i,j)], x_i) = \mathsf{Group}[i] = \overline{\mathsf{Group}}[k''] = \chi(X_{k''}[1, \mathsf{strDepth}(w)], x_{k''})$. This implies $\delta(k'', i) > \mathsf{lcp}(i,j)$, a contradiction. Similarly, choosing $w = \mathsf{lca}(i,j)$ yields a contradiction. ■

Lemma 6.12. *Let* $\alpha\mathsf{LCP}[i] < 0$ *and* $j = \mathsf{succ}(i)$. *Let* v *be the lowest ancestor of* ℓ_i *that has a child* u, *where* $\mathsf{pre\text{-}order}(u) < \mathsf{pre\text{-}order}(\ell_i)$, u *is not an ancestor of* ℓ_i, *and there exists a* $k \in [\mathsf{lmostLeaf}(u), \mathsf{rmostLeaf}(u)]$ *with* **(a)** $\overline{\alpha\mathsf{LCP}}[k] = \alpha\mathsf{LCP}[i]$, **(b)** $\overline{\mathsf{Group}}[k] = \mathsf{Group}[i]$ *and* **(c)** $\mathsf{strDepth}(v)+1$ *is a point of change of* k. *Then, we claim that both* u *and* k *are unique. Moreover,* $j = k$.

Proof. We first note that $j \in [1, \mathsf{rmostLeaf}(u)]$; otherwise, this contradicts the definition of v and u. Let $k = \mathsf{succ}(l)$ for some l. Since $l > k$ and $\mathsf{strDepth}(v)+1$ is a point of change of k, we have $\mathsf{lcp}(k,l) \leq \mathsf{lcp}(k,i)$; otherwise, $\overline{\alpha\mathsf{LCP}}[k] = -\mathsf{zeroDepth}(\mathsf{lca}(k,l)) \leq -(1+\mathsf{zeroDepth}(\mathsf{lca}(k,i))) \leq -(1+\mathsf{zeroDepth}(\mathsf{lca}(j,i))) < \alpha\mathsf{LCP}[i]$, a contradiction. Also, $1+\mathsf{lcp}(k,l)$ is a point of change of k (applying Observation 6.4). If $\mathsf{lcp}(k,l) < \mathsf{lcp}(j,i)$,

then $\mathsf{zeroDepth}(\mathsf{lca}(j,i)) > \mathsf{zeroDepth}(\mathsf{lca}(k,l))$, which contradicts $\alpha\mathsf{LCP}[l] = \alpha\mathsf{LCP}[i]$. If $\mathsf{lcp}(j,i) \leq \mathsf{lcp}(k,l) < \mathsf{lcp}(k,i)$, then applying Observations 6.4, it is straightforward to show that either $j = \mathsf{succ}(i)$ or $k = \mathsf{succ}(l)$ is violated. Hence, $\mathsf{lcp}(j,i) \leq \mathsf{lcp}(k,i) = \mathsf{lcp}(k,l)$. Since $\alpha\mathsf{LCP}[l] = \alpha\mathsf{LCP}[i]$, we have $\mathsf{zeroDepth}(\mathsf{lca}(j,i)) = \mathsf{zeroDepth}(\mathsf{lca}(k,l))$. Therefore, no position on the path between $\mathsf{lca}(j,i)$ and $\mathsf{lca}(k,l)$ is a point of change (refer to Observation 6.3). Then, $\chi(X_i[1,\mathsf{lcp}(k,i)],x_i) = \chi(X_i[1,\mathsf{lcp}(j,i)],x_i) = \mathsf{Group}[i] = \overline{\mathsf{Group}}[k] = \chi(X_k[1,\mathsf{lcp}(k,l)],x_k) = \chi(X_k[1,\mathsf{lcp}(k,i)],x_k)$. Now, by applying Lemma 6.10, we get $\delta(k,i) > \mathsf{lcp}(k,i)$. Using Lemma 6.9, we have $j = k$. ■

6.7 Data Structure Toolkit

Moving forward, any array (where required) has been pre-processed with Fact 2.3. Trees are pre-processed using Fact 2.4. We also need a few supporting data structures, summarized in Lemmas 6.13, 6.14, 6.15, and 6.16.

Lemma 6.13. *By using an $O(n\log\sigma)$ bit data structure, given a leaf ℓ_i and an integer W, we can find the highest ancestor w of ℓ_i satisfying $\mathsf{zeroDepth}(w) \geq W$ in $O(\log\sigma)$ time.*

Proof. Create an array A such that $A[k] = \mathsf{zeroDepth}(u)$, where u is the node with pre-order rank k. We maintain a wavelet tree over A; see Fact 2.3. Given the leaf ℓ_i, we first find the rightmost entry $l < \mathsf{pre\text{-}order}(\ell_i)$ in A such that $A[l] < W$ by using $\mathsf{predecessor}_A(\mathsf{pre\text{-}order}(\ell_i), W-1)$. (Note that since the root node r has $\mathsf{zeroDepth}(r) = 0$, the position l exists.) Let $v' = \mathsf{lca}(\ell_i, v)$, where v is the node with pre-order rank l. Then, $w = \mathsf{levelAncestor}(\ell_i, \mathsf{nodeDepth}(v') + 1)$. To see correctness observe that $\mathsf{zeroDepth}(v') \leq \mathsf{zeroDepth}(v) < W$. If $\mathsf{zeroDepth}(w) < W$, the $\mathsf{predecessor}$-query should have returned w instead of v (since $\mathsf{pre\text{-}order}(v) < \mathsf{pre\text{-}order}(w) \leq \mathsf{pre\text{-}order}(\ell_i)$). ■

Lemma 6.14. *Let C be a positive integral constant. Consider a collection of arrays $X_i[1,t]$, $i \in [1,C]$, with elements from $[\sigma]$. By using an $O(t\log\sigma)$-bit data structure, we can compute the following in $O(\log\sigma)$ time:*

- $\mathsf{rank}_{X_1,X_2,\ldots,X_C}(L, R, \alpha_1, \alpha_2, \ldots, \alpha_C)$ = *the number of positions* $k \in [L, R]$, *where* $X_i[k] = \alpha_i, \forall i \in [1, C]$.

- $\mathsf{select}_{X_1,X_2,\ldots,X_C}(L, R, q, \alpha_1, \alpha_2, \ldots, \alpha_C)$ = *the* q*th (if exists) position* $k \in [L, R]$, *where* $X_i[k] = \alpha_i, \forall i \in [1, C]$.

Proof. Let $\tau_i = \langle X_1[i], X_2[i], \ldots, X_C[i] \rangle$, $i \in [1, t]$. Note that τ_i can be mapped uniquely to a symbol from an alphabet Σ' of size σ^C; the mapping is computed in $O(1)$ time. Let $X[1, t]$ be an array, where $X[i] \in \Sigma'$ corresponds to τ_i. Now, the required operations are computed easily via rank and select operations on X, which is facilitated by Fact 2.3. The space needed is $O(t \log \sigma^C) = O(t \log \sigma)$ bits. The time for each operation is $O(\log \sigma^C) = O(\log \sigma)$. ■

Lemma 6.15. *Consider a tree having* t *nodes, where each non-leaf node has at least two children. Also, each node is associated with a* 0 *or* 1*. By using an* $O(t)$*-bit data structure, in* $O(1)$ *time, we can find the lowest ancestor of a leaf that is associated with a* 1*.*

Proof. Starting from the leftmost leaf, every $g = c\lceil \log t \rceil$ leaves form a group, where c is a constant to be decided later. (The last group may have fewer than g leaves.) Mark the lowest common ancestor of the first and last leaf of each group. At each marked node, write the node-depth of its lowest ancestor which is associated with a 1. The space needed is $O(\frac{t}{g} \log t) = O(t)$ bits. Let τ_u be the subtree rooted at a marked node u; note that τ_u has at most $2g$ nodes. Since each node in τ_u is associated with a 0 or 1, the number of possible trees is 2^{2g}. We store a pointer from u to τ_u. The total space needed for storing all pointers is $O(\frac{t}{g} \log 2^{2g}) = O(t)$ bits.

For each possible τ_u, store the following satellite data in an additional array. Consider the kth leftmost leaf ℓ_k in τ_u. Let v be the lowest node on the path from u to ℓ_k associated with a 1. If v exists, store the node-depth of v relative to u, else store -1. The space needed for each τ_u is $O(2g \log g) = O(g \log \log t)$ bits. Therefore, the total space for all such trees is $O(2^{2g} g \log \log t)$. By choosing $c = 1/4$, this space is bounded by $o(t)$ bits. Thus, the total space is bounded by $O(t)$ bits.

Given a query leaf ℓ_k, we first locate the lowest marked node u^* of ℓ_k; this is given by $u^* = \mathsf{lca}(1 + g\lfloor k/g \rfloor, \max\{t, g(1 + \lfloor k/g \rfloor)\})$. Let d^* be the depth stored at u^*. Let $k' = k - g\lfloor k/g \rfloor$. Check the k'th entry of the satellite array of u^*, and let it be d. If $d = -1$, then assign $D = d^*$, else assign $D = \mathsf{nodeDepth}(u^*) + d$. The desired ancestor of ℓ_k is given by the query $\mathsf{levelAncestor}(\ell_k, D)$. ■

Lemma 6.16. *Consider a tree of t nodes, where each node is either special or normal. By using an $O(t)$-bit data structure, in $O(1)$ time, given a node v, we can find a node u (if any) such that u is the rightmost special child of $\mathsf{parent}(v)$ and $\mathsf{pre\text{-}order}(u) < \mathsf{pre\text{-}order}(v)$.*

Proof. For each node w, we store a bit-vector $B_w[t_w]$, where t_w is the number of children of w. Assign $B_w[i] = 1$ iff $\mathsf{child}(w, i)$ is special. The space needed is $O(t)$ bits. Given the query node v, we go to the bit vector $B_{v'}$, where $v' = \mathsf{parent}(v)$. Let $k = \mathsf{rank}_{B_{v'}}(\mathsf{sibRank}(v), 1)$. If $k = 0$, then u does not exist; otherwise, $u = \mathsf{child}(v', \mathsf{select}_{B_{v'}}(k, 1))$. ■

6.8 Wrapping Up

In this section, we prove Theorem 6.2 by combining the data structure tools in Section 6.7 with Lemmas 6.3, 6.11, and 6.12. Specifically, in Lemma 6.17 we show that in each of the three scenarios of Sections 6.5, 6.6.1, and 6.6.2, we can compute $\mathsf{succ}(i)$ in $O(\log \sigma)$ time by using an $O(n \log \sigma)$-bit data structure. As a straightforward consequence, we get Theorem 6.2.

Lemma 6.17. *By maintaining an $O(\kappa \log \sigma)$-bit data structure, we can find the $j = \mathsf{succ}(i)$ in $O(\log \sigma)$ time when either* **(a)** $\alpha\mathsf{Disagree}[i] > 0$, *or* **(b)** $\alpha\mathsf{LCP}[i] > 0$, *or* **(c)** $\alpha\mathsf{LCP}[i] < 0$.

Proof. We treat Group as two separate arrays $G_<$ and $G_>$, where $\mathsf{Group}[i] = \langle G_<[i], G_>[i] \rangle$. Similarly, $\overline{\mathsf{Group}}$ is maintained as $\bar{G}_<$ and $\bar{G}_>$. We use $\mathsf{rank}_{\alpha\mathsf{LCP},\mathsf{Group}}(L, R, \alpha, \beta)$ to denote $\mathsf{rank}_{\alpha\mathsf{LCP},G_<,G_>}(L, R, \alpha, \beta_<, \beta_>)$ query of Lemma 6.14, where $\beta = \langle \beta_<, \beta_> \rangle$. The select operation in Lemma 6.14 is analogously defined; so are the operations on $\overline{\alpha\mathsf{LCP}}$ and $\overline{\mathsf{Group}}$.

(a) Using Lemma 6.13, in $O(\log \sigma)$ time, we first find the highest node u on the root to ℓ_i path such that $\alpha\mathsf{Disagree}[i] \leq \mathsf{zeroDepth}(u)$. Then $j = \mathsf{select}_{\overline{\alpha\mathsf{Disagree}}}(c+1, \alpha\mathsf{Disagree}[i])$, where $c = \mathsf{rank}_{\overline{\alpha\mathsf{Disagree}}}(\mathsf{lmostLeaf}(u)-1, \alpha\mathsf{Disagree}[i])$; both j and c are computed in $O(\log \sigma)$ time using Facts 2.3 and 2.4. The correctness follows from Lemma 6.3.

(b) Let u and w be the nodes corresponding to ℓ_i as defined in Lemma 6.11. We store a bit-vector $B[1, \kappa]$ to determine whether $u = \ell_i$, or not. If $u \neq \ell_i$, then by storing another bit-vector $B'[1, \kappa]$ we can identify whether $u = w$ or $u = \mathsf{parent}(w)$. We store the $\alpha_i = \alpha(X_i[1, \delta(i, k)])$ with the leaf ℓ_i, where k is defined in Lemma 6.11. Then, node w is the highest node w' on the root to ℓ_i path such that $\mathsf{zeroDepth}(w') \geq \alpha_i$; we can find w in $O(\log \sigma)$ time using Lemma 6.13 and α_i. Finally, $j = \mathsf{select}_{\overline{\alpha\mathsf{LCP}}, \overline{\mathsf{Group}}}(\mathsf{rmostLeaf}(u) + 1, \kappa, 1, \alpha\mathsf{LCP}[i], \mathsf{Group}[i])$ is computed using Lemma 6.14 in $O(\log \sigma)$ time.

(c) Strip the tree $\mathcal{T}_{\mathcal{X}}$ into several compacted tries as follows. Consider a leaf ℓ_k such that $\alpha\mathsf{LCP}[k] = \alpha < 0$ and $\mathsf{Group}[k] = \beta$. Keep only the leaves ℓ_l in the tree for which either $\alpha\mathsf{LCP}[l] = \alpha$ and $\mathsf{Group}[l] = \beta$ or $\overline{\alpha\mathsf{LCP}}[l] = \alpha$ and $\overline{\mathsf{Group}}[l] = \beta$. Transform the tree into a compacted trie, and denote it by $\tau_{\alpha,\beta}$ having $\kappa_{\alpha,\beta}$ nodes and $m_{\alpha,\beta}$ leaves.

Given a leaf p in $\mathcal{T}_{\mathcal{X}}$, let $\mathsf{leaf}_{\alpha,\beta}(p)$ be the equivalent leaf in $\tau_{\alpha,\beta}$. Likewise, given a leaf q in $\tau_{\alpha,\beta}$, let $\mathsf{leaf}^{-1}_{\alpha,\beta}(q)$ be its equivalent leaf in $\mathcal{T}_{\mathcal{X}}$. We store a bit-vector $B_{\alpha,\beta}[1, m_{\alpha,\beta}]$ (resp. $\bar{B}_{\alpha,\beta}[1, m_{\alpha,\beta}]$) as follows. We set $B_{\alpha,\beta}[q] = 1$ (resp. $\bar{B}_{\alpha,\beta}[q] = 1$) iff $\alpha\mathsf{LCP}[\mathsf{leaf}^{-1}_{\alpha,\beta}(q)] = \alpha$ and $\mathsf{Group}[\mathsf{leaf}^{-1}_{\alpha,\beta}(q)] = \beta$ (resp. $\overline{\alpha\mathsf{LCP}}[\mathsf{leaf}^{-1}_{\alpha,\beta}(q)] = \alpha$ and $\overline{\mathsf{Group}}[\mathsf{leaf}^{-1}_{\alpha,\beta}(q)] = \beta$). Using $B_{\alpha,\beta}$, $\bar{B}_{\alpha,\beta}$, and Lemma 6.14, we compute $\mathsf{leaf}_{\alpha,\beta}$ and $\mathsf{leaf}^{-1}_{\alpha,\beta}$ in $O(\log \sigma)$ time as follows.

$$\mathsf{leaf}_{\alpha,\beta}(p) = \begin{cases} \mathsf{select}_{B_{\alpha,\beta}}(\mathsf{rank}_{\alpha\mathsf{LCP},\mathsf{Group}}(1, p, \alpha, \beta), 1) \text{ if } \alpha\mathsf{LCP}[p] = \alpha \text{ and } \mathsf{Group}[p] = \beta \\ \mathsf{select}_{\bar{B}_{\alpha,\beta}}(\mathsf{rank}_{\overline{\alpha\mathsf{LCP}},\overline{\mathsf{Group}}}(1, p, \alpha, \beta), 1) \text{ if } \overline{\alpha\mathsf{LCP}}[p] = \alpha \text{ and } \overline{\mathsf{Group}}[p] = \beta \end{cases}$$

$$\mathsf{leaf}^{-1}_{\alpha,\beta}(q) = \begin{cases} \mathsf{select}_{\alpha\mathsf{LCP},\mathsf{Group}}(1, n, \mathsf{rank}_{B_{\alpha,\beta}}(q, 1), \alpha, \beta) \textbf{ if } B_{\alpha,\beta}[q] = 1 \\ \mathsf{select}_{\overline{\alpha\mathsf{LCP}},\overline{\mathsf{Group}}}(1, n, \mathsf{rank}_{\bar{B}_{\alpha,\beta}}(q, 1), \alpha, \beta) \textbf{ if } \bar{B}_{\alpha,\beta}[q] = 1 \end{cases}$$

Since the number of possible trees $\tau_{\alpha,\beta}$ is at most σ^3, a pointer from a leaf in $\mathcal{T}_{\mathcal{X}}$ to its corresponding tree $\tau_{\alpha,\beta}$ needs $\lceil 3 \log \sigma \rceil$ bits. Since each leaf in $\mathcal{T}_{\mathcal{X}}$ appears in at most two trees (once for $\alpha\mathsf{LCP}$ and Group, and once for $\overline{\alpha\mathsf{LCP}}$ and $\overline{\mathsf{Group}}$), we have $\sum \kappa_{\alpha,\beta} = O(\kappa)$.

We *associate* a node u in $\tau_{\alpha,\beta}$ with 1 iff $\mathsf{parent}(u)$ has a child v before u in pre-order, such that there is a leaf q in v's subtree satisfying the following conditions: $\overline{\alpha\mathsf{LCP}}[q] = \alpha$, $\overline{\mathsf{Group}}[q] = \beta$, and the leading position on the edge $(\mathsf{parent}(u), v)$ is a point of change of q (i.e., $\mathsf{strDepth}(\mathsf{parent}(u)) + 1$ is a point of change of $\mathsf{leaf}^{-1}_{\alpha,\beta}(q)$). Also we denote the node v as a *special* node. Pre-process $\tau_{\alpha,\beta}$ with Lemmas 6.15 and 6.16.

Consider two leaves k and k' in $\mathcal{T}_{\mathcal{X}}$, where $k' < k$, $\alpha\mathsf{LCP}[k] = \overline{\alpha\mathsf{LCP}}[k'] = \alpha$ and $\mathsf{Group}[k] = \overline{\mathsf{Group}}[k'] = \beta$. We observe that there is a node u in $\tau_{\alpha,\beta}$ such that $\mathsf{strDepth}(u) = \mathsf{lcp}(k, k')$. Moreover, if $1 + \mathsf{lcp}(k, k')$ is a point of change of k', then the child of u on the u to $\mathsf{leaf}_{\alpha,\beta}(k')$ path is a special node, and the child of u on the u to $\mathsf{leaf}_{\alpha,\beta}(k)$ path is associated with a 1. Hence, given the query i, we can find j as follows. We first jump to the leaf $\ell = \mathsf{leaf}_{\alpha\mathsf{LCP}[i],\mathsf{Group}[i]}(i)$ in $\tau_{\alpha\mathsf{LCP}[i],\mathsf{Group}[i]}$. Using Lemma 6.15, locate the lowest ancestor u of ℓ associated with a 1. We find the special sibling v of u to its left using Lemma 6.16. Finally, $j = \mathsf{leaf}^{-1}_{\alpha\mathsf{LCP}[i],\mathsf{Group}[i]}(\mathsf{select}_{\bar{B}}(\mathsf{rank}_{\bar{B}}(\mathsf{rmostLeaf}(v), 1), 1)$. The total time to locate j is $O(\log \sigma)$ time, as desired. ■

Chapter 7
Compact Index for Order-preserving Dictionary Matching

Recall the order-preserving matching (o-match) of two strings as outlined in Definition 6.1. We consider the *Order-preserving Dictionary Indexing* problem of Kim et al. [KEF+14].

Problem 7.1 (Order-preserving Dictionary Indexing [KEF+14])**.** *Let $\mathcal{D}$ be a collection of d patterns $\{P_1, P_2, \ldots, P_d\}$ of total length n; each P_i contains characters from $\Sigma = \{1, 2, \ldots, \sigma\}$. The task is to index $\mathcal{D}$, such that given a text T (also over Σ), we can report all pairs $\langle j, P_i \rangle$, such that there exists a pattern $P_i \in \mathcal{D}$ which is an order-preserving match with $T[j - |P_i| + 1, j]$.*

Largely based on the Aho-Corasick (AC) automaton [AC75], Kim et al. [KEF+14] presented a $\Theta(m \log m)$-bit index, where $m \leq n+1$ is the number of states in the automaton, that can report all *occ* pairs in time $O(|T| \log \sigma + occ)$. We present the following space-efficient alternative.

Theorem 7.1. *By using an $O(m \log \sigma + d \log \frac{m}{d})$-bit index, all pairs $\langle j, P_i \rangle$, such that there exists a pattern $P_i \in \mathcal{D}$ which is an order-preserving match with $T[j - |P_i| + 1, j]$, can be found in $O(|T| \log m + occ)$ time, where occ is the number of such pairs and $m \leq n + 1$ is the number of states in the automaton of Kim et al. [KEF+14].*

7.1 Overview of Techniques

We begin with a linear-space based on the classical AC automaton [AC75] and the encoding scheme of Crochemore et al. [CIK+13]. The idea is to first encode each pattern using Crochemore et al.'s encoding defined in Section 6.2.1, and then create a compacted trie $\mathcal{T}$ of these encode patterns. Each node in the trie is associated with three components: a next link that says what is the next node in the trie where we should go in case of a match, a failure link that says what is the next node in the trie where we should go in case

of a mismatch, and a report link that says what are the patterns which have an occurrence ending at the current position of the text. Section 7.2 contains the detailed description.

The first question is how to represent the nodes succinctly, in the sense that given a node, we can quickly find out if there is a pattern which has the same encoding as the path from root to this node; these nodes are termed final nodes. The other importance of node representation is to quickly simulate the three transitions above. Given a node v in the trie, let $\overleftarrow{\mathsf{order}}(v) = \mathsf{order}(\overleftarrow{P_i[1, \delta_v]})$, where P_i is a pattern corresponding to a final node in the subtree of v, δ_v is the node-depth of v, and $\overleftarrow{S}$ denotes the reverse of a string S. We conceptually label a node u with the lexicographic rank of $\overleftarrow{\mathsf{order}}(u)$ in the set $\{\overleftarrow{\mathsf{order}}(v) \mid v \text{ is a node in the trie}\}$. Labels of final nodes are explicitly stored, enabling us to find the pattern corresponding to any final node quickly. These ideas are borrowed (and extended) from Belazzougui [Bel10]. Section 7.3 contains the details.

The next task (and arguably the most difficult one) is to simulate the next transition. Here, given the label of u, we are seeking the label of a child v of u, where we should proceed in case of a match. We create another trie $\overleftarrow{\mathcal{T}}$ of the strings $\overleftarrow{\mathsf{order}}(w)$ for each node w in $\mathcal{T}$. Now each node u in $\mathcal{T}$ can be mapped to a leaf ℓ_u in $\overleftarrow{\mathcal{T}}$, and the label of u can be found by applying a LF mapping-like transform on the corresponding leaf ℓ_u. The central question is how to implement this transform. We observe that in essence what we are trying to do is as follows. There is a collection of strings (corresponding to the nodes in $\mathcal{T}$) and each string is associated with a character (labeling the edge from a node to one of its child). Now, we are trying to find what is the lexicographic rank of a string (corresponding to a node) when prepended by its associated character (labeling the edge to its child). Clearly, one may apply Theorem 6.2 to achieve this. Section 7.4 contains the details.

The final piece is to simulate the failure and report transitions. However, this is rather simple and is immediate from Belazzougui [Bel10]; the details are in Section 7.5. Once, we have all the above components, the final query procedure is largely a straightforward extension of the query procedure of the linear index; Section 7.6 contains the details.

7.2 Linear Space Index

We first present a linear space index, which is based on the AC automaton [AC75]. Recall the following encoding scheme of Crochemore et al. [CIK+13], outlined in Section 6.2.1. We convert a string S to a string $\mathsf{order}(S)$ as follows. Let $i \in [1, |S|]$ and p_- (resp. p_+) be the highest value (resp. lowest value) in $S[1, i-1]$ that is at most $S[i]$ (resp. at least $S[i]$). Let j_- (resp. j_+) be the rightmost occurrence of p_- (resp. p_+) in $[1, i)$. If p_- (resp. p_+) does not exist, then assign $j_- = i$ (resp. $j_+ = i$). Assign, $\mathsf{order}(S)[i] = \langle i - j_-, i - j_+ \rangle$.

Two pairs $\langle x_i, y_i \rangle$ and $\langle x_j, y_j \rangle$ in this encoding scheme are the same iff $x_i = x_j$ and $y_i = y_j$. Two strings X and Y are order-preserving iff $\mathsf{order}(X) = \mathsf{order}(Y)$. Also, X is order-preserving with a prefix of Y iff $\mathsf{order}(X)$ is a prefix of $\mathsf{order}(Y)$. See Fact 6.1.

7.2.1 The 3 Main Components of the Linear Index

Compute $\mathsf{order}(P_i)$ for every P_i in $\mathcal{D}$, and then create a trie $\mathcal{T}$ for all the encoded patterns. Let the number of nodes in the trie be m, where $m \leq n + 1$. For a node u, denote by $\mathsf{order}(u)$ the string formed by concatenating the edge labels from root to u. Mark a node u in the trie as *final* iff $\mathsf{order}(u) = \mathsf{order}(P_i)$ for some P_i in $\mathcal{D}$. Clearly, the number of final nodes is d, i.e., the number of patterns in the dictionary. For any node u, define $\mathsf{strDepth}(u) = |\mathsf{order}(u)|$ and $\zeta(u, j) = \mathsf{order}(P_i[j, \mathsf{strDepth}(u)])$, where P_i is a pattern whose corresponding final node lies in the subtree rooted at u. Each node u is associated with 3 links as defined below:

- $\mathsf{next}(u, c) = v$ iff the label on the edge from u to v is labeled by the character c.
- $\mathsf{failure}(u) = v$ iff $\mathsf{order}(v) = \zeta(u, j)$, where $j > 1$ is the smallest index for which such a node v exists. If no such j exists, then $\mathsf{failure}(u)$ points to the root node. This represents the smallest shift to be performed in T in case of a mismatch.
- $\mathsf{report}(u) = v$ iff v is a final node and $\mathsf{order}(v) = \zeta(u, j)$, where $j > 1$ is the smallest index for which v exists. If no such j exists, then $\mathsf{report}(u)$ points to the root node. This represents a pattern with an occurrence ending at the current text position.

The total space needed is $\Theta(m \log m)$ bits. We note that the number of states in $\mathcal{T}$ is the same as the number of states in the trie of Kim et al. [KEF+14] for the collection $\mathcal{D}$.

7.2.2 The Querying Algorithm

To find the occurrences, we use a balanced binary search tree (BST) that stores the symbols in Σ that appear within a certain sliding window of T. To this end, we maintain an array $A[1, \sigma]$ such that $A[c]$ equals the position of the latest occurrence of $c \in \Sigma$. Now, match T in the trie as follows. Suppose, we are considering the position j in T (initially, $j = 1$), and we are at a node u, i.e., we have matched $T[j - \mathsf{strDepth}(u) + 1, j]$ in the trie. First, repeatedly follow report-links starting from u until the root node is reached, thereby, reporting all patterns with a match ending at j. Now, look at the character $T[j + 1]$ to match. We have to obtain $c_{j,u} = \mathsf{order}(T[j - \mathsf{strDepth}(u) + 1, |T|])[\mathsf{strDepth}(u) + 1]$. Using the BST, find the largest (resp. smallest) number within the window that is at most (resp. at least) $T[j + 1]$. Now, use the array A to find the rightmost occurrence of these numbers within the window to obtain the desired encoding. If $v = \mathsf{next}(u, c_{j,u})$ is defined, follow it, update the BST (by including $T[j+1]$ if it is already not present) and the array A (by letting $A[T[j+1]] = j+1$) to incorporate the symbol $T[j + 1]$. Repeat by letting $v = u$ and $j = j + 1$; in this case, the right boundary of the sliding window has slid by one position. Otherwise if $v = \mathsf{next}(u, c_{j,u})$ is not defined, follow $\mathsf{failure}(u)$ to a node w and repeat by letting $w = u$; in this case, the left boundary of the sliding window is going to change. Specifically, we are going to slide over $\mathsf{strDepth}(u) - \mathsf{strDepth}(w)$ characters. For each character c slid over, we check $A[c]$ to check if it is in the current sliding window. If it is not, we remove c from the BST. We continue this process until the last character of T is read. The number of all deletion, search, and insertion operations in the BST is at most $3|T|$, each requiring $O(\log \sigma)$ time. Hence, each character in T is encoded in $O(\log \sigma)$ amortized time. On following a report link, either we report an occurrence or we reach the root. Then, either we take a next transition or we follow a $\mathsf{failure}$ link; the number of such operations combined is $\leq 2|T|$. Each transition takes $O(1)$ time. Therefore, the total time required is $O(|T| \log \sigma + occ)$.

7.3 Representing States Succinctly

Broadly speaking, we use Belazzougui [Bel10]'s succinct representation of the AC automaton [AC75]. Let $\mathcal{T}$ be the trie in Section 7.2. We observe that any node $u \in \mathcal{T}$ has a final node in its subtree $\mathcal{T}(u)$. Let $\overleftarrow{\mathsf{order}}(u) = \mathsf{order}(P_i[\mathsf{strDepth}(u)] \circ P_i[\mathsf{strDepth}(u)-1] \circ \cdots \circ P_i[1])$, where P_i is the pattern corresponding to a final node in $\mathcal{T}(u)$ and $p_i = |P_i|$. Each state u is *conceptually labeled* by the lexicographic rank of $\overleftarrow{\mathsf{order}}(u)$ in the set $\{\overleftarrow{\mathsf{order}}(v) \mid v \text{ is a node in the trie}\}$. Thus, each state is labeled by a number in $[1, m]$, where the root is labeled by 1.

Convention 7.1. *Without loss of generality, assume that no two patterns P_i and P_j exist such that $\mathsf{order}(P_i) = \mathsf{order}(P_j)$. Also, assume that $i < j$ iff $\overleftarrow{\mathsf{order}}(P_i)$ precedes $\overleftarrow{\mathsf{order}}(P_j)$ in the lexicographic order, where $\overleftarrow{\mathsf{order}}(P) = \mathsf{order}(P[p] \circ P[p-1] \circ \cdots \circ P[1])$ and $p = |P|$.*

We *explicitly store* the labels of the final states using the SID of Fact 2.5. Since there are d final nodes, the space required is $d\log(m/d) + O(d)$ bits. Given the label of a final state v, we first find the rank of v among all the final states using Fact 2.5. If the rank is r, then v corresponds to the pattern P_r by Convention 7.1. Thus, given the label of a final state, we can find the corresponding pattern in $O(1)$ time, leading to the following lemma:

Lemma 7.1. *Given the label of a final state, we can find the corresponding pattern in $O(1)$ time by using an $d\log(m/d) + O(d)$-bit data structure.*

Lastly, we maintain a bit-vector $\mathsf{leaf}[1, m]$ such that $\mathsf{leaf}[j] = 1$ iff the state with label j is a leaf in $\mathcal{T}$. The total space for representing the states is $m + d\log(m/d) + O(d)$ bits.

7.4 Handling next Transitions Compactly

We create a compressed $\overleftarrow{\mathcal{T}}$ as follows. Initially $\overleftarrow{\mathcal{T}}$ is empty. For each non-leaf node u in $\mathcal{T}$ and each child u_i of u, we add the string $\overleftarrow{\mathsf{order}}(u) \circ \$_{u,i}$ to $\overleftarrow{\mathcal{T}}$. Clearly, the string corresponds to a unique leaf, say $\ell_{u,i}$, in $\overleftarrow{\mathcal{T}}$. For any node $u \in \overleftarrow{\mathcal{T}}$, we define $\mathsf{path}(u)$ as the concatenation of the order-encoded edge labels from the root to u.

Let $\ell_{u,i}$ denote the ith leftmost leaf in $\overleftarrow{\mathcal{T}}$, where u_i is the non-root node in $\mathcal{T}$ that corresponds to $\ell_{u,i}$, i.e., $\mathsf{path}(\ell_{u,i}) = \overleftarrow{\mathsf{order}}(u) \circ \$_{u,i}$, and $u = \mathsf{parent}(u_i)$. We order the leaves in $\overleftarrow{\mathcal{T}}$ as follows. Consider the leaves $\ell_{u,i}$ and $\ell_{u,j}$ in $\overleftarrow{\mathcal{T}}$ such that the corresponding nodes u_i and u_j in $\mathcal{T}$ share the same parent u. (Note that $\ell_{u,i}$ and $\ell_{u,j}$ also share the same parent in $\overleftarrow{\mathcal{T}}$.) Then, the leaf $\ell_{u,i}$ lies to the left of $\ell_{u,j}$ in $\overleftarrow{\mathcal{T}}$ (i.e., $i < j$) if $\overleftarrow{\mathsf{order}}(u_i)$ is lexicographically smaller than $\overleftarrow{\mathsf{order}}(u_j)$. Now, consider two leaves $\ell_{u,i}$ and $\ell_{v,j}$ in $\overleftarrow{\mathcal{T}}$ with the corresponding nodes u_i and v_j in $\mathcal{T}$ having different parents u and v. Then, $\ell_{u,i}$ lies to the left of $\ell_{v,j}$ (i.e., $i < j$) if $\overleftarrow{\mathsf{order}}(u)$ is lexicographically smaller than $\overleftarrow{\mathsf{order}}(v)$.

Note that the number leaves in $\overleftarrow{\mathcal{T}}$ is same as the number of edges in $\mathcal{T}$. Therefore, it has $(m-1)$ leaves and at most $(m-2)$ internal nodes.

Lemma 7.2. *Consider the ith leftmost leaf $\ell_{u,i}$ in $\overleftarrow{\mathcal{T}}$. By using an $O(m \log \sigma)$-bit data structure, we can compute the label of the corresponding node $u_i \in \mathcal{T}$ in $O(\log m)$ time.*

Proof. Define $\mathsf{leafRank}(i)$ as the rank of the string $\overleftarrow{\mathsf{order}}(u_i)$ among the strings $\{\overleftarrow{\mathsf{order}}(x) \mid x \in \mathcal{T}$ is a non-root node$\}$. Note that the root node in $\mathcal{T}$ is the only node that does not have a corresponding leaf in $\overleftarrow{\mathcal{T}}$. Since the label of the root node is 1, this definition implies that the label of a node $u_i \in \mathcal{T}$ is given by $1 + \mathsf{leafRank}(i)$; recall that u_i corresponds to the ith leftmost leaf in $\overleftarrow{\mathcal{T}}$. Therefore, the main task is to compute $\mathsf{leafRank}(i)$.

Let $u = \mathsf{parent}(u_i)$. Define the collection of strings $\mathcal{X} = \{X_1, X_2, \ldots, X_{m-1}\}$ as defined in Theorem 6.2, such that $X_i = P_k[\mathsf{strDepth}(u)] \circ P_k[\mathsf{strDepth}(u) - 1] \circ \cdots \circ P_k[1] \circ \$_{u,i}$ and $x_i = P_k[\mathsf{strDepth}(u_i)]$, where P_k is a pattern corresponding to a final node in the subtree of u_i in $\mathcal{T}$. By letting $\mathsf{order}(X_i) = \mathsf{order}(X_i[1, |X_i| - 1])\langle \$_{u,i}, \$_{u,i} \rangle$, we ensure that the collection is prefix-free. It follows that $\mathsf{leafRank}(i) = \mathsf{lexRank}(i)$ of Theorem 6.2. To compute $\mathsf{lexRank}(i)$, we use the following procedure.

We employ perfect hashing [FKS84] to store the $\langle j, \mathsf{lexRank}(j) \rangle$ key-value pairs for all j such that $\mathsf{lexRank}(j)$ belongs to $\{1, 1+\Delta, 1+2\Delta, 1+3\Delta, \ldots, m-1\}$, where $\Delta = \lceil \log m \rceil$. Using this, given a j, one can check if $\mathsf{lexRank}(j)$ has been stored (and also retrieve the value) in $O(1)$ time. The space needed is $O((m/\Delta) \log m)$ bits. Then $\mathsf{lexRank}(i)$ is computed in

$O(1)$ time if its value has been explicitly stored. Otherwise, we apply the succ operation of Theorem 6.3 starting from i until we reach an index j such that $\mathsf{lexRank}(j)$ has been explicitly stored. Let the number of succ operations be k, and let the indexes yielded be $i_1, i_2, \ldots, i_k = j$ in that order. We have the following telescoping series: $\mathsf{lexRank}(i_1) = \mathsf{lexRank}(i) + 1, \mathsf{lexRank}(i_2) = \mathsf{lexRank}(i_1) + 1, \ldots, \mathsf{lexRank}(i_k) = \mathsf{lexRank}(i_{k-1}) + 1$. Then, $\mathsf{lexRank}(i_k) = \mathsf{lexRank}(j) = \mathsf{lexRank}(i) + k$. Clearly, $k \leq \Delta$ and the time required is $O(\Delta \cdot \log \sigma) = O(\log m)$. ■

Since explicitly storing the order-encoded label of an edge in $\mathcal{T}$ requires $O(\log m)$ bits, we are stuck with the question of uniquely identifying an outgoing edge of a node v. To this end, we introduce the following terminologies.

Definition 7.1. *For any string S and a character c, recall the definition of $\chi(S, c)$ in Definition 6.6. For a non-root node u, we define $\beta(u) = \chi(P_r[1, \mathsf{strDepth}(u)-1], P_r[\mathsf{strDepth}(u)])$, where P_r is a pattern corresponding to a final node in the subtree of u.*

Lemma 7.3. *For any two children v_i and v_j of a node $v \in \mathcal{T}$, we have $\mathsf{order}(v_i) \neq \mathsf{order}(v_j)$ implying $\beta(v_i) \neq \beta(v_j)$. Therefore, for any string $S[1, s]$, we have $\mathsf{order}(S) = \mathsf{order}(v)$ iff $\beta(v) = \chi(S[1, s-1], S[s])$ and $\mathsf{order}(S[1, s-1]) = \mathsf{order}(\mathsf{parent}(v))$.*

Proof. Follows from Lemma 6.10. ■

Hence, an equivalent way of checking if the next transition of a node u points to v is by checking $\beta(v)$. We formalize the notion in the following lemma.

Lemma 7.4. *Let u be a node in $\mathcal{T}$ having label k. By using an $O(m \log \sigma)$-bit data structure, in $O(\log \sigma)$ time, we can detect if u has a child u_i such that $\beta(u_i) = \langle x, y \rangle$ for two integers $0 \leq x, y \leq \sigma$. If u_i exists, we can find the leaf $\ell_{u,i} \in \overleftarrow{\mathcal{T}}$ that corresponds to u_i.*

Proof. If u is a leaf node, i.e., $\mathsf{leaf}[k] = 1$, then u does not have a child and the lemma trivially holds. So, assume otherwise. Maintain a bit-vector $B[1, m-1]$ such that $B[1] = 1$, and for $r > 1$, $B[r] = 1$ iff rth leftmost leaf in $\overleftarrow{\mathcal{T}}$ has a different parent than the previous

leaf. Maintain two arrays $A_<[1, m-1]$ and $A_>[1, m-1]$ such that $\langle A_<[r], A_>[r]\rangle = \beta(w_r)$, where $w_r \in \mathcal{T}$ is the node corresponding to $\ell_{w,r} \in \overleftarrow{\mathcal{T}}$. We pre-process the arrays $A_<$ and $A_>$ with Lemma 6.14.

Given the node $u \in \mathcal{T}$, we can find the node $u' \in \overleftarrow{\mathcal{T}}$ such that $\mathsf{path}(u') = \overleftarrow{\mathsf{order}}(u)$ as follows. Note that u' is the parent of the $(k - \mathsf{rank}_{\mathsf{leaf}}(k, 1))$th leftmost leaf in $\overleftarrow{\mathcal{T}}$ that has a different parent than the previous leaf. Based on this, $u' = \mathsf{parent}(\ell_j)$, where $j = \mathsf{select}_B(k - \mathsf{rank}_{\mathsf{leaf}}(k, 1), 1)$. Find $j' = \mathsf{select}_{A_<,A_>}(1, m-1, 1 + \mathsf{rank}_A(1, j-1, x, y), x, y)$ using Lemma 6.14. Then u_i exists iff $j' \in [\mathsf{lmostLeaf}(u'), \mathsf{rmostLeaf}(u')]$. If u_i exists, by applying Lemma 7.3, we conclude that the desired leaf $\ell_{u,i}$ is the j'th leftmost leaf in $\overleftarrow{\mathcal{T}}$. ■

Now we have the ingredients to prove the following main result in this section.

Lemma 7.5. *Let $T_0 = T[j_0, |T|]$, $1 \le j_0 \le |T|$. Suppose we are at a node u in $\mathcal{T}$ and have matched $T_0[1, j]$. Given the label k of u, we can find the label (if any) of $\mathsf{next}(u, \mathsf{order}(T_0)[j+1])$ in $O(\log \sigma)$ amortized time by using an $O(m \log \sigma)$-bit data structure.*

Proof. Based on Lemma 7.3, the task is to detect whether there exists a child u_i of u such that $\beta(u_i) = x_j$, where $x_j = \chi(T_0[1, j], T_0[j+1])$. Once x_j is known, we use Lemma 7.4 to test this condition, and if u_i exists, find the leaf $\ell_{u,i} \in \overleftarrow{\mathcal{T}}$ corresponding to u_i. Then apply Lemma 7.2 to compute the label of u_i in $O(\log m)$ time.

Since storing $\mathsf{strDepth}$ requires $m\lceil \log m \rceil$ bits in total, we cannot directly store it for computing $\beta(u_i)$. Instead, we store an array Z such that $Z[r]$ equals the number of distinct characters in $P_x[1, \mathsf{strDepth}(u)]$, where P_x is a pattern corresponding to a final node in the subtree of u and u is the node with label r. The total space needed is $O(m \log \sigma)$ bits.

To compute $\beta(u_i)$, we make sure that only $Z[k]$ distinct symbols remain in the current sliding window of T (recall the query procedure in Section 7.2.2). Crucially, observe that x_j can now be computed by finding the number of symbols within the window that are less/more than $T[j+1]$. Therefore, x_j is computed in $O(\log \sigma)$ amortized time, and the proof follows. ■

7.5 Handling failure and report Transitions Succinctly

Note that for any two nodes u and v, if $\mathsf{failure}(u) = v$, then it $\overleftarrow{\mathsf{order}}(v)$ is the longest prefix of $\overleftarrow{\mathsf{order}}(u)$ that appears in $\mathcal{T}$. Similar remarks hold for $\mathsf{report}(u) = v$, where v is a final node. Therefore, these behave exactly in the same manner as in the case of traditional pattern matching, and we can simply use the idea of Belazzougui [Bel10] to perform these transitions (see Sections 3.3 and 3.4 in [Bel10]). We get the following lemma.

Lemma 7.6. *By using an $O(m + d\log(m/d))$-bit data structure, given the label of a node u, we can find the label of* $\mathsf{failure}(u)$ *or* $\mathsf{report}(u)$ *in $O(1)$ time.*

7.6 The Final Query Procedure

Suppose we are at a node u in $\mathcal{T}$ with label k. (Initially we are at the root node with label 1.) Suppose, we have matched $T[j', j-1]$. First repeatedly apply Lemma 7.6 to find the label of the final states reached by the report links. The occurrences of all the patterns ending at the position $j-1$ corresponding to these final states are found using Lemma 7.1. Now, use Lemma 7.5 to find the label k_v of the node (if any) v such that $v = \mathsf{next}(u, T'[j - j' + 1])$, where $T' = \mathsf{order}(T[j', |T|])$. If v exists, then repeat the process with node v and label k_v. If v does not exist, then find the label k_w of the node (if any) w such that $w = \mathsf{failure}(u)$, and repeat the process with node w and label k_w. The total time required is $O(|T|\log m + occ)$[1]. The total space needed to implement Lemmas 7.1, 7.5, and 7.6 is $O(m\log\sigma + d\log\frac{m}{d})$ bits. This concludes the proof of Theorem 7.1.

[1] When we follow a failure link we remove at least one character from the beginning of the text. When we follow a next link we advance one character at the beginning of the text. The number of such operations is at most $2|T|$ and at least one of the operations is executed in each step. On following each report link, we report an occurrence. Each failure and report transition needs $O(1)$ time, and each next transition needs $O(\log m)$ time.

Chapter 8
Open Problems

8.1 Compact Construction

Perhaps the first question that comes to mind is to construct a succinct index for the parameterized pattern matching problem efficiently. For classical full text indexes, designing linear time construction algorithm in compact space has been an active area as seen from the seminal work of Hon, Sadakane, and Sung [HSS03] and the more recent work of Belazzougui [Bel14]. Can we design a (possibly randomized) construction algorithm of a succinct index for the p-pattern matching problem that attains the same bounds of the best-known construction algorithm for p-suffix trees [CH03, Kos95]? More specifically,

Problem 8.1 (Compact Space Construction for Parameterized Text Indexing)**.** *Construct a succinct index for the p-pattern matching problem using $O(n \log \sigma)$-bits of working space.*

8.2 Compressed Indexes for $2D$ Pattern Matching

Consider the two-dimensional pattern matching problem. Let M be square matrix of dimension $n \times n$. Each cell in M contains characters from an alphabet Σ of size σ. The task is to index M such that given any square matrix P having dimensions $m \times m$, where $m \leq n$, we can find all positions (top-left corner) in M where P appears as a sub-matrix.

An $O(n^2 \log n)$-bit index with $O(m^2 \log \sigma + occ)$ query time solution for the above problem is known due to Giancarlo [Gia93]. However, no compressed space indexes are known. Although, the notion of a suffix/prefix is known [Gia93], the crucial understanding of suffix links, and its preserving property is not obvious.

Can we design a succinct (or, compact) index for this problem? More specifically,

Problem 8.2 (Two-Dimensional Pattern Matching)**.** *Can we solve the two-dimensional pattern matching problem using an index that occupies $O(n^2 \log \sigma)$ bits and answers queries in time $O((m^2 + occ) \operatorname{polylog} n)$ time?*

8.3 Streaming Algorithms

A well-known result [CEPP08] says that for any pattern P, at least $\Omega(|P|)$ words are required to answer pattern matching queries in the streaming model, i.e., when the text characters arrive in a stream. Porat and Porat [PP09] showed that randomized algorithms can be used to break the space barrier and presented an algorithm that requires $O(\log |P|)$ words. Their index can report all matches in $O(\log |P|)$ per incoming character, with the probability of reporting a false positive (or, negative) being $\frac{1}{n^3}$. They also showed that k-mismatch version of the problem can be solved in $O(k^2 \operatorname{polylog} |P|)$ time and $O(k^3 \operatorname{polylog} |P|)$ space. This promulgated the study of streaming problems using randomized algorithms. Unfortunately, in [CJPS11], it was shown for a large class of problems that it is not possible to break the $\Omega(|P|)$ words barrier, even using randomized algorithms.

To the best of our knowledge, the only other pattern matching variant that has been solved in the streaming model setting is the p-pattern matching problem [JPS13].

Problem 8.3. *Can we solve the order-isomorphic pattern matching problem in the streaming model using* $O(|\Sigma| \operatorname{polylog}(|P|, |\Sigma|))$ *space? Can we close the (upper bound and lower bound) space-gap in* [JPS13] *for the parameterized pattern matching problem? Can* k*-mismatch version of these problems be solved in the streaming model?*

References

[AAC+03] Amihood Amir, Yonatan Aumann, Richard Cole, Moshe Lewenstein, and Ely Porat. Function matching: Algorithms, applications, and a lower bound. In *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings*, pages 929–942, 2003.

[AC75] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[AFM94] Amihood Amir, Martin Farach, and S. Muthukrishnan. Alphabet dependence in parameterized matching. *Inf. Process. Lett.*, 49(3):111–115, 1994.

[AN09] Amihood Amir and Gonzalo Navarro. Parameterized matching on non-linear structures. *Inf. Process. Lett.*, 109(15):864–867, 2009.

[BA13] Richard Beal and Donald A. Adjeroh. Compressed parameterized pattern matching. In *2013 Data Compression Conference, DCC 2013, Snowbird, UT, USA, March 20-22, 2013*, pages 461–470, 2013.

[Bak93] Brenda S. Baker. A theory of parameterized pattern matching: algorithms and applications. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*, pages 71–80, 1993.

[Bak95] Brenda S. Baker. On finding duplication and near-duplication in large software systems. In *2nd Working Conference on Reverse Engineering, WCRE '95, Toronto, Canada, July 14-16, 1995*, pages 86–95, 1995.

[Bel10] Djamal Belazzougui. Succinct dictionary matching with no slowdown. In *Combinatorial Pattern Matching, 21st Annual Symposium, CPM 2010, New York, NY, USA, June 21-23, 2010. Proceedings*, pages 88–100, 2010.

[Bel14] Djamal Belazzougui. Linear time construction of compressed text indices in compact space. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 148–193, 2014.

[BM77] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[BM98] Brenda S Baker and Udi Manber. Deducing similarities in java sources from bytecodes. In *USENIX Annual Technical Conference*, pages 179–190, 1998.

[BN14] Djamal Belazzougui and Gonzalo Navarro. Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms*, 10(4):23, 2014.

[BW94] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.

[BYdM+98] Ira D. Baxter, Andrew Yahin, Leonardo Mendonça de Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *1998 International Conference on Software Maintenance, ICSM 1998, Bethesda, Maryland, USA, November 16-19, 1998*, pages 368–377, 1998.

[CEPP08] Raphaël Clifford, Klim Efremenko, Benny Porat, and Ely Porat. A black box for online approximate pattern matching. In *Combinatorial Pattern Matching, 19th Annual Symposium, CPM 2008, Pisa, Italy, June 18-20, 2008, Proceedings*, pages 143–151, 2008.

[CGT15] Tamanna Chhabra, Emanuele Giaquinta, and Jorma Tarhio. Filtration algorithms for approximate order-preserving matching. In *String Processing and Information Retrieval - 22nd International Symposium, SPIRE 2015, London, UK, September 1-4, 2015, Proceedings*, pages 177–187, 2015.

[CH03] Richard Cole and Ramesh Hariharan. Faster suffix tree construction with missing suffix links. *SIAM J. Comput.*, 33(1):26–42, 2003.

[CIK+13] Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Marcin Kubica, Alessio Langiu, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Order-preserving incomplete suffix trees and order-preserving indexes. In *String Processing and Information Retrieval - 20th International Symposium, SPIRE 2013, Jerusalem, Israel, October 7-9, 2013, Proceedings*, pages 84–95, 2013.

[CJPS11] Raphaël Clifford, Markus Jalsenius, Ely Porat, and Benjamin Sach. Space lower bounds for online pattern matching. In *Combinatorial Pattern Matching - 22nd Annual Symposium, CPM 2011, Palermo, Italy, June 27-29, 2011. Proceedings*, pages 184–196, 2011.

[CLI] CLoning Analysis and Categorization System (CLICS). `http://www.swag.uwaterloo.ca/clics/`. Accessed: 2015-11-01.

[CNPS15] Sukhyeun Cho, Joong Chae Na, Kunsoo Park, and Jeong Seop Sim. A fast algorithm for order-preserving pattern matching. *Inf. Process. Lett.*, 115(2):397–402, 2015.

[CP09] Hagai Cohen and Ely Porat. Range non-overlapping indexing. In *Algorithms and Computation, 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16-18, 2009. Proceedings*, pages 1044–1053, 2009.

[CT14] Tamanna Chhabra and Jorma Tarhio. Order-preserving matching with filtration. In *Experimental Algorithms - 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29 - July 1, 2014. Proceedings*, pages 307–314, 2014.

[DLDPFG01] Giuseppe Antonio Di Lucca, Massimiliano Di Penta, Anna Rita Fasolino, and Pasquale Granato. Clone analysis in the web era: An approach to identify cloned web pages. In *Seventh IEEE Workshop on Empirical Studies of Software Maintenance*, pages 107–113, 2001.

[dMRS05] Cédric du Mouza, Philippe Rigaux, and Michel Scholl. Efficient evaluation of parameterized pattern queries. In *Proceedings of the 2005 ACM CIKM International Conference on Information and Knowledge Management, Bremen, Germany, October 31 - November 5, 2005*, pages 728–735, 2005.

[dMRS07] Cédric du Mouza, Philippe Rigaux, and Michel Scholl. Parameterized pattern queries. *Data Knowl. Eng.*, 63(2):433–456, 2007.

[FH07] Johannes Fischer and Volker Heun. A new succinct representation of rmq-information and improvements in the enhanced suffix array. In *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies, First International Symposium, ESCAPE 2007, Hangzhou, China, April 7-9, 2007, Revised Selected Papers*, pages 459–470, 2007.

[Fis10] Johannes Fischer. Wee LCP. *Inf. Process. Lett.*, 110(8-9):317–320, 2010.

[FKS84] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, 1984.

[FLMM09] Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *J. ACM*, 57(1), 2009.

[FM00] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12-14 November 2000, Redondo Beach, California, USA*, pages 390–398, 2000.

[FM05] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

[GGG+07] Alexander Golynski, Roberto Grossi, Ankur Gupta, Rajeev Raman, and S. Srinivasa Rao. On the size of succinct indices. In *Algorithms - ESA 2007, 15th Annual European Symposium, Eilat, Israel, October 8-10, 2007, Proceedings*, pages 371–382, 2007.

[GGV03] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA.*, pages 841–850, 2003.

[GHS+16a] Arnab Ganguly, Wing-Kai Hon, Kunihiko Sadakane, Rahul Shah, Sharma V. Thankachan, and Yilin Yang. Space-efficient dictionaries for parameterized

and order-preserving pattern matching. In *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, June 27-29, 2016, Tel Aviv, Israel*, pages 2:1–2:12, 2016.

[GHS16b] Arnab Ganguly, Wing-Kai Hon, and Rahul Shah. A framework for dynamic parameterized dictionary matching. In *15th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2016, June 22-24, 2016, Reykjavik, Iceland*, pages 10:1–10:14, 2016.

[Gia93] Raffaele Giancarlo. The suffix of a square matrix, with applications. In *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas.*, pages 402–411, 1993.

[GM16] Travis Gagie and Giovanni Manzini. Toward a succinct index for order-preserving pattern matching. *CoRR*, abs/1610.02865, 2016.

[GST15] Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. Succinct non-overlapping indexing. In *Combinatorial Pattern Matching - 26th Annual Symposium, CPM 2015, Ischia Island, Italy, June 29 - July 1, 2015, Proceedings*, pages 185–195, 2015.

[GST17] Arnab Ganguly, Rahul Shah, and Sharma V Thankachan. pBWT: Achieving succinct data structures for parameterized pattern matching and related problems. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 397–407. Society for Industrial and Applied Mathematics, 2017.

[GU16] Pawel Gawrychowski and Przemyslaw Uznanski. Order-preserving pattern matching with k mismatches. *Theor. Comput. Sci.*, 638:136–144, 2016.

[Gus97] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997.

[GV00] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, pages 397–406, 2000.

[GV05] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005.

[HJHC10] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-based code clone detection: incremental, distributed, scalable. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–9, Sept 2010.

[HKC+15] Myoungji Han, Munseong Kang, Sukhyeun Cho, Geonmo Gu, Jeong Seop Sim, and Kunsoo Park. Fast multiple order-preserving matching algorithms. In *Combinatorial Algorithms - 26th International Workshop, IWOCA 2015, Verona, Italy, October 5-7, 2015, Revised Selected Papers*, pages 248–259, 2015.

[HLS04] Carmit Hazay, Moshe Lewenstein, and Dina Sokol. Approximate parameterized matching. In *Algorithms - ESA 2004, 12th Annual European Symposium, Bergen, Norway, September 14-17, 2004, Proceedings*, pages 414–425, 2004.

[HLS+08] Wing-Kai Hon, Tak Wah Lam, Rahul Shah, Siu-Lung Tam, and Jeffrey Scott Vitter. Compressed index for dictionary matching. In *2008 Data Compression Conference (DCC 2008), 25-27 March 2008, Snowbird, UT, USA*, pages 23–32, 2008.

[HPST13] Wing-Kai Hon, Manish Patil, Rahul Shah, and Sharma V. Thankachan. Compressed property suffix trees. *Inf. Comput.*, 232:10–18, 2013.

[HSS03] Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Breaking a time-and-space barrier in constructing full-text indices. In *44th Symposium on Foundations of Computer Science (FOCS 2003), 11-14 October 2003, Cambridge, MA, USA, Proceedings*, pages 251–260, 2003.

[HSSY10] W.K. Hon, K. Sadakane, R. Shah, and S.M. Yiu. *Unpublished Manuscript*, 2010.

[IS94] Ramana M. Idury and Alejandro A. Schäffer. Multiple matching of parameterized patterns. In *Combinatorial Pattern Matching, 5th Annual Symposium, CPM 94, Asilomar, California, USA, June 5-8, 1994, Proceedings*, pages 226–239, 1994.

[JPS13] Markus Jalsenius, Benny Porat, and Benjamin Sach. Parameterized matching in the streaming model. In *30th International Symposium on Theoretical Aspects of Computer Science, STACS 2013, February 27 - March 2, 2013, Kiel, Germany*, pages 400–411, 2013.

[KEF+14] Jinil Kim, Peter Eades, Rudolf Fleischer, Seok-Hee Hong, Costas S. Iliopoulos, Kunsoo Park, Simon J. Puglisi, and Takeshi Tokuyama. Order-preserving matching. *Theor. Comput. Sci.*, 525:68–79, 2014.

[KFF06] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *13th Working Conference on Reverse Engineering (WCRE 2006), 23-27 October 2006, Benevento, Italy*, pages 253–262, 2006.

[KJP77] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[KKP03] Dong Kyue Kim, Yoo Ah Kim, and Kunsoo Park. Generalizations of suffix arrays to multi-dimensional matrices. *Theor. Comput. Sci.*, 302(1-3):223–238, 2003.

[Kos95] S. Rao Kosaraju. Faster algorithms for the construction of parameterized suffix trees (preliminary version). In *36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, 23-25 October 1995*, pages 631–637, 1995.

[KR87] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[Kur99] Stefan Kurtz. Reducing the space requirement of suffix trees. *Softw., Pract. Exper.*, 29(13):1149–1171, 1999.

[Lew15] Moshe Lewenstein. Parameterized pattern matching. In *Encyclopedia of Algorithms*. 2015.

[MM93] Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. volume 22, pages 935–948, 1993.

[MNN+14] J. Ian Munro, Gonzalo Navarro, Jesper Sindahl Nielsen, Rahul Shah, and Sharma V. Thankachan. Top- k term-proximity in succinct space. In *Algorithms and Computation - 25th International Symposium, ISAAC 2014, Jeonju, Korea, December 15-17, 2014, Proceedings*, pages 169–180, 2014.

[MP15] Juan Mendivelso and Yoan J. Pinzón. Parameterized matching: Solutions and extensions. In *Proceedings of the Prague Stringology Conference 2015, Prague, Czech Republic, August 24-26, 2015*, pages 118–131, 2015.

[Mun96] J Ian Munro. Tables. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 37–42. Springer Berlin Heidelberg, 1996.

[Nav13] Gonzalo Navarro. Spaces, trees, and colors: The algorithmic landscape of document retrieval on sequences. *ACM Comput. Surv.*, 46(4):52:1–52:47, 2013.

[Nav14] Gonzalo Navarro. Wavelet trees for all. *J. Discrete Algorithms*, 25:2–20, 2014.

[NM07] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1), 2007.

[NS14] Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Trans. Algorithms*, 10(3):16:1–16:39, 2014. *An extended abstract appeared in SODA 2010.*

[NT13] Gonzalo Navarro and Sharma V. Thankachan. Faster top-k document retrieval in optimal space. In *String Processing and Information Retrieval - 20th International Symposium, SPIRE 2013, Jerusalem, Israel, October 7-9, 2013, Proceedings*, pages 255–262, 2013.

[OGK10] Enno Ohlebusch, Simon Gog, and Adrian Kügel. Computing matching statistics and maximal exact matches on compressed full-text indexes. In *String Processing and Information Retrieval - 17th International Symposium, SPIRE 2010, Los Cabos, Mexico, October 11-13, 2010. Proceedings*, pages 347–358, 2010.

[PP09] Benny Porat and Ely Porat. Exact and approximate pattern matching in the streaming model. In *50th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2009, October 25-27, 2009, Atlanta, Georgia, USA*, pages 315–323, 2009.

[RD04] Filip Van Rysselberghe and Serge Demeyer. Evaluating clone detection techniques from a refactoring perspective. In *19th IEEE International Conference on Automated Software Engineering (ASE 2004), 20-25 September 2004, Linz, Austria*, pages 336–339, 2004.

[RNO11] Luís M. S. Russo, Gonzalo Navarro, and Arlindo L. Oliveira. Fully compressed suffix trees. *ACM Transactions on Algorithms*, 7(4):53, 2011.

[RRS07] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4), 2007. *An extended abstract appeared in SODA 2002.*

[Sad02a] Kunihiko Sadakane. Space-efficient data structures for flexible text retrieval systems. In *Algorithms and Computation, 13th International Symposium, ISAAC 2002 Vancouver, BC, Canada, November 21-23, 2002, Proceedings*, pages 14–24, 2002.

[Sad02b] Kunihiko Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA.*, pages 225–232, 2002.

[Sad07] Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007.

[Shi00] Tetsuo Shibuya. Generalization of a suffix tree for RNA structural pattern matching. In *Algorithm Theory - SWAT 2000, 7th Scandinavian Workshop on Algorithm Theory, Bergen, Norway, July 5-7, 2000, Proceedings*, pages 393–406, 2000.

[Shi10] Tetsuo Shibuya. Geometric suffix tree: Indexing protein 3-d structures. *J. ACM*, 57(3):15:1–15:17, 2010.

[Tsu13] Dekel Tsur. Top-k document retrieval in optimal space. *Inf. Process. Lett.*, 113(12):440–443, 2013.

[Ukk95] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[Wei73] Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11, 1973.

Vita

Arnab Ganguly was born in Kolkata, India in 1986. He obtained his Bachelor's degree in Computer Science from Jadavpur University, Kolkata in 2009. During his doctoral studies at the Louisiana State University, Arnab has co-authored 10 papers in peer-reviewed conferences. His research interest falls in the areas of compressed data structures, computational geometry, and graph algorithms. During the final year of his doctoral studies, Arnab was awarded with the Graduate School Dissertation Fellowship.