Relative Compressed Reverse Suffix Array

Muhammed Oguzhan Kulekci
Miami University, Oxford, OH, U.S.A

Mano Prakash Parthasarathi[1]
North Carolina State University, Raleigh, NC, U.S.A

Rahul Shah
Louisiana State University, Baton Rouge, LA, U.S.A

Sharma V. Thankachan
North Carolina State University, Raleigh, NC, U.S.A

Abstract

Suffix trees and suffix arrays are two fundamental data structures in the field of string algorithms. For a string (a.k.a. text or sequence) of length n over an alphabet of size σ, these structures typically require $O(n \log n)$ bits of space. The FM-index provides a compressed representation of the suffix array in $\approx n \log \sigma$ bits, allowing for efficient queries on both the suffix array and its inverse array in near logarithmic time. In certain applications, such as approximate pattern matching (i.e., with wildcards, mismatches, edits), there is a need to access the suffix array of a text, as well as the suffix array of text's reverse. Motivated by this, we explore the possibility of encoding the suffix array of the reversed text in a compact form, assuming the availability of the FM-index for the original text. Our first solution is an $O(n)$-bit (relative) encoding of the suffix array of the reversed text, with the time for decoding an entry being only $O(\log^* n)$ times that of decoding an entry in the text's suffix array using FM-index. We then demonstrate how to reduce the space to $O(n/\kappa)$ bits for a parameter κ, while multiplicative factor in time becomes approximately $O(\kappa \log^* n + \kappa^3)$. We can also support inverse suffix array and longest common extension queries on the reversed text. These results are achieved through some careful and non-trivial application of various succinct data structure techniques.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases String Matching, Text Indexing, Data Structures, Suffix Trees

Digital Object Identifier 10.4230/LIPIcs.STACS.2026.68

Funding *Mano Prakash Parthasarathi*: U.S. National Science Foundation (NSF) award CCF-2315822.
Rahul Shah: U.S. NSF awards CCF-2434261 and CCF-2137057
Sharma V. Thankachan: U.S. NSF awards CCF-2316691 and CCF-2315822.

[1] Corresponding author

1 Introduction

The suffix tree data structure [22, 33, 34] has long been a fundamental tool in string processing, enabling tasks such as pattern matching in time linear in the pattern length. However, their use has been limited in practice due to their high memory usage. Suffix arrays [21] were introduced as a more memory-efficient alternative that can perform most of the tasks as suffix trees. Both suffix trees and suffix arrays of a text of length n still require $O(n \log n)$ bits to store, while the text itself only needs $n \log \sigma$ bits, where σ is the alphabet size. The major breakthrough came in 2000 with the FM-index [7], along with another important development called the compressed suffix array [14]; both of these encode the suffix array in succinct (even compressed) space. Since then, compressed text indexing has continued to be an active area of research, with the FM-index emerging as a foundational component in many bioinformatics tools [18, 20].

In its standard form, the FM-index is essentially the Burrows-Wheeler Transform (BWT) of the text, stored in a data structure that supports *rank* queries (typically using a wavelet tree [13, 24]), along with a sampled subset of suffix array values. The wavelet tree part uses $n \log \sigma + o(n) + O(\sigma \log n)$ bits, and supports rank queries in time $t_{wt} = O(1 + \log \sigma / \log \log n)$. For a parameter τ, the number of sampled suffix array values is n/τ and its storage takes $O((n/\tau) \log n)$ bits. To keep this extra space within $o(n)$ bits, τ is typically chosen as $\omega(\log n)$. With this, the suffix array (SA) and inverse suffix array (ISA) queries can be supported in time $t_{sa} = O(\tau \cdot t_{wt})$. With additional $o(n)$ bits, we can obtain an encoding of the suffix tree itself that supports various queries in time close to $O(t_{sa})$ [31, 30]. Recent results have revealed the possibility of achieving even more space-efficient encodings of suffix arrays/trees when the data is highly repetitive [10, 16, 26, 27].

We investigate the problem of compactly encoding the suffix array of the reverse of a text, assuming access to the text's FM-index in its standard form (i.e., the Burrows–Wheeler Transform represented as a wavelet tree together with a sampled suffix array). While this is a natural question in its own right, solutions to it have important applications in *text indexing for approximate string matching*. A common technique for supporting pattern matching with mismatches or "don't care" characters involves maintaining both the suffix array or tree of the text and that of its reverse, along with a range query data structure built from specific points derived from these structures [2]. For a comprehensive survey of various results based on this technique, we refer the reader to Lewenstein [19]. See also the solutions for contextual pattern matching [1, 25], bidirectional indexes [17], and affix arrays [32] for further applications. Reducing the memory footprint of the data structures associated with the reversed text can significantly reduce the overall space usage in such applications. We highlight that our problem aligns with the broader theme of relative text indexing, such as relative suffix trees [6] and relative FM-indexes [3], where the goal is to encode a data structure over a dataset by leveraging the availability of the same structure over a closely related dataset. The fact that, in our case, one text is essentially the reverse of the other introduces significant structure to the problem.

This problem was introduced by Ganguly et al. [12], who presented an algorithm to decode the suffix array values of the reversed text directly from the text's FM-index, without requiring any additional data structures. However, it was heuristic in nature. Specifically, its query time is proportional to the length of the *shortest unique substring* of the text ending at the value returned, which can be $\Theta(n)$ in the worst case. Therefore, our goal is to support suffix array (and inverse suffix array) queries on the reversed text in time close to that of suffix array queries on the original text. We present our results next.

1.1 Our Results

Let $\mathsf{T}[1\,..\,n]$ denote the text to be indexed, where the characters belong to an integer alphabet of size σ. We can represent the text's Burrows–Wheeler Transform (BWT) as a (multiary) wavelet tree in $n\log\sigma + o(n) + O(\sigma\log n)$ bits of space, supporting basic wavelet tree operations (rank, select, access) in time $t_{wt} = O(1 + \log\sigma/\log\log n)$ and range quantile queries in time $t_{quantile} = O(\log\sigma)$. By maintaining some auxiliary structures of space $O(n/\kappa)$ bits for an integer parameter $\kappa \geq 1$, we can support suffix array (SA) and inverse suffix array (ISA) queries in time $t_{sa} = O(t_{wt}\cdot\kappa\log n)$. We obtain the following results.

1. Our first result is that suffix array queries on the reversed text can be supported in time $t_{pa} = t_{sa}\cdot O(\log^* n)$ with an auxiliary structure of $O(n)$ bits of space.
2. We then generalize the above result by introducing a parameter κ, achieving a space usage of $O(n/\kappa)$ bits and a query time $t_{pa} = t_{sa}\cdot O((\kappa\log^* n + \kappa^3\log\kappa)\cdot\log^2\kappa)$.

We also show how to support inverse suffix array queries on the reversed text within the same space and in time $O(t_{pa}\cdot\log n\cdot\log\log n)$. Our approach is based on a simple method for answering longest common extension queries on the reversed text. As shown in [31, 30], we can use $o(n)$ extra bits to obtain a succinct encoding of the suffix tree of the reversed text that supports various standard operations efficiently.

2 Notation and Preliminary

Throughout this paper, let $\mathsf{T} = \mathsf{T}[1\,..\,n]$ denotes the text to be indexed and Σ denotes the alphabet set. The i-th character of T is denoted by $\mathsf{T}[i]$ and the substring $\mathsf{T}[i]\circ\mathsf{T}[i+1]\circ\cdots\circ\mathsf{T}[j-1]\circ\mathsf{T}[j]$ is denoted by $\mathsf{T}[i\,..\,j]$, where $1\leq i\leq j\leq n$ and $\circ$ denotes concatenation. The string $\mathsf{T}[i\,..\,j]$ is empty if $i > j$. We also use $\mathsf{T}[i\,..\,j)$ for $\mathsf{T}[i\,..\,j-1]$ and $\mathsf{T}(i\,..\,j]$ for $\mathsf{T}[i+1\,..\,j]$. A substring $\mathsf{T}[i\,..\,j]$ is called a suffix if $j = n$ and a prefix if $i = 1$; also $\mathsf{T}[1\,..\,j]$ is a proper prefix if $j\neq n$ and $\mathsf{T}[i\,..\,n]$ is a proper suffix if $i\neq 1$. For any string $S = S[1\,..\,s]$, we denote its reverse $S[s]\circ S[s-1]\circ\cdots\circ S[2]\circ S[1]$ by $\overleftarrow{S}$. Without loss of generality, assume that the characters in our alphabet set Σ corresponds to integers in the range $[0,\sigma)$, where $\sigma = |\Sigma|$. Also, let \$ denote the first character in lexicographic order (equivalently, 0) and $\mathsf{T}[i] = \$$ iff $i = n$, ensuring that no suffix is a prefix of another suffix.

We call a data structure *auxiliary* if it is used to support operations assuming some related structures are available; otherwise, it is considered *standalone.* We assume the standard word RAM model of computation with word size $\Omega(\log n)$ bits.

2.1 Succinct Data Structure Toolkit

2.1.1 Rank/select Queries, Indexible Dictionaries and Wavelet Trees

Let $S[1\,..\,n]\in\Sigma^n$, where $\Sigma = \{0,1,2,\ldots,\sigma-1\}$. We define three basic operations.

- $\mathsf{access}_S(i)$ is $S[i]$.
- $\mathsf{rank}_S(i,\alpha)$, where $\alpha\in\Sigma$, is the number of α's in $S[1\,..\,i]$.
- $\mathsf{select}_S(j,\alpha)$ is the position of j-th occurrence of α in S.

When $\Sigma = \{0,1\}$, we can maintain S in $n + o(n)$ bits and support all three operations in $O(1)$ time [28]. Another representation, called *indexible dictionary* takes $m\log(n/m) + O(m)$ bits, where m is the number of 1's in S [29]. It supports the above operations in $O(\log\log n)$ time. Interestingly, it can support $\mathsf{select}_S(j,1)$ in $O(1)$ time, and also $\mathsf{rank}_S(i,1)$ when $S[i] = 1$ (referred to as *partial rank*) in $O(1)$ time. We will use this result extensively.

For a general alphabet Σ, we have the wavelet tree data structure that uses $n \log \sigma + o(n) + O(\sigma \log n)$ bits of space and supports all three basic operations in $O(\log \sigma)$ time [13]. It can also support *range quantile* queries in $O(\log \sigma)$ time [11], where the goal is to find the k-th smallest element in $S[x\,..\,y]$, given the tuple (x, y, k). Multiary wavelet trees offer further improvements, reducing the time complexity of the three basic operations to $O(1 + \log \sigma / \log \log n)$ [8]; however, the time for range quantile queries remains $O(\log \sigma)$. Also note that the $o(n)$ term in space can be made $O(n/\log^c n)$ for any constant c using efficient bit string representations [28]. We refer to Navarro's survey for further reading on this topic [24]. Moving forward, we use t_{wt} to denote the time complexity of the basic operations, and $t_{quantile}$ to represent the time complexity of a range quantile query.

2.1.2 Range Minimum Query (RMQ)

Let $A[1\,..\,n]$ be an array of numbers. A range minimum query $\text{RMQ}_A(x, y)$ returns the position $k \in [x, y]$, where $A[k]$ is the smallest element in $A[x\,..\,y]$. By maintaining a *standalone* data structure of space $2n + o(n)$ bits, we can support RMQ in $O(1)$ time [9].

2.1.3 Succinct Representation of Ordinal Trees

The topology of an ordinal tree of n nodes can be represented in $2n + o(n)$ bits and support the following operations in $O(1)$ time [15, 23]. We specify a node by its *preorder* rank.

- $\mathsf{parent}(u)$ returns the parent of node u.
- $\text{LCA}(u, v)$ returns the lowest (farthest from the root) common ancestor of nodes u and v.
- $\mathsf{size}(u)$ returns the number of *leaves* in the subtree of node u.
- $\mathsf{depth}(u)$ returns the number of ancestors of node u.
- $\mathsf{level\_ancestor}(u, d)$ returns the ancestor of node u at depth d.
- *range of leaves* $[a, b]$ of any given node u, where a-th (resp., b-th) leaf in the tree represents the leftmost (resp., rightmost) leaf in the subtree of u.

2.2 Suffix Tree, Suffix Array and Longest Common Prefix (LCP) Array

The suffix of T starting at position i is $\mathsf{T}[i\,..\,n]$. The *circular* suffix of T, starting at position i is $\mathsf{T}[i\,..\,n] \circ \mathsf{T}[1\,..\,i) = \mathsf{T}[i] \circ \mathsf{T}[i+1] \circ \cdots \circ \mathsf{T}[n] \circ \mathsf{T}[1] \circ \mathsf{T}[2] \circ \cdots \circ \mathsf{T}[i-1]$.

We define the *suffix tree* of text T as a compacted trie of all circular suffixes of T. This is equivalent to the standard definition (i.e., without the term "circular"), since $\mathsf{T}[i] = \$$ iff $i = n$. It consists of n leaves and less than n internal nodes. The edges are (implicitly) labeled with substrings of T. We use $\text{str}(u)$ to denote the concatenation of edge labeled on the path from root to node u. Also, let $\text{strlen}(u)$ be the length of $\text{str}(u)$ (called string depth). The i-th leftmost leaf in the suffix tree is denoted by ℓ_i. The *suffix array* and the *inverse suffix array* of T are the arrays $\text{SA}[1\,..\,n]$ and $\text{ISA}[1\,..\,n]$, respectively, where $\text{SA}[j] = i$ and $\text{ISA}[i] = j$ iff the j-th smallest circular suffix in lexicographic order is $\mathsf{T}[i\,..\,n] \circ \mathsf{T}[1\,..\,i)$. Equivalently, $\text{str}(\ell_j) = \mathsf{T}[i\,..\,n] \circ \mathsf{T}[1\,..\,i)$.

For a string P, the *locus* node u (if it exists) is the node closest to root such that P is a prefix of $\text{str}(u)$ and the range of leaves of u is called the *suffix range* of P, denoted by $[\mathsf{sp}(P), \mathsf{ep}(P)]$. The set $\{\text{SA}[k] \mid k \in [\mathsf{sp}(P), \mathsf{ep}(P)]\}$ denotes the occurrences of P in T. The *longest common extension*, denoted by $\text{LCE}(i, j)$ is the length of the longest common prefix of the suffixes of T starting positions i and j, which is the same as $\text{strlen}(u)$, where u is the *lowest common ancestor* (LCA) of leaves $\ell_{\text{ISA}[i]}$ and $\ell_{\text{ISA}[j]}$.

The *longest common prefix* array $\mathrm{LCP}[1\,..\,n)$ is an array where $\mathrm{LCP}[i] = \mathrm{LCE}(\mathrm{SA}[i], \mathrm{SA}[i+1])$. Therefore, $\mathrm{LCE}(i,j)$ is the smallest element in $\mathrm{LCP}[x\,..\,y)$, where $x = \min\{\mathrm{ISA}[i], \mathrm{ISA}[j]\}$ and $y = \max\{\mathrm{ISA}[i], \mathrm{ISA}[j]\}$. The *permuted longest common prefix* array $\mathrm{pLCP}[1\,..\,n)$ is defined as $\mathrm{pLCP}[j] = \mathrm{LCP}[\mathrm{ISA}[j]] = \mathrm{LCE}(j, \mathrm{SA}[\mathrm{ISA}[j]+1])$. Note that the argument of $\mathrm{LCP}[\cdot]$ is a leaf position in the suffix tree, whereas that of $\mathrm{pLCP}[\cdot]$ is a position in the text. We will later see that pLCP array can be encoded in $2n + o(n)$ bits. See Table 1 for an example.

i	1	2	3	4	5	6	7	8	9	10	11	12
T[1..12]	m	i	s	s	i	s	s	i	p	p	i	$
SA[1..12]	12	11	8	5	2	1	10	9	7	4	6	3
ISA[1..12]	6	5	12	10	4	11	9	3	8	7	2	1
LCP[1..12]	0	1	1	4	0	0	1	0	2	1	3	-
pLCP[1..12]	0	0	-	1	4	3	2	1	0	1	1	0
BWT[1..12]	i	p	s	s	m	$	p	i	s	s	i	i
PA[1..12]	12	2	11	5	8	1	9	10	3	6	4	7
IPA[1..12]	6	2	9	11	4	10	12	5	7	8	3	1
LCS[1..12]	0	1	1	4	0	0	1	0	2	1	2	-
pLCS[1..12]	0	1	2	2	4	1	-	0	1	0	1	0

Table 1 SA, ISA, LCP, pLCP, BWT, PA, IPA, LCS, and pLCS values for $\mathsf{T}[1\,..\,12] = \mathit{mississippi}\$$

2.3 Prefix Tree, Prefix Array and Longest Common Suffix (LCS) Array

We now introduce the *prefix tree and prefix array* of the text, which are equivalent to the suffix tree and suffix array of the reversed text. These alternative definitions are introduced purely for notational convenience.

The prefix of T ending at position i is $\mathsf{T}[1\,..\,i]$. The *circular* prefix of T ending at i is $\mathsf{T}(i\,..\,n] \circ \mathsf{T}[1\,..\,i]$ and its reverse is $\mathsf{T}[i] \circ \mathsf{T}[i-1] \circ \cdots \circ \mathsf{T}[1] \circ \mathsf{T}[n] \circ \mathsf{T}[n-1] \circ \cdots \circ \mathsf{T}[i+1]$. The *prefix tree* of T is a compacted trie of the *reverse of all circular prefixes* of T. It will have n leaves and less than n internal nodes, and the edges will be labeled as in the suffix tree, but with substrings of $\overleftarrow{\mathsf{T}}$. We shall reuse the notations ℓ_i, $\mathrm{str}(\cdot)$ and $\mathrm{strlen}(\cdot)$ as before when the context is clear. We define *prefix array* $\mathrm{PA}[1\,..\,n]$ and *inverse prefix array* $\mathrm{IPA}[1\,..\,n]$ of T as $\mathrm{PA}[j] = i$ and $\mathrm{IPA}[i] = j$ iff $\mathrm{str}(\ell_j)$ is the "reverse" of the circular prefix of T ending at i. Define $\overleftarrow{\mathrm{LCE}}(i,j)$ as the length of the *longest common suffix* of the prefixes of T ending at positions i and j. Therefore, $\overleftarrow{\mathrm{LCE}}(i,j) = \mathrm{strlen}(u)$, where u is the LCA of leaves $\ell_{\mathrm{IPA}[i]}$ and $\ell_{\mathrm{IPA}[j]}$ in the prefix tree. The *longest common suffix* (LCS) array $\mathrm{LCS}[1\,..\,n)$ is an array where $\mathrm{LCS}[i] = \overleftarrow{\mathrm{LCE}}(\mathrm{PA}[i], \mathrm{PA}[i+1])$ and the *permuted longest common suffix* (pLCS) array is such that $\mathrm{pLCS}[j] = \mathrm{LCS}[\mathrm{IPA}[j]] = \overleftarrow{\mathrm{LCE}}(j, \mathrm{PA}[\mathrm{IPA}[j]+1])$. See Table 1 for an example.

2.4 The FM-index

The FM-index [7] is a compressed representation of the suffix array SA, based on the text's BWT [5]. It is a *self-index*, meaning it does not require access to the original text in order to function. The BWT of T, denoted by $\mathrm{BWT}[1\,..\,n]$ is a permutation of characters in T, where $\mathrm{BWT}[i]$ denotes the last character of i-th lexicographically smallest circular suffix of T. In other words, $\mathrm{BWT}[i] = \mathsf{T}[\mathrm{SA}[i]-1]$ if $\mathrm{SA}[i] \neq 1$ and is \$ otherwise. See Table 1 for an example.

The *Last-to-Front* operation is defined as $\mathrm{LF}[i] = \mathrm{ISA}[\mathrm{SA}[i]-1]$ if $\mathrm{SA}[i] \neq 1$ and is 1 otherwise. Moreover, $\mathrm{LF}[i] = \mathit{Count}(c) + \mathsf{rank}_{\mathrm{BWT}}(i, c)$, where $c = \mathrm{BWT}[i]$ and $\mathit{Count}(c) = |\{j \mid \mathrm{BWT}[j] < c\}|$. By maintaining BWT as a wavelet tree, along with $\mathit{Count}(c)$ for all $c \in \Sigma$, in $O(\sigma \log n)$ bits, we can support $\mathrm{LF}[\cdot]$ operation in $O(t_{wt})$ time.

We next focus on $\mathrm{SA}[\cdot]$ and $\mathrm{ISA}[\cdot]$ operations. Let $\mathrm{LF}^k[i] = \mathrm{LF}[\mathrm{LF}^{k-1}[i]]$ if $k > 1$ and is $\mathrm{LF}[i]$ if $k = 1$, then $\mathrm{SA}[\mathrm{LF}^k[i]] = \mathrm{SA}[i] - k$. For a parameter τ, explicitly store the pairs $(i, \mathrm{SA}[i])$ with $\mathrm{SA}[i]$ being a multiple of τ; space is $O((n/\tau)\log n)$ bits. To decode $\mathrm{SA}[i]$, where $(i, \mathrm{SA}[i])$ is not stored, we compute $\mathrm{LF}[i], \mathrm{LF}^2[i], \mathrm{LF}^3[i], \ldots$ until we find an $\mathrm{LF}^k[i]$, where $(\mathrm{LF}^k[i], \mathrm{SA}[\mathrm{LF}^k[i]])$ is stored, and return $\mathrm{SA}[i] = k + \mathrm{SA}[\mathrm{LF}^k[i]]$. To decode $\mathrm{ISA}[j]$, where $(\mathrm{ISA}[j], j)$ is not stored, we find $j' = \tau\lceil j/\tau \rceil$ and $i' = \mathrm{ISA}[j']$ (note that (i', j') is stored) and return $\mathrm{LF}^k[i']$, where $k = j' - j$. In both cases, the sampling scheme guarantees $k \leq \tau$ and the resulting time complexity, denoted by t_{sa}, is $k \cdot t_{wt} = O(\tau \cdot t_{wt})$.

We can also extract any substring $\mathsf{T}[j\,..\,j+r)$ in time $O(t_{sa} + r \cdot t_{wt})$ as follows: find $x = \mathrm{ISA}[j+r]$, $\mathrm{LF}[x], \mathrm{LF}^2[x], \mathrm{LF}^3[x], \ldots, \mathrm{LF}^{r-1}[x]$ successively and output the string $\mathrm{BWT}[\mathrm{LF}^{r-1}[x]] \circ \ldots \mathrm{BWT}[\mathrm{LF}[x]] \circ \mathrm{BWT}[x]$.

▶ **Lemma 1.** *Suppose the text's* BWT *is available as a wavelet tree supporting basic operations (i.e.,* $\mathsf{rank}, \mathsf{select}$ *and* access*) in t_{wt} time, then with $O((n/\tau)\log n)$ extra bits, we can support* SA / ISA *queries in time $t_{sa} = O(\tau \cdot t_{wt})$ and extract any substring $\mathsf{T}[j\,..\,j+r)$ in time $O(t_{sa} + r \cdot t_{wt})$. Here, τ is a parameter.*

2.5 Longest Common Extension (LCE) and Related Queries

The following two results, along with the succinct representation of the suffix tree topology as in Section 2.1.3, gives the compressed representation of the suffix tree. We present sketches of the proofs for these known results, since we will be adapting them in subsequent sections.

▶ **Lemma 2** (Sadakane [31])**.** *The* pLCP *array can be encoded in $2n + o(n)$ bits, such that* $\mathrm{pLCP}[j]$ *for any position j in the text can be retrieved in $O(1)$ time.*

Proof. Key observation is that $\mathrm{pLCP}[j+1] \geq \mathrm{pLCP}[j] - 1$ for all j. To prove this, note that it is trivially true when $\mathrm{pLCP}[j] = 0$. Otherwise, let $\mathrm{SA}[\mathrm{ISA}[j]+1] = j'$. Then $\mathrm{ISA}[j+1] < \mathrm{ISA}[j'+1]$, therefore $\mathrm{pLCP}[j+1]$ is atleast $\mathrm{LCE}(j+1, j'+1) = \mathrm{pLCP}[j] - 1$. By rearranging the inequality and by adding j on both sides, we get $j + \mathrm{pLCP}[j] \leq (j+1) + \mathrm{pLCP}[j+1]$. This means, the function $f(j) = j + \mathrm{pLCP}[j]$ is non-decreasing. Moreover $f(j) \in [1, n]$, because $\mathrm{pLCP}[n] = 0$ and hence $f(n) = n$. Therefore, we can store the function $f(\cdot)$ as a bit string $B = 0^{f(1)}10^{f(2)-f(1)}10^{f(3)-f(2)}1\ldots$ with $O(1)$ time rank/select support in $2n + o(n)$ bits. To decode $\mathrm{pLCP}[j]$, we first find $p = \mathsf{select}_B(j, 1)$, the position of j-th 1 in B in $O(1)$ time. Then we have $f(j) = p - j$, the number of 0's until p, and $\mathrm{pLCP}[j] = f(j) - j$. ◀

▶ **Lemma 3** (LCE Queries)**.** *By maintaining an auxiliary structure of space $O(n)$ bits along with the FM-index, we can answer* $\mathrm{LCE}(\cdot,\cdot)$ *queries in $O(t_{sa})$ time.*

Proof. Maintain the encoding of the pLCP array described in Lemma 2 and an RMQ data structure over the LCP array, but without explicitly storing the LCP array itself. To compute $\mathrm{LCE}(i, j)$, find $i' = \mathrm{ISA}[i]$ and $j' = \mathrm{ISA}[j]$. Without loss of generality, assume that $i' < j'$. Then find $k' \in [i', j')$, where $\mathrm{LCP}[k']$ is the smallest in $\mathrm{LCP}[i'\,..\,j')$ using an RMQ in $O(1)$ time. Then compute $k = \mathrm{SA}[k']$ and return $\mathrm{pLCP}[k]$. Total time is $O(t_{sa})$. ◀

We next present an additional result that will be used later.

▶ **Lemma 4** ($\overleftarrow{\text{LCE}}$ Queries). *Let D be an integer parameter. By maintaining an auxiliary structure of space $O((n/D)\log n)$ bits, we can compute $\overleftarrow{\text{LCE}}(i,j)$ for any given i,j via $O(\log D)$ number of* LCE *queries.*

Proof. For any given text of length n, there exists a *standalone* data structure of space $O(n/D)$ words, equivalently $O((n/D)\log n)$ bits, that in constant time either correctly computes $\text{LCE}(i,j)$ or determines that $\text{LCE}(i,j) \leq D^2$ for any given i,j [4]. We maintain this data structure over the reversed text, which allows us to correctly compute $\overleftarrow{\text{LCE}}(i,j)$ or to determine that $\overleftarrow{\text{LCE}}(i,j) \leq D^2$. When $\overleftarrow{\text{LCE}}(i,j) \leq D^2$, we exploit the following observation: $\overleftarrow{\text{LCE}}(i,j) \geq t$ if and only if $\text{LCE}(i-t+1, j-t+1) \geq t$, and binary search on $t \in [0, D^2]$. This will give us $\overleftarrow{\text{LCE}}(i,j)$ in $O(\log D)$ number of LCE queries. ◀

3 Relative Encoding of PA in $O(n)$ Extra bits

The *prefix tree/array* of T corresponds to the *suffix tree/array* of $\overleftarrow{T}$. Specifically, the i-th entry in the suffix array of $\overleftarrow{\mathsf{T}}$ is $(n - \text{PA}[i] + 1)$. Therefore, it suffices to design an encoding of the prefix array.

3.1 Basic Components

Let $\pi \neq 0$ be an integer parameter (positive or negative). We now introduce arrays LCP_π and pLCP_π, which are generalizations of LCP array and pLCP array as follows:

$$\text{LCP}_\pi[i] = \text{LCE}(\text{SA}[i], \text{SA}[i+\pi]), \text{ where } i+\pi \leq n$$

$$\text{pLCP}_\pi[j] = \text{LCP}_\pi[\text{ISA}[j]] = \text{LCE}(j, \text{SA}[\text{ISA}[j]+\pi]), \text{ where } \text{ISA}[j]+\pi \leq n$$

The encoding technique by Sadakane (see Lemma 2) extends to the pLCP_π array as follows.

▶ **Lemma 5.** *Let $\pi \neq 0$ be an integer parameter (positive or negative). We can encode pLCP_π in $O(n)$ bits, and return $\text{pLCP}_\pi[j]$ for any text position j in $O(1)$ time.*

Proof. We first prove that $\text{pLCP}_\pi[j+1] \geq \text{pLCP}_\pi[j] - 1$ for all j. Clearly, the inequality holds when $\text{pLCP}_\pi[j] = 0$. Otherwise, let $\text{SA}[\text{ISA}[j]+\pi] = j'$ and $\text{SA}[\text{ISA}[j+1]+\pi] = j''$. Then we can see that $\text{ISA}[j+1] < \text{ISA}[j''] \leq \text{ISA}[j'+1]$ if π is positive and $\text{ISA}[j+1] > \text{ISA}[j''] \geq \text{ISA}[j'+1]$ if π is negative. In both cases, $\text{pLCP}_\pi[j+1] = \text{LCE}(j+1, j'') \geq \text{LCE}(j+1, j'+1) \geq \text{LCE}(j,j') - 1 = \text{pLCP}_\pi[j] - 1$. By rearranging the inequality as in the proof of Lemma 2, we obtain a non-decreasing function $f_\pi(j) = j + \text{pLCP}_\pi[j] \in \mathcal{O}(n)$. We store f_π as bit string $B = 0^{f_\pi(1)}10^{f_\pi(2)-f_\pi(1)}10^{f_\pi(3)-f_\pi(2)}1\ldots$ with $O(1)$ time rank/select support in $O(n)$ bits. Then $\text{pLCP}_\pi[j]$ is given by $f_\pi(j) - j = (\mathsf{select}_B(j,1) - j) - j$. ◀

▶ **Lemma 6** (Sampled pLCP_π). *Let $\pi \neq 0$ be an integer parameter (positive or negative) and $S \subseteq [n]$ be a set of size s. There exists a standalone structure of space $O(s\log(n/s))$ bits that returns $\text{pLCP}_\pi[j]$ for any given $j \in S$ in $O(1)$ time.*

Proof. We modify the proof in Lemma 5 as follows. Define a bit string $C[1\,..\,n]$, where $C[j] = 1$ iff $j \in S$. Define another bit string $B = 0^{f_\pi(s_1)}10^{f_\pi(s_2)-f_\pi(s_1)}10^{f_\pi(s_3)-f_\pi(s_2)}1\ldots$, where $s_i = \mathsf{select}_C(i,1)$. Maintain C and B as indexible dictionaries in $O(s\log(n/s))$ bits with efficient rank/select support. Given any $j \in S$, we find $p = \mathsf{rank}_C(j,1)$ in $O(1)$ time using a partial-rank query. Then find $f_\pi(j) = \mathsf{select}_B(p,1) - p$ and report $\text{pLCP}_\pi[j] = f_\pi(j) - j$. ◀

The *Longest common suffix* (LCS) array and the *permuted longest common suffix* (pLCS) array can also be generalized as follows.

$$\mathrm{LCS}_\pi[i] = \overleftarrow{\mathrm{LCE}}(\mathrm{PA}[i], \mathrm{PA}[i+\pi]), \text{ where } i+\pi \le n$$

$$\mathrm{pLCS}_\pi[j] = \mathrm{LCS}_\pi[\mathrm{IPA}[j]] = \overleftarrow{\mathrm{LCE}}(j, \mathrm{PA}[\mathrm{IPA}[j]+\pi]), \text{ where } \mathrm{IPA}[j]+\pi \le n$$

Since LCS_π and pLCS_π are equivalent to LCP_π and pLCP_π of the reverse of T, the following result is an immediate corollary of Lemma 6.

▶ **Corollary 7** (Sampled pLCS_π)**.** *Let $\pi \neq 0$ be an integer parameter (positive or negative) and $S \subseteq [n]$ be a set of size s. There exists a standalone structure of space $O(s \log(n/s))$ bits that returns $\mathrm{pLCS}_\pi[j]$ for any given $j \in S$ in $O(1)$ time.*

3.2 The Data Structure

Assuming the availability of an FM-index of T in its standard from as described in Section 2.4, we now present components in the relative encoding of the prefix array PA of T. For $h \in [1, \log^* n]$, define Δ_h as $(\log^{(h)} n)^2$ round to the integer which is the next power of 2. i.e., $\Delta_h = 2^{\lceil 2\log^{(h+1)} n \rceil}$. Here $\log^{(h)} n = \log n$ if $h = 1$ and is $\log(\log^{(h-1)} n)$ if $h > 1$, and $\log^* n$ is the smallest h where $\log^{(h)} n \le 1$. Therefore, $\Delta_{\log^* n} = 1$. Also let $\Delta_0 = n+1$. We denote the predecessor and successor of i that are multiples of Δ_h by $i'_h = \Delta_h \cdot \lfloor i/\Delta_h \rfloor$ and $i''_h = \Delta_h \cdot \lceil i/\Delta_h \rceil$, respectively. Therefore, when Δ_h divides i, we have $i'_h = i''_h = i$.

3.2.1 Technical Overview

If i is divisible by Δ_1, we explicitly store $\mathrm{PA}[i]$ using $O(\log n)$ bits. If i is divisible by Δ_2, but not by Δ_1, we encode $\mathrm{PA}[i]$ relative to either $\mathrm{PA}[i'_1]$ or $\mathrm{PA}[i''_1]$ (depending on the case, to be described) using $\log \Delta_1$ bits. Similarly, if i is divisible by Δ_3, but not by Δ_2, we encode $\mathrm{PA}[i]$ relative to either $\mathrm{PA}[i'_2]$ or $\mathrm{PA}[i''_2]$ using $\log \Delta_2$ bits. More generally, we encode $\mathrm{PA}[i]$ relative to either $\mathrm{PA}[i'_{f-1}]$ or $\mathrm{PA}[i''_{f-1}]$ using $\log \Delta_{f-1}$ bits, where $f \le \log^* n$ is the smallest integer such that i is divisible by Δ_f. The function Δ_h is chosen carefully such that the total space usage is bounded by a sum that converges to $O(n)$ bits. The algorithm for decoding $\mathrm{PA}[i]$ works as follows: for $h = 1, 2, \ldots,$ decode either i'_h or i''_h (depending on the case) and terminate at $h = f$ where $i'_f = i''_f = i$, thereby yielding $\mathrm{PA}[i]$. We present the details next.

3.2.2 Components of the Data Structure

For each $h \in [1, \log^* n]$, maintain arrays $\mathrm{PA}_h[1, n/\Delta_h]$ as described below.

- If i is divisible by Δ_1, we store $\mathrm{PA}[i]$ explicitly in PA_1 as $\mathrm{PA}_1[i/\Delta_1] = \mathrm{PA}[i]$.
- If i is not divisible by Δ_1, we find the smallest integer f where i is divisible by Δ_f; such an $f \in [2, \log^* n]$ exists since $\Delta_{\log^* n} = 1$. Then store a value θ_i in PA_f as $\mathrm{PA}_f[i/\Delta_f] = \theta_i$. The value θ_i is determined by certain nodes in the prefix tree, which in turn are based on i'_{f-1} and i''_{f-1} as illustrated in Figure 1. Let $u' = \mathrm{LCA}(\ell_{i'_{f-1}}, \ell_i)$ and $u'' = \mathrm{LCA}(\ell_i, \ell_{i''_{f-1}})$, and u be the lowest node among u' and u''. Also let $Q = \mathrm{str}(u)$ and node w be the child of u on the path to ℓ_i and β be the leading character on the edge from u to w. This means that a substring of T matching $\beta \circ \overleftarrow{Q}$ ends at position $\mathrm{PA}[i]$, and therefore starts at $\mathrm{PA}[i] - |Q|$. Consequently, $\mathrm{SA}[\mathsf{sp}(\beta \circ \overleftarrow{Q}) \,..\, \mathsf{ep}(\beta \circ \overleftarrow{Q})]$ must contain the entry $\mathrm{PA}[i] - |Q|$. Then θ_i is defined as the offset within the suffix range of $\beta \circ \overleftarrow{Q}$, Specifically,

$$\mathrm{SA}[\mathsf{sp}(\beta \circ \overleftarrow{Q}) + \theta_i] = \mathrm{PA}[i] - |Q|.$$

Finally, $0 \leq \theta_i \leq \mathsf{ep}(\beta \circ \overleftarrow{Q}) - \mathsf{sp}(\beta \circ \overleftarrow{Q}) + 1 = \mathsf{size}(w) < \Delta_{f-1}$; therefore, we can store θ_i using $\log \Delta_{f-1} = O(\log^{(f)} n)$ bits.

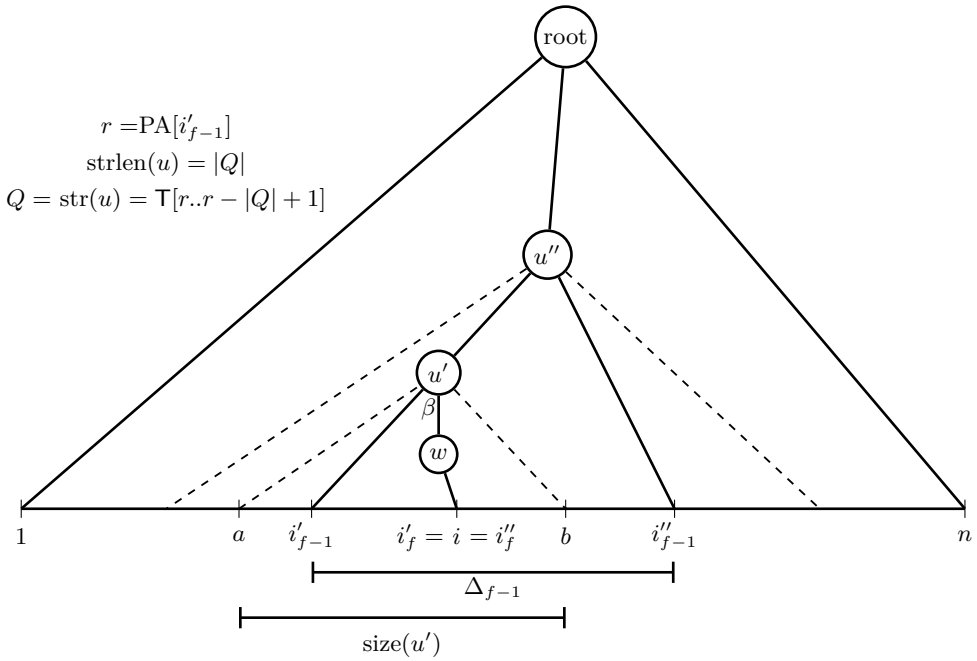

Figure 1 Here $u = u'$. The dashed lines denote the range leaves under the respective nodes.

While decoding PA[i], the answer is readily available from PA_1 if i is a multiple of Δ_1. Otherwise, we utilize the relationship $\mathrm{PA}[i] = \mathrm{SA}[\mathsf{sp}(\beta \circ \overleftarrow{Q}) + \theta_i] + |Q|$, where θ_i is retrieved from PA_f; however, $|Q|$ and $\mathsf{sp}(\beta \circ \overleftarrow{Q})$ are not explicitly stored. To find them, we maintain the following auxiliary structures.

- The succinct encodings of the topologies of both the suffix tree and the prefix tree.
- For each $h \in [1, \log^* n]$, we maintain a collection of structures as in Corollary 7. Precisely, for a fixed h, we assign $S = \{j \mid \mathrm{ISA}[j] \text{ is divisible by } \Delta_{h-1}\}$ and make a separate structure for each *non-zero* value of $\pi \in \{-\Delta_{h-1}, -\Delta_{h-1} + \Delta_h, \ldots, \Delta_{h-1} - \Delta_h, \Delta_{h-1}\}$. Note that for a fixed h, $|S| = n/\Delta_{h-1}$ and the number of structures is $2\Delta_{h-1}/\Delta_h$.

This completes the description of our data structure.

3.2.3 Space Complexity Analysis

The length of PA_h is n/Δ_h and each entry in PA_h is encoded in $\log \Delta_{h-1} = O(\log^{(h)} n)$ bits. Therefore, space is $n/\Delta_h \cdot \log \Delta_{h-1} = O(n/\log^{(h)} n)$ bits for a fixed h and $\sum_{h=1}^{\log^* n} \frac{n}{\log^{(h)} n} = O(n)$ bits for all $h \in [1, \log^* n]$ combined. The tree topologies takes $O(n)$ bits. The space (in bits) required by all versions of the structures in Corollary 7 can also be bounded by $O(n)$, as shown below, resulting in an overall space complexity of $O(n)$ bits.

$$\sum_{h=1}^{\log^* n} \frac{2\Delta_{h-1}}{\Delta_h} \cdot \frac{n}{\Delta_{h-1}} \log \Delta_{h-1} = 2 \sum_{h=1}^{\log^* n} \frac{n}{\log^{(h)} n} = O(n).$$

3.3 Algorithm for Decoding $\mathrm{PA}[i]$

At first, compute $f \in [1, \log^* n]$, the smallest integer such that Δ_f divides i. Then, identify nodes $u' = \mathrm{LCA}(\ell_{i'_{f-1}}, \ell_i)$ and $u'' = \mathrm{LCA}(\ell_i, \ell_{i''_{f-1}})$ in the prefix tree, and let u be the lowest node among them. This step takes only $O(1)$ time using tree encodings. Let $Q = \mathrm{str}(u)$ and β be the leading character on the edge from u to its child w on the path to ℓ_i. Our next goal is to compute $|Q|$ and the suffix range $[\mathsf{sp}(\beta \circ \overleftarrow{Q}), \mathsf{ep}(\beta \circ \overleftarrow{Q})]$ of $\beta \circ \overleftarrow{Q}$, then return $\mathrm{PA}[i] = \mathrm{SA}[\mathsf{sp}(\beta \circ \overleftarrow{Q}) + \theta_i] + |Q|$. There are two cases.

1. **Case $u = u'$ (i.e., $u = u' = u''$ or $u = u' \neq u''$)**: In this case, $\mathrm{PA}[i]$ is encoded relative to $\mathrm{PA}[i'_{f-1}]$. We start with an assumption that $r = \mathrm{PA}[i'_{f-1}]$ has already been computed. Observe that $|Q| = \mathrm{strlen}(u) = \overleftarrow{\mathrm{LCE}}(\mathrm{PA}[i'_{f-1}], \mathrm{PA}[i]) = \mathrm{pLCS}_\pi[r]$, where $\pi = i - i'_{f-1}$. Note also that $r \in S = \{j \mid \mathrm{ISA}[j] \text{ is divisible by } \Delta_{f-1}\}$ and $\pi \in \{\Delta_h, 2\Delta_h, \ldots, \Delta_{h-1}\}$. Recall that we already maintain the data structure of Corollary 7 for this specific choice of S and π; querying it yields $\mathrm{pLCS}_\pi[r]$, and thus $|Q|$, in $O(1)$ time. We then infer $\overleftarrow{Q} = \mathsf{T}[r - |Q| + 1 \,..\, r]$ and invoke the steps below.

   a. **Find the suffix range $[\mathsf{sp}(\overleftarrow{Q}), \mathsf{ep}(\overleftarrow{Q})]$ of $\overleftarrow{Q}$:** Find the leaf ℓ_g in the suffix tree, where $g = \mathrm{ISA}[r - |Q| + 1]$ (this leaf corresponds to the suffix $\mathsf{T}[r - |Q| + 1 \,..]$), then find its ancestor v such that $\mathsf{size}(v) = \mathsf{size}(u)$, and return the range of leaves of v as the answer. Note that v exists and it can be computed via binary search, specifically $O(\log n)$ number of $\mathsf{level\_ancestor}$ queries. The first step requires an ISA query (in time t_{sa}), the second step takes $O(\log n)$ time and the third step takes $O(1)$ time.
   b. **Find β:** Let $[a, b]$ be the range of leaves of u (in the prefix tree) and let $k = i - a + 1$. Then, ℓ_i will be the k-th leftmost leaf in the subtree of u, which implies β will be the k-th smallest character in $\mathrm{BWT}[\mathsf{sp}(\overleftarrow{Q}) \,..\, \mathsf{ep}(\overleftarrow{Q})]$. So, perform a range quantile query in time $t_{quantile}$ and find β.
   c. **Find the suffix range $[\mathsf{sp}(\beta \circ \overleftarrow{Q}), \mathsf{ep}(\beta \circ \overleftarrow{Q})]$ of $\beta \circ \overleftarrow{Q}$:** Let x (resp., y) be the first (resp., last) occurrence of β in $\mathrm{BWT}[\mathsf{sp}(\overleftarrow{Q}) \,..\, \mathsf{ep}(\overleftarrow{Q})]$. Then, $[\mathrm{LF}[x], \mathrm{LF}[y]]$ will be the suffix range of $\beta \circ \overleftarrow{Q}$ which can be answered in $O(t_{wt})$ time.
   d. Compute $\mathrm{PA}[i] = \mathrm{SA}[\mathsf{sp}(\beta \circ \overleftarrow{Q}) + \theta_i] + |Q|$ and return it. Note that θ_i is stored explicitly, and hence this step takes $\mathcal{O}(t_{sa})$ time.

   Total time is $O(t_{sa} + \log n + t_{quantile} + t_{wt})$.

2. **Case $u \neq u'$ (i.e, $u = u'' \neq u'$):** In this case, $\mathrm{PA}[i]$ is encoded relative to $\mathrm{PA}[i''_{f-1}]$. As before, we assume that $r = \mathrm{PA}[i''_{f-1}]$ has already been computed. Observe that $|Q| = \mathrm{strlen}(u) = \overleftarrow{\mathrm{LCE}}(\mathrm{PA}[i''_{f-1}], \mathrm{PA}[i]) = \mathrm{pLCS}_\pi[r]$, where $\pi = i - i''_{f-1}$. Note also that $r \in S = \{j \mid \mathrm{ISA}[j] \text{ is divisible by } \Delta_{f-1}\}$ and $\pi \in \{-\Delta_h, -2\Delta_h, \ldots, -\Delta_{h-1}\}$. Recall that we already maintain the structure of Corollary 7 for this specific choice of S and π; querying it yields $|Q| = \mathrm{pLCS}_\pi[r]$ in $O(1)$ time. We then infer $\overleftarrow{Q} = \mathsf{T}[r - |Q| + 1 \,..\, r]$, invoke the same steps as before and obtain $\mathrm{PA}[i]$ in $O(t_{sa} + \log n + t_{quantile} + t_{wt})$ time.

In summary, decoding $\mathrm{PA}[i]$ reduces to decoding either $\mathrm{PA}[i'_{f-1}]$ or $\mathrm{PA}[i''_{f-1}]$ (depending on the case) in $O(t_{sa} + \log n + t_{quantile} + t_{wt})$ time, which recursively reduces to decoding $\mathrm{PA}[i'_{f-2}]$ or $\mathrm{PA}[i''_{f-2}]$, and so forth, until reaching $\mathrm{PA}[i'_1]$ or $\mathrm{PA}[i''_1]$, which are explicitly stored. The overall time is $O((t_{sa} + \log n + t_{quantile} + t_{wt}) \cdot f) \subseteq O((\tau \cdot t_{wt} + \log n + t_{quantile}) \log^* n)$.

▶ **Theorem 8.** *Let the Burrows-Wheeler Transform (BWT) of a text* $\mathsf{T}[1..n]$ *be maintained in a data structure that supports basic queries (*rank, select, *and* access*) in* t_{wt} *time, and range quantile queries in* $t_{quantile}$ *time. Then, for any parameter* $\tau \geq 1$*, we can maintain a sampled subset of* n/τ *suffix array values using* $O((n/\tau)\log n)$ *bits of additional space, and support suffix array (SA), inverse suffix array (ISA), and longest common extension (LCE) queries in time* $t_{sa} = O(\tau \cdot t_{wt})$*.*

By maintaining an auxiliary structure of space $O(n)$ *bits, we can support prefix array (PA) queries in time* $t_{pa} = O((\tau \cdot t_{wt} + t_{quantile} + \log n) \cdot \log^* n)$*.*

By employing the multiary wavelet tree, where $t_{wt} = O(1+\log\sigma/\log\log n)$ and $t_{quantile} = O(\log\sigma)$ [8], and setting $\tau = \log n$, we obtain the following result.

▶ **Corollary 9.** *For any text* $\mathsf{T}[1..n]$ *over an alphabet of size* σ*, we can represent its Burrows–Wheeler Transform using a (multiary) wavelet tree in* $n\log\sigma + o(n) + O(\sigma\log n)$ *bits of space, supporting basic wavelet tree operations (rank, select, access) in time* $O(1 + \log\sigma/\log\log n)$ *and range quantile queries in time* $O(\log\sigma)$*. By maintaining some auxiliary structures of space* $O(n)$ *bits, we can support suffix array (SA), inverse suffix array (ISA), and longest common extension (LCE) queries in time* $t_{sa} = O((1 + \log\sigma/\log\log n)\log n)$*, and prefix array (PA) queries in time* $t_{pa} = O(t_{sa} \cdot \log^* n)$*.*

4 Relative Encoding of PA in Sublinear Space

In this section, we present an $O((n/\delta)\log\delta)$-bit (relative) encoding of PA, where $\delta > 0$ is an integer parameter. Without loss of generality, we assume that δ is a power of 2.

4.1 Basic Components

We now present several useful results.

▶ **Lemma 10** (Approximate pLCP_π in small space)**.** *Let* $\pi \neq 0$ *be an integer parameter (positive or negative). There exists a standalone structure of space* $O((n/\delta)\log\delta)$ *bits that returns* $\text{pLCP}^{\approx}_{\pi}[j]$ *in* $O(\log\delta)$ *time, where* $\text{pLCP}_\pi[j] - \delta \leq \text{pLCP}^{\approx}_{\pi}[j] \leq \text{pLCP}_\pi[j]$*.*

Proof. We obtain this result by modifying Lemma 6. The idea is to select a set $S \subseteq \{1, 2, 3, \ldots, n\}$ of sampled positions and store $f_\pi(j)$ values of only those $j \in S$. The set S is constructed as follows. Initially $S = \{1\}$. Then for $j = 2$ to n, we include j in S iff j is a multiple of δ or $f_\pi(j) - f_\pi(j') > \delta$, where $j' < j$ is the largest value in S currently. The size of S is $\Theta(n/\delta)$ since $f_\pi(\cdot) \leq 2n$ and is non-decreasing. For a query j, we return $f_\pi(j') - j$ as $\text{pLCP}^{\approx}_{\pi}[j]$, where j' is the predecessor of j in S. The correctness follows from that fact that $f_\pi(j') \leq f_\pi(j) = \text{pLCP}_\pi[j] + j \leq f_\pi(j') + \delta$ and $j' \leq j \leq j' + \delta$, which implies $\text{pLCP}_\pi[j] - \delta \leq f_\pi(j') - j \leq \text{pLCP}_\pi[j]$.

We now present the implementation details. Define a bit string C corresponding to S, where $C[j] = 1$ iff $j \in S$. Define another bit string $B = 0^{f_\pi(s_1)}10^{f_\pi(s_2)-f_\pi(s_1)}10^{f_\pi(s_3)-f_\pi(s_2)}1\ldots$, where $s_i = \mathsf{select}_C(i, 1)$. We maintain both C and B as indexible dictionaries with efficient rank/select support. The space required is $O((n/\delta)\log\delta)$ bits since both B and C are of size $O(n)$ with $\Theta(n/\delta)$ number of 1's. To recover an approximate value of $\text{pLCP}_\pi[j]$, we first find $k = \delta\lfloor j/\delta \rfloor \in S$ and $r = \mathsf{rank}_C(k, 1)$ using a partial rank query in $O(1)$ time. Then j', the predecessor of j in S is $\mathsf{select}_C(t, 1)$ for the $t \in [r, r+\delta]$, where $\mathsf{select}_C(t, 1) \leq j < \mathsf{select}_C(t+1, 1)$, which can be computed in $O(\log\delta)$ time via binary search. Finally, get $f_\pi(j') = \mathsf{select}_B(t, 1) - t$ and return $\text{pLCP}^{\approx}_{\pi}[j] = f_\pi(j') - j$. ◀

▶ **Lemma 11** (Approximate pLCS_π in small space). *Let $\pi \neq 0$ be an integer parameter (positive or negative). There exists a standalone structure of space $O((n/\delta)\log\delta)$ bits that returns $\text{pLCS}^{\approx}_{\pi}[j]$ in $O(\log\delta)$ time, where $\text{pLCS}_\pi[j] - \delta \leq \text{pLCS}^{\approx}_{\pi}[j] \leq \text{pLCS}_\pi[j]$.*

Proof. Follows from Lemma 10, because pLCS_π is equivalent to pLCP_π of text's reverse. ◀

▶ **Lemma 12** (Exact pLCP in small space). *By maintaining an $O((n/\delta)\log\delta)$-bit auxiliary structure with the FM-index, we can return $\text{pLCP}[j]$ for any j in $O(\delta \cdot t_{wt} + t_{sa})$ time.*

Proof. Maintain the structure in Lemma 10 for $\pi = 1$. Let $j' = \text{SA}[\text{ISA}[j] + 1]$, then $\text{pLCP}[j] = \text{LCE}(j, j')$. Find $h = \text{pLCP}^{\approx}_{\pi}[j]$ first. Now we know $\text{LCE}(j, j') = h + \text{LCE}(j + h, j'+h)$ and $\text{LCE}(j+h, j'+h) \leq \delta$. Therefore, simply extract the substrings $\mathsf{T}[j+h\,..\,j+h+\delta)$ and $\mathsf{T}[j'+h\,..\,j'+h+\delta)$ and compute $\text{LCE}(j+h, j'+h)$ naively. ◀

▶ **Lemma 13** (Exact $\text{LCE}(\cdot,\cdot)$ in small space). *By maintaining an $O((n/\delta)\log\delta)$-bit auxiliary structure with FM-index, we can return $\text{LCE}(i, j)$ in time $t_{\text{LCE}} = O((\delta \cdot t_{wt} + t_{sa}) \cdot \delta)$ time.*

Proof. We divide the LCP array into blocks of size δ. i.e., $\text{LCP}[1\,..\,\delta], \text{LCP}[\delta{+}1\,..\,2\delta], \text{LCP}[2\delta{+}1\,..\,3\delta]$, etc. Also let $\text{LCP}'[x]$ be the smallest element in x-th block. Maintain a range minimum query structure over LCP' array (space required is $2n/\delta + o(n/\delta)$ bits). Besides this, we will use the result in Lemma 12. When a query (i, j) comes, we compute $\text{LCE}(i, j)$ as follows. Let $i' = \text{ISA}[i]$ and $j' = \text{ISA}[j]$. Without loss of generality, lets assume that $i' < j'$. To efficiently find our answer $\min_{k\in[i',j')} \text{LCP}[k]$, we invoke the following procedure.

- If $j' - i' < 3\delta$, we extract $\text{LCP}[k] = \text{pLCP}[\text{SA}[k]]\ \forall\ k \in [i', j')$ and return the minimum.
- Otherwise, we extract $\text{LCP}[k] = \text{pLCP}[\text{SA}[k]]\ \forall\ k \in [i', i'+\delta) \cup (\delta\cdot k', \delta\cdot k'+\delta] \cup (j'-\delta, j')$, where $k' = \text{RMQ}_{\text{LCP}'}(1 + \lceil i'/\delta\rceil, \lfloor j'/\delta\rfloor)$ and return the minimum.

In both cases, we are required to make two ISA queries, $O(\delta)$ number of SA queries, followed by $O(\delta)$ number of $\text{pLCP}[\cdot]$ queries, which makes the total time $O((\delta \cdot t_{wt} + t_{sa})\delta)$. ◀

4.2 The Data Structure

4.2.1 Technical Overview

We categorize a query $\text{PA}[i]$ as Type-A if i is divisible by δ (recall that δ is a power of 2), and as Type-B otherwise. These two types are handled separately. For Type-A queries, we modify our $O(n)$-bit solution in a relatively simple manner. Instead of maintaining structures up to $\log^* n$ levels, we stop at an earlier stage, which suffices to handle such queries and also helps bound the space. Furthermore, we substitute the standard suffix and prefix tree topologies with their sampled counterparts. However, for Type-B queries, we first decode the PA values of the predecessor and the successor which are multiples of δ (i.e., $i' = \delta\lfloor i/\delta\rfloor$ and $i'' = \delta\lceil i/\delta\rceil$), by treating them as Type-A queries. Then, we decode all the entries in $\text{PA}[i'\,..\,i'']$, first in an unsorted manner; afterward, we sort them to obtain $\text{PA}[i]$. We accomplish this by splitting this range into the minimal number of subranges such that each subrange corresponds to a subtree of the prefix tree, and processing each subrange carefully. This part is more technical.

4.2.2 Handling Type-A Queries

Observe that our $O(n)$-bit encoding of PA composed of two tree topologies and several components corresponding to each $h \in [1, \log^* n]$. However, only the components corresponding to $h \in [1, f]$ are utilized when decoding PA$[i]$, where f is the smallest integer in $[1, \log^* n]$ such that Δ_f divides i (recall that Δ_h is $(\log^{(h)} n)^2$ round to the next power of 2). We exploit this fact to design a space efficient structure for Type-A queries. Let $\lambda \in [1, \log^* n]$ be the integer, where $\Delta_{\lambda+1} < \delta^2 \leq \Delta_\lambda$. We now maintain the components in Section 3.2.2 for only selected values of h (with some modifications) as follows:

1. All values of $h \in [1, \lambda - 1]$;
2. $h = \lambda$, but modify the value of Δ_h to δ^2.
3. $h = \lambda + 1$, but modify the value of Δ_h to δ.

The resulting space in bits is

$$2\sum_{h=1}^{\lambda+1} \frac{n}{\Delta_h} \log \Delta_{h-1} \leq \frac{2n}{\delta} \log \delta^2 + \frac{2n}{\delta^2} \log \Delta_{\lambda-1} + 2\sum_{h=1}^{\lambda-1} \frac{n}{\Delta_h} \log \Delta_{h-1} = O((n/\delta) \log \delta).$$

Our previous algorithm (as described in Section 3.3) can be applied to determine PA$[i]$ with the same time complexity. However, the topologies of suffix tree and prefix tree still takes $O(n)$ bits. We replace them with the topologies of the sparse prefix tree (SPT) and sparse suffix tree (SST), which require less space. Additionally, we maintain the small space structure in Lemma 13 supporting LCE queries.

4.2.2.1 Replacing Prefix Tree with Sparse Prefix Tree (SPT)

We sample a subset of $O(n/\delta)$ nodes in the prefix tree as follows:

1. leaves ℓ_k, where k is a multiple of δ;
2. the lowest common ancestor (LCA) of every pair of consecutive leaves sampled in step 1;
3. the leftmost and rightmost descendant leaves of sampled nodes obtained in step 2.

Let $\mathcal{L}$ be the set of sampled leaves obtained after the above procedure. Then, the SPT is the compacted trie built from the reverse of circular prefixes corresponding only to the leaves in $\mathcal{L}$ — that is, a sparse prefix tree. We maintain its topology in succinct space and a bit string $B[1..n]$, where $B[k] = 1$ iff $\ell_k \in \mathcal{L}$, as an indexible dictionary. The space required is $O((n/\delta) \log \delta)$ bits as there are $\mathcal{O}(\frac{n}{\delta})$ 1's in B. We now present some useful results.

▶ **Lemma 14.** *Given two sampled leaves $\ell_k, \ell_{k'}$ in the prefix tree, the size of their* LCA *can be returned in $O(1)$ time using SPT and the associated bit string.*

Proof. The sampled leaf ℓ_k (resp., $\ell_{k'}$) in the prefix tree corresponds to the leaf $\ell_{\mathsf{rank}_B(k,1)}$ (resp., $\ell_{\mathsf{rank}_B(k',1)}$) in SPT. Therefore, find the range $[s, e]$ of leaves in the subtree of the node $\text{LCA}(\ell_{\mathsf{rank}_B(k,1)}, \ell_{\mathsf{rank}_B(k',1)})$ in SPT and return $\mathsf{select}_B(e, 1) - \mathsf{select}_B(s, 1) + 1$. ◀

The prefix tree topology play the following role in our earlier algorithm. Given i, (i) find which among the nodes $u' = \text{LCA}(\ell_{i'_{f-1}}, \ell_i)$ and $u'' = \text{LCA}(\ell_i, \ell_{i''_{f-1}})$ is lower (call it u) to decide which case to use in the decoding procedure and (ii) return the range of leaves $[a, b]$ of u (note that we don't require the pre-order rank of u). Since i, i'_{f-1}, i''_{f-1} are all divisible by δ (hence sampled) in the case of Type-A queries, we can apply Lemma 14 and find the range of leaves of u' and u'' first, then infer $u = u'$ iff the range of u' is the same or contained within the range of u''. The subsequent steps in the recurrence also work since both i'_h and i''_h are divisible by δ for all $h \leq f$. In summary, a sparse prefix tree topology can perform the required functionalities in our algorithm without any performance slowdown.

4.2.2.2 Replacing Suffix Tree with Sparse Suffix Tree (SST)

The SST is the compacted trie constructed from the circular suffixes corresponding to the leaves ℓ_k of the suffix tree, where k is a multiple of δ. We maintain its topology in succinct space, requiring $O(n/\delta)$ bits.

The suffix tree topology serves the purpose of finding the suffix range $[\mathsf{sp}(\overleftarrow{Q}), \mathsf{ep}(\overleftarrow{Q})]$ of $\overleftarrow{Q}$ in $O(\log n)$ time, given (i) the length of Q (this can be computed using pLCS_π as in Section 3.3), (ii) a leaf ℓ_g in the suffix tree, where $\overleftarrow{Q}$ is a prefix of $\text{str}(\ell_g)$, (iii) the size of the suffix range of Q, i.e., $\mathsf{ep}(\overleftarrow{Q}) - \mathsf{sp}(\overleftarrow{Q}) + 1$ (specifically, $\mathsf{size}(u)$ is given, where u is a node in the prefix tree). It is worth noting that this task can be accomplished without the suffix tree topology, and even without the third input parameter. We can exploit the fact that $\text{LCE}(\text{SA}[g], \text{SA}[p]) \geq |Q|$ if and only if $p \in [\mathsf{sp}(\overleftarrow{Q}), \mathsf{ep}(\overleftarrow{Q})]$, and hence we can apply binary search on p and determine the answer using $O(\log n)$ LCE queries.

We speed up this process by using the additional information in (iii) as follows. If $\mathsf{size}(u) \leq 2\delta$, we still apply the binary search. The number of LCE / SA queries required is $O(\log \delta)$. Otherwise, find an approximate range using SST first, and then apply binary search on ranges of size $O(\delta)$ as follows.

1. Find a value g^* closer to g, such that g^* is a multiple of δ (thus ℓ_{g^*} is sampled) and $\text{str}(\ell_{g^*})$ is also prefixed by $\overleftarrow{Q}$ as follows. Let $g' = \delta\lfloor g/\delta \rfloor$ and $g'' = \delta\lceil g/\delta \rceil$. We fix $g^* = g'$ if $\text{LCE}(\text{SA}[g], \text{SA}[g']) \geq |Q|$, otherwise $g^* = g''$ (in this case $\text{LCE}(\text{SA}[g], \text{SA}[g'']) \geq |Q|$ is guaranteed as $\mathsf{size}(u) > 2\delta$).
2. Find (g^*/δ)-th leaf in SST, and return the range $[sp^*, ep^*]$ of leaves of its lowest ancestor with $\mathsf{size}(\cdot) \geq \lfloor \mathsf{size}(u)/\delta \rfloor - 1$. This step takes only $O(\log n)$ time (via $\mathsf{level\_ancestor}$ queries as before, but on SST). Observe that our sampling scheme guarantees $\mathsf{sp}(\overleftarrow{Q}) \in [\delta \cdot sp^* - O(\delta), \delta \cdot sp^*]$ and $\mathsf{ep}(\overleftarrow{Q}) \in [\delta \cdot ep^*, \delta \cdot ep^* + O(\delta)]$. Therefore we can now restrict our binary search into a smaller range of size $O(\delta)$, requiring only $O(\log \delta)$ LCE and SA queries.

This subroutine for finding suffix range $[\mathsf{sp}(\overleftarrow{Q}), \mathsf{ep}(\overleftarrow{Q})]$ of $\overleftarrow{Q}$ takes $O(\log n + (t_{sa} + t_{\text{LCE}}) \cdot \log \delta) \in O(\log n + (\delta \cdot t_{wt} + t_{sa})\delta \cdot \log \delta)$ time. After that, we continue with steps (b), (c), and (d) as in Section 3.3.

By substituting the space–time complexities of the components, incorporating updated data structures, we obtain the following bound for Type-A queries.

▶ **Lemma 15** (Type-A Queries). *By maintaining auxiliary structures of space $O((n/\delta)\log\delta)$ bits in total, we can answer Type-A queries in time*

$$O\left((\log n + t_{quantile} + (\delta \cdot t_{wt} + t_{sa})\delta \cdot \log \delta) \log^* n\right).$$

4.2.3 Handling Type-B Queries

Type-B queries require a more involved solution than Type-A queries. Maintain the structure in Lemma 11 supporting $\text{pLCS}^{\approx}_{\pi}$ queries, for values of π in $\{-1, +1\}$. Also associate $O(\log \delta)$ bits of satellite information (to be defined shortly) with each integer in $[1, n]$ that is a multiple of δ. The total space is $O((n/\delta)\log\delta)$ bits.

To retrieve $\text{PA}[i]$, where i is not divisible by δ, we employ a procedure that decodes a segment of PA containing $\text{PA}[i]$. Let $i' = \delta\lfloor i/\delta \rfloor$, $i'' = \delta\lceil i/\delta \rceil = i' + \delta$, and $u = \text{LCA}(\ell_{i'}, \ell_{i''})$. We now proceed to define some nodes in the prefix tree as illustrated in Figure 2.

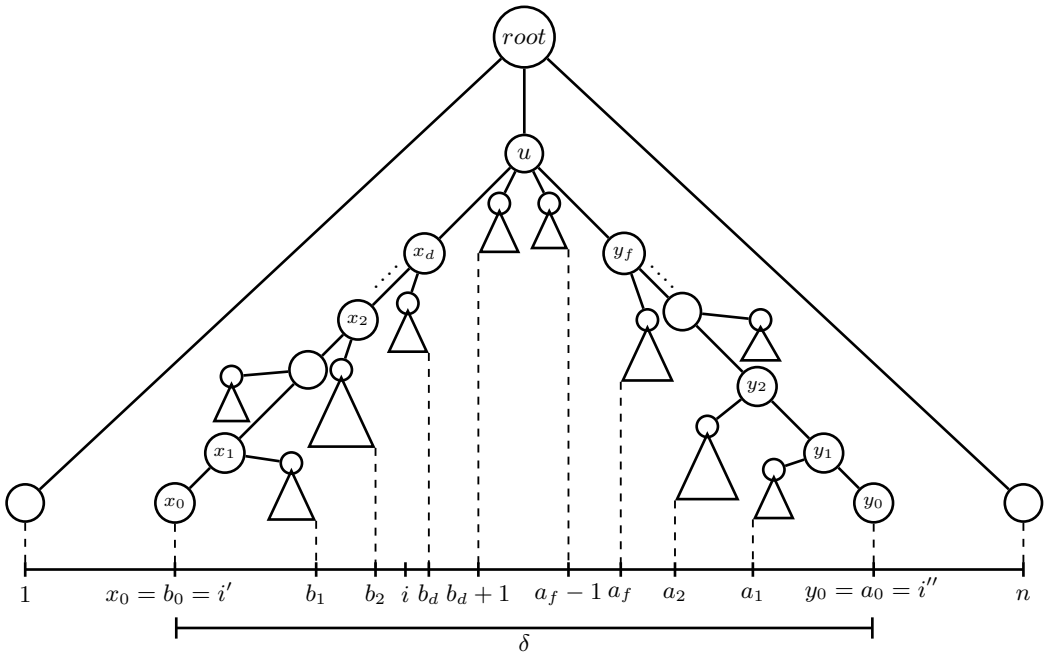

Figure 2 Illustration of Type-B queries

- $x_1, x_2, \ldots, x_d$ are nodes that lie on the path from $\ell_{i'}$ to u (excluding endpoints), ordered in descending $\mathsf{depth}(\cdot)$, such that $x_0 = \ell_{i'}$ and x_k for $k \in [1, d]$ is the lowest ancestor of x_{k-1} whose child on the path to $\ell_{i'}$ has a right sibling. Let $[\cdot, b_k]$ denote the range of leaves of x_k for $k \in [0, d]$. Therefore, $(i', b_d]$ denotes the range of leaves after i', such that the LCA of any of those leaves with $\ell_{i'}$ is a node in $\{x_1, x_2, \ldots, x_d\}$. The following observation allows us to jump from node x_{k-1} to node x_k for all $k \in [1, d]$.

  ▶ **Observation 1.** *For all $k \in [1, d]$ we have* $\mathrm{LCA}(\ell_{b_{k-1}}, \ell_{b_{k-1}+1}) = x_k$*; therefore* $\mathrm{strlen}(x_k) = \mathrm{pLCS}_{+1}[\mathrm{PA}[b_{k-1}]]$.

- $y_1, y_2, \ldots, y_f$ are nodes that lie on the path from $\ell_{i''}$ to u (excluding endpoints), ordered by descending $\mathsf{depth}(\cdot)$, such that $y_0 = \ell_{i''}$ and y_k for $k \in [1, f]$ is the lowest ancestor of y_{k-1} whose child on the path to $\ell_{i''}$ has a left sibling. Let $[a_k, \cdot]$ denote the range of leaves of y_k for $k \in [0, f]$. Therefore, $[a_f, i'')$ denotes the range of leaves before i'', such that the LCA of any of those leaves with $\ell_{i''}$ belongs to $\{y_1, y_2, \ldots, y_f\}$. The following observation allows us to jump from node y_{k-1} to node y_k for all $k \in [1, f]$.

  ▶ **Observation 2.** *For all $k \in [1, f]$, we have* $\mathrm{LCA}(\ell_{a_{k-1}}, \ell_{a_{k-1}-1}) = y_k$*; therefore* $\mathrm{strlen}(y_k) = \mathrm{pLCS}_{-1}[\mathrm{PA}[a_{k-1}]]$.

Our procedure for decoding $\mathrm{PA}[i]$ depends on which of the intervals $(i', b_d]$, $[b_d + 1, a_f - 1]$, or $[a_f, i'')$ contains i. The satellite information associated with i' is $(d, f, b_d - i', a_f - i')$ (all four values are bounded by δ), which helps to decide the case.

4.2.3.1 CASE $i \in (b_0, b_d]$

First obtain $\mathrm{PA}[i']$ using a Type-A query, then initialize $b_0 = i'$ and $\mathrm{strlen}(x_0) = \mathrm{PA}[b_0]$, the length of the prefix ending at b_0. Then decode $\mathrm{PA}(b_0, b_d]$ in d steps; at step $k \in [1, d]$, we compute $\mathrm{strlen}(x_k), \mathsf{size}(x_k), b_k$ and $\mathrm{PA}(b_{k-1}, b_k]$. Note that $(b_0, b_d] = (b_0, b_1] \cup (b_1, b_2] \cup \ldots (b_{d-1}, b_d]$. Let Q_t be the prefix of $\mathrm{str}(\ell_{i'})$ of length t and α_t be the $(t+1)$-th character of $\mathrm{str}(\ell_{i'})$. Therefore, $\overleftarrow{Q_t} = \mathsf{T}[\mathrm{PA}[i'] - t + 1 \,..\, \mathrm{PA}[i']]$ and $\alpha_t = \mathsf{T}[\mathrm{PA}[i'] - t]$. When $t = \mathrm{strlen}(x_k)$, we let $Q = Q_t$ and $\alpha = \alpha_t$.

1. **Finding** $\mathrm{strlen}(x_k)$**:** Observe that $\mathrm{strlen}(x_k)$ is the largest $t < \mathrm{strlen}(x_{k-1})$ such that $\mathrm{BWT}[\mathsf{sp}(\overleftarrow{Q_t}), \mathsf{ep}(\overleftarrow{Q_t})]$ *contains a character* $\beta_t > \alpha_t$. However, finding $[\mathsf{sp}(\overleftarrow{Q_t}), \mathsf{ep}(\overleftarrow{Q_t})]$ can be expensive. Thankfully, we have an alternative procedure to verify this condition for a given t. Find $z = \mathrm{ISA}[\mathrm{PA}[i'] - t + 1]$, which is an entry in $[\mathsf{sp}(\overleftarrow{Q_t}), \mathsf{ep}(\overleftarrow{Q_t})]$, then find the largest number z' before z, and the smallest number z'' after z, such that $\mathrm{BWT}[z'], \mathrm{BWT}[z''] > \alpha_t = \mathrm{BWT}[z]$, using wavelet tree queries. The condition is satisfied if and only if $\mathrm{LCE}(\mathrm{ISA}[z], \mathrm{ISA}[z']) \geq t$ or $\mathrm{LCE}(\mathrm{ISA}[z], \mathrm{ISA}[z'']) \geq t$. We next determine an l such that $\mathrm{strlen}(x_k) \in [l - \delta, l]$, meaning we only need to iterate over the values of t in $[l - \delta, l]$ for finding $\mathrm{strlen}(x_k)$. To find l, we rely on observation 1 that $x_k = \mathrm{LCA}(\ell_{b_{k-1}}, \ell_{b_{k-1}+1})$ and $\mathrm{strlen}(x_k) = \mathrm{pLCS}[\mathrm{PA}[b_{k-1}]]$. Therefore, $\mathrm{pLCS}_1^{\approx}[\mathrm{PA}[b_{k-1}]] \leq \mathrm{strlen}(x_k) \leq \mathrm{pLCS}_1^{\approx}[\mathrm{PA}[b_{k-1}]] + \delta$, and we compute $\mathrm{pLCS}_1^{\approx}[\mathrm{PA}[b_{k-1}]]$ in $O(\log \delta)$ time (see Lemma 11), and set $l = \min\{\mathrm{strlen}(x_{k-1}) - 1, \mathrm{pLCS}_1^{\approx}[\mathrm{PA}[b_{k-1}]] + \delta\}$. The total time is $O((t_{wt} + t_{sa} + t_{\mathrm{LCE}})\delta) \subseteq O(t_{\mathrm{LCE}} \cdot \delta)$.
2. **Finding** $\mathsf{size}(x_k)$**:** At first, we find an approximation $\mathsf{size}^{\approx}(x_k) \in [\mathsf{size}(x_k) - 2\delta, \mathsf{size}(x_k)]$ using SPT and the associated bit vector B as follows. Note that $x_k = \mathrm{LCA}(\ell_{i'}, \ell_{b_{k-1}+1})$ and the leaf $\ell_{i'}$ is sampled (recall SPT's construction). Therefore, if $\ell_{b_{k-1}+1}$ is also sampled (i.e., $B[b_{k-1} + 1] = 1$), then we can find the exact value of $\mathsf{size}(x_k)$ in $O(1)$ time using Lemma 14. Otherwise, find the leaves $\ell_{\mathsf{rank}_B(b_{k-1}+1,1)}, \ell_{\mathsf{rank}_B(b_{k-1}+1,1)+1}$ in SPT, the child of their LCA on the path to $\ell_{\mathsf{rank}_B(i',1)}$, and the range $[s, s']$ of leaves of that child. Then return $\mathsf{select}_B(s', 1) - \mathsf{select}(s, 1) + 1$ as $\mathsf{size}^{\approx}(x_k)$. Our sampling scheme guarantees that the value returned lies in the range $[\mathsf{size}(x_k) - 2\delta, \mathsf{size}(x_k)]$. The time complexity is $O(1)$. We now proceed to computing the exact value of $\mathsf{size}(x_k)$. Since $Q = \mathrm{str}(x_k)$, we have $\mathsf{size}(x_k) = \mathsf{ep}(\overleftarrow{Q}) - \mathsf{sp}(\overleftarrow{Q}) + 1$. Therefore we compute the range $[\mathsf{sp}(\overleftarrow{Q}), \mathsf{ep}(\overleftarrow{Q})]$ as follows. To begin with, we know an entry $\mathrm{ISA}[\mathrm{PA}[i'] - |Q| + 1] \in [\mathsf{sp}(\overleftarrow{Q}), \mathsf{ep}(\overleftarrow{Q})]$, as well as $\mathsf{size}^{\approx}(x_k)$. At the point, we can follow the same procedure detailed in Section 4.2.2.2, where we first identify an approximate range that is sufficiently close to the actual one using sparse suffix tree in $O(\log n)$ time, then perform a binary search requiring only $O(\log \delta)$ LCE and ISA queries. The time required is $O(\log n + (t_{sa} + t_{\mathrm{LCE}}) \cdot \log \delta) \subseteq O(\log n + t_{\mathrm{LCE}} \cdot \log \delta)$.
3. **Finding** b_k**:** Let α be the leading character on the path from x_k to its child on the path to $\ell_{i'}$. Count the entries in $\mathrm{BWT}[\mathsf{sp}(\overleftarrow{Q}), \mathsf{ep}(\overleftarrow{Q})]$ that are above $\alpha = \mathsf{T}[\mathrm{PA}[i'] - |Q|]$ (using a wavelet tree query). This gives the number of leaves in the range $(b_{k-1}, b_k]$. Therefore we simply add b_{k-1} with this count and return as b_k. Time required is $O(t_{wt})$.
4. **Finding** $\mathrm{PA}(b_{k-1}, b_k]$**:** The set $\mathcal{P} = \{\mathrm{SA}[z] + |Q| - 1 \mid z \in [\mathsf{sp}(\overleftarrow{Q}), \mathsf{ep}(\overleftarrow{Q})], \mathrm{BWT}[z] > \alpha\}$ is a collection of elements in $\mathrm{PA}(b_{k-1}, b_k]$, which can be obtained in $O(t_{wt} + t_{sa})$ time per element using wavelet tree and $\mathrm{SA}[\cdot]$ queries. However, we do not know the order in which they appear in $\mathrm{PA}(b_{k-1}, b_k]$; therefore our next goal is to recover $\mathrm{PA}(b_{k-1}, b_k]$ by sorting the elements in $\mathcal{P}$ in the ascending order of their $\mathrm{IPA}[\cdot]$ values. We first present a simple solution, and then describe a slightly improved one.

a. A simple solution using $\overleftarrow{\mathrm{LCE}}(\cdot,\cdot)$ queries: By fixing $D = \delta \log n$, we maintain the structure in Lemma 4 without changing the asymptotic space complexity, which supports $\overleftarrow{\mathrm{LCE}}(\cdot,\cdot)$ queries in $O(t_{\mathrm{LCE}} \cdot \log(\delta \log n))$ time. Then, for any $p, q \in \mathcal{P}$, we can decide the relative order of IPA$[p]$ and IPA$[q]$ by comparing the characters $\mathsf{T}[p - \overleftarrow{\mathrm{LCE}}(p,q)]$ and $\mathsf{T}[q - \overleftarrow{\mathrm{LCE}}(p,q)]$. Therefore, an efficient comparison-based sorting algorithm (such as merge sort) can be used to sort the elements of $\mathcal{P}$ in ascending order of their IPA$[\cdot]$ values using $O((b_k - b_{k-1}) \log(b_k - b_{k-1}))$ number of $\overleftarrow{\mathrm{LCE}}(\cdot,\cdot)$ queries. The resulting time complexity for this step is

$$(b_k - b_{k-1}) \log(b_k - b_{k-1}) \cdot t_{\mathrm{LCE}} \cdot (\log(\delta \log n) \subseteq O((b_k - b_{k-1}) \log \delta \cdot t_{\mathrm{LCE}} \cdot (\log \delta + \log \log n)).$$

b. An alternative solution: Our goal here is to avoid the direct use of $\overleftarrow{\mathrm{LCE}}(\cdot,\cdot)$ queries, and thereby removing the $\log \log n$ term in the previous time complexity. To achieve this, we reconstruct the subtree rooted at x_k on the fly. For any node w in the subtree rooted at x_k, let $\mathcal{P}(w) \subseteq \mathcal{P}$ denotes the set of PA$[\cdot]$ values corresponding to the leaves in the subtree of w. Iterating over $p \in \mathcal{P}$, our algorithm return the sets $\mathcal{P}(w_1), \mathcal{P}(w_2), \ldots$, in that order, where nodes $w_1, w_2, \ldots$ are the children (in the left to right order) of node $w = \mathrm{LCA}(\ell_{\mathrm{IPA}[p]}, \ell_{\mathrm{IPA}[p]+1})$. Note that the preorder rank of w is not known even after processing p. Therefore, based on $\mathcal{P}(w)$, we assign to w a unique label, defined as the ordered pair consisting of the size of $\mathcal{P}(w)$ and the smallest element of $\mathcal{P}(w)$. With this labeling and parent–child information for all internal nodes, we can easily reconstruct the tree in a top-down fashion and recover PA$(b_{k-1}, b_k]$. We now present the details of processing a specific $p \in \mathcal{P}$. The first step is to compute $\mathrm{strlen}(w) = \mathrm{pLCS}[p]$, which is given by the expression below when $p \neq \mathrm{PA}[b_k]$ (in that case $\mathrm{PA}[\mathrm{IPA}[p] + 1] \in \mathcal{P}$); when $p = \mathrm{PA}[b_k]$, it will not return any value.

$$\max\{\overleftarrow{\mathrm{LCE}}(p,q) \mid q \in \mathcal{P} \backslash \{p\}, \overleftarrow{T[1\,..\,q]} \texttt{ comes after } \overleftarrow{T[1\,..\,p]} \texttt{ lexicographically}\}$$

Instead of evaluating this expression by applying $\overleftarrow{\mathrm{LCE}}(\cdot,\cdot)$ directly, we compute $L = \mathrm{pLCS}^{\approx}[p]$ in $O(\log \delta)$ time (see Lemma 11), where $\mathrm{pLCS}[p] - \delta \leq L \leq \mathrm{pLCS}[p]$. Then, iterate over $q \in \mathcal{P}$, compute $\mathrm{LCE}(p - L + 1, q - L + 1)$, and consider the following two cases:

- **i.** If $\mathrm{LCE}(p - L + 1, q - L + 1) < L$, then we know $\overleftarrow{\mathrm{LCE}}(p,q) < L \leq \mathrm{strlen}(w)$. Therefore discard q.
- **ii.** Otherwise we have $\overleftarrow{\mathrm{LCE}}(p,q) = L + \overleftarrow{\mathrm{LCE}}(p - L, q - L)$. In that case, extract the substrings $\mathsf{T}[p - L - \delta\,..\,p - L]$ and $\mathsf{T}[q - L - \delta\,..\,q - L]$ and compare them. If the substrings differ, we can compute $\overleftarrow{\mathrm{LCE}}(p - L, q - L)$ (thus compute $\overleftarrow{\mathrm{LCE}}(p,q)$) and determine their lexicographic order (and thus the lexicographic order of $\overleftarrow{T[1\,..\,q]}$ and $\overleftarrow{T[1\,..\,p]}$). If the substrings match, we infer that $\overleftarrow{\mathrm{LCE}}(p,q) > L + \delta \geq \mathrm{strlen}(w)$, and therefore both $\ell_{\mathrm{IPA}[q]}$ and $\ell_{\mathrm{IPA}[p]}$ lie in the subtree of the same child of w, and $\overleftarrow{T[1\,..\,q]}$ comes before $\overleftarrow{T[1\,..\,p]}$ in lexicographically.

The above step for a specific p, q takes $O(t_{\mathrm{LCE}} + t_{sa} + \delta \cdot t_{wt})$ time. By taking the maximum value of $\overleftarrow{\mathrm{LCE}}(p,q)$ over all q that satisfy the required condition on the lexicographic order, we obtain $\mathrm{strlen}(w)$. We next collect all positions q such that $\overleftarrow{\mathrm{LCE}}(p,q) \geq \mathrm{strlen}(w)$, group them into sets according to the character $T[q - \mathrm{strlen}(w)]$, and output these sets in order of increasing character value they correspond to. It is straightforward to verify that the sets produced in this manner are exactly $\mathcal{P}(w_1), \mathcal{P}(w_2), \ldots$. Moreover, every internal node in the subtree rooted at x_k will be

processed after we have iterated over all $p \in \mathcal{P} \backslash \{\text{PA}[b_k]\}$. The overall time complexity of this step is $O((b_k - b_{k-1})^2 \cdot (t_{\text{LCE}} + t_{sa} + \delta \cdot t_{wt})) \subseteq O((b_k - b_{k-1})^2 \cdot t_{\text{LCE}})$.

We use the second method, although its time complexity is quadratic in the size of the tree, because the first method would add extra $\log \log n$ factors to the time complexity.

The combined time for all 4 steps for all $k \in [1, d]$, where $d \leq \delta$ is bounded as follows:

$$\sum_{k=1}^{d} \Big(t_{\text{LCE}} \cdot \delta + (\log n + t_{\text{LCE}} \cdot \log \delta) + t_{wt} + (b_k - b_{k-1})^2 \cdot t_{\text{LCE}} \Big)$$

$$\leq t_{\text{LCE}} \cdot \delta^2 + (\log n + t_{\text{LCE}} \cdot \log \delta) \cdot \delta + t_{wt} \cdot \delta + (b_d - b_0)^2 \cdot t_{\text{LCE}}$$

By substituting $b_d - b_0 = O(\delta)$ and $t_{\text{LCE}} = O((t_{sa} + \delta \cdot t_{wt})\delta)$, the expression simplifies to $O(\delta \cdot \log n + (t_{sa} + \delta \cdot t_{wt})\delta^3)$.

4.2.3.2 CASE $i \in [b_d + 1, a_f - 1]$

Let $Q = \text{str}(u)$ and α be the leading character on the path from u to its child on the path to $\ell_{i'}$ and β be the leading character on the path from u to its child on the path to $\ell_{i''}$. Then the set $\mathcal{P} = \{\text{SA}[z] + |Q| - 1 \mid z \in [\mathsf{sp}(\overleftarrow{Q}), \mathsf{ep}(\overleftarrow{Q})], \text{BWT}[z] \in [\alpha + 1, \beta - 1]\}$ is the collection of elements in $\text{PA}[b_d + 1, a_f - 1]$. Since $\mathsf{size}(u)$, $\text{strlen}(u), \text{PA}[i'], \text{PA}[i'']$ are readily available (via Type-A query), the elements in the set $\mathcal{P}$ can be easily extracted in time $O(t_{wt} + t_{sa})$ per element, but not in the order as they appear in PA. Finally, to recover $\text{PA}[b_d + 1, a_f - 1]$, we follow the same procedure as in the first case. The resulting time is $O((t_{sa} + \delta \cdot t_{wt})\delta^3)$.

4.2.3.3 CASE $i \in [a_f, i'')$

Here we obtain $\text{PA}[i'']$ first using a Type-A query, then initialize $a_0 = i''$ and $\text{strlen}(y_0) = \text{PA}[a_0]$, the length of the prefix ending at a_0. Then decode $\text{PA}[a_f, a_0)$ in f steps, where at step $k \in [1, f]$, we compute $\text{strlen}(y_k)$, $\mathsf{size}(y_k)$, a_k and $\text{PA}[a_k, a_{k-1})$. Note that $[a_f, a_0) = [a_f, a_{f-1}) \cup \cdots \cup [a_2, a_1) \cup [a_1, a_0)$. Suppose $Q = \text{str}(y_k)$ and $\beta = \mathsf{T}[\text{PA}[i''] - |Q|]$, then the set elements in $\text{PA}[a_k, a_{k-1})$ is given by $\mathcal{P} = \{\text{SA}[z] + |Q| - 1 \mid z \in [\mathsf{sp}(\overleftarrow{Q}), \mathsf{ep}(\overleftarrow{Q})], \text{BWT}[z] < \beta\}$. This is analogous to that of the first case, therefore we invoke a symmetric procedure to handle this case. In contrast to the first case, where we rely on Observation 1 and use $\text{pLCS}^{\approx}_{1}[\cdot]$ queries, here we rely on Observation 2 and use $\text{pLCS}^{\approx}_{-1}[\cdot]$ queries. Total time is $O(\delta \cdot \log n + (t_{sa} + \delta \cdot t_{wt})\delta^3)$ time (similar to the first case).

▶ **Lemma 16** (Type-B Queries). *The combined space of all auxiliary structures for Type-B queries is $O((n/\delta) \log \delta)$ bits and the query time is $O\left(\delta \cdot \log n + (\delta \cdot t_{wt} + t_{sa})\delta^3\right)$.*

By combining Lemma 15 and Lemma 16, we obtain the Theorem below.

▶ **Theorem 17.** *Let the Burrows-Wheeler Transform (BWT) of a text $\mathsf{T}[1\,..\,n]$ be maintained in a data structure that supports basic queries (*rank, select, *and* access*) in t_{wt} time, and range quantile queries in $t_{quantile}$ time. Then, for any parameter $\tau \geq 1$, we can maintain a sampled subset of n/τ suffix array values using $O((n/\tau) \log n)$ bits of additional space, and support suffix array (SA) and inverse suffix array (ISA) queries in time $t_{sa} = O(\tau \cdot t_{wt})$. By maintaining an auxiliary structure of space $O((n/\delta) \log \delta)$ bits for a parameter δ, we can support prefix array (PA) queries in time*

$$t_{pa} = O\left((\log n \cdot (\log^* n + \delta) + t_{quantile} \cdot \log^* n + (\delta + \tau) \cdot t_{wt} \cdot (\delta \log \delta \log^* n + \delta^3)\right)$$

$$\subseteq O((\log n + t_{quantile} + (\delta + \tau) \cdot t_{wt}) \cdot (\delta \log \delta \log^* n + \delta^3)).$$

Additionally we can support $\mathrm{LCE}(\cdot,\cdot)$ *queries in time* $t_{\mathrm{LCE}} = O((\delta \cdot t_{wt} + t_{sa})\delta)$ *and* $\overleftarrow{\mathrm{LCE}}(\cdot,\cdot)$ *queries in time* $O(t_{\mathrm{LCE}} \cdot (\log \delta + \log\log n))$.

By employing the multiary wavelet tree, where $t_{wt} = O(1 + \log\sigma/\log\log n)$, $t_{quantile} = O(\log\sigma)$ and setting $\delta = \kappa \log \kappa$ and $\tau = \kappa \log n$, we obtain the following result.

▶ **Corollary 18.** *Let* $\mathsf{T}[1\,..\,n]$ *be a text over an alphabet of size* σ*. We can maintain its Burrows-Wheeler Transform (BWT) as a wavelet tree data structure in* $n\log\sigma + o(n) + O(\sigma \log n)$ *bits that supports basic queries (rank, select, access) in time* $O(1 + \log\sigma/\log\log n)$ *and range quantile queries in time* $O(\log\sigma)$*. For an integer parameter* κ*, we can maintain an auxiliary structure of space* $O(n/\kappa)$ *bits and support suffix array (SA) and inverse suffix array (ISA) queries in time* $t_{sa} = O((1 + \log\sigma/\log\log n) \cdot \kappa \log n)$*, and prefix array (PA) queries in time* $t_{pa} = t_{sa} \cdot O((\kappa^3 \log\kappa + \kappa \log^* n) \cdot \log^2 \kappa)$.

Additionally we can support $\mathrm{LCE}(\cdot,\cdot)$ *queries in time* $t_{\mathrm{LCE}} = O(t_{sa} \cdot \kappa \log\kappa)$ *and* $\overleftarrow{\mathrm{LCE}}(\cdot,\cdot)$ *queries in time* $O(t_{\mathrm{LCE}} \cdot (\log\kappa + \log\log n))$.

5 A Note On Inverse Prefix Array (IPA) Queries

We present a simple approach for supporting IPA[·] queries. Assuming the availability of data structures supporting PA[·] and $\mathrm{LCE}(\cdot,\cdot)$ queries, we maintain the auxiliary structure from Lemma 4 with the parameter set to $D = \kappa \log n$. This structure requires $O(n/\kappa)$ additional bits of space and supports $\overleftarrow{\mathrm{LCE}}(\cdot,\cdot)$ queries in $\overleftarrow{\mathrm{LCE}}(\cdot,\cdot)$ time $O(t_{\mathrm{LCE}} \cdot \log(\kappa \log n))$.

Given a text position j, we compute IPA[j] by performing a binary search over $i \in [1, n]$. For a candidate index i, we first compute $x = \mathrm{PA}[i]$. If $x = j$, we return i and terminate the search. Otherwise, we compute $l = \overleftarrow{\mathrm{LCE}}(x, j)$ and compare the characters $\mathsf{T}[x - l]$ and $\mathsf{T}[j - l]$ (which must be different). If $\mathsf{T}[x - l] < \mathsf{T}[j - l]$, then we conclude that $i < \mathrm{IPA}[j]$; otherwise, $i > \mathrm{IPA}[j]$. This provides the necessary comparison predicate for binary search. The time complexity is $O((t_{pa} + t_{\mathrm{LCE}} \cdot \log(\kappa \log n)) \log n) \subseteq O(t_{pa} \cdot \log n \cdot \log\log n)$.

▶ **Theorem 19.** *With the structure described in Theorem 17, we can maintain an auxiliary data structure of space* $O(n/\kappa)$ *bits that supports inverse prefix array (IPA) queries in time* $O(t_{pa} \cdot \log n \cdot \log\log n)$.

6 Conclusions

We present a solution for encoding the suffix array and inverse suffix array of the reversed text, assuming the availability of the FM-index of the original text. We can answer suffix array queries on the reversed text in time close to that of suffix array queries on the original text. However, for inverse suffix array queries on the reversed text, we require roughly a factor of $(\log n)$ more time than for inverse suffix array queries on the original text. Narrowing this gap is an interesting problem, which we leave open for future research.

References

1 Paniz Abedin, Oliver A. Chubet, Daniel Gibney, and Sharma V. Thankachan. Contextual pattern matching in less space. In *Data Compression Conference, DCC 2023, Snowbird, UT, USA, March 21-24, 2023*, pages 160–167. IEEE, 2023. doi:10.1109/DCC55655.2023.00024.

2 Amihood Amir, Dmitry Keselman, Gad M. Landau, Moshe Lewenstein, Noa Lewenstein, and Michael Rodeh. Text indexing and dictionary matching with one error. *J. Algorithms*, 37(2):309–325, 2000. URL: https://doi.org/10.1006/jagm.2000.1104, doi:10.1006/JAGM.2000.1104.

3 Djamal Belazzougui, Travis Gagie, Simon Gog, Giovanni Manzini, and Jouni Sirén. Relative fm-indexes. In *String Processing and Information Retrieval: 21st International Symposium, SPIRE 2014, Ouro Preto, Brazil, October 20-22, 2014. Proceedings 21*, pages 52–64. Springer, 2014.

4 Philip Bille, Inge Li Gørtz, Mathias Bæk Tejs Knudsen, Moshe Lewenstein, and Hjalte Wedel Vildhøj. Longest common extensions in sublinear space. In Ferdinando Cicalese, Ely Porat, and Ugo Vaccaro, editors, *Combinatorial Pattern Matching - 26th Annual Symposium, CPM 2015, Ischia Island, Italy, June 29 - July 1, 2015, Proceedings*, volume 9133 of *Lecture Notes in Computer Science*, pages 65–76. Springer, 2015. doi:10.1007/978-3-319-19929-0_6.

5 Michael Burrows. A block-sorting lossless data compression algorithm. *SRS Research Report*, 124, 1994.

6 Andrea Farruggia, Travis Gagie, Gonzalo Navarro, Simon J Puglisi, and Jouni Sirén. Relative suffix trees. *The Computer Journal*, 61(5):773–788, 2018.

7 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 390–398. IEEE, 2000.

8 Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms*, 3(2):20, 2007. doi:10.1145/1240233.1240243.

9 Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011. doi:10.1137/090779759.

10 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *J. ACM*, 67(1):2:1–2:54, 2020. doi:10.1145/3375890.

11 Travis Gagie, Simon J. Puglisi, and Andrew Turpin. Range quantile queries: Another virtue of wavelet trees. In *String Processing and Information Retrieval (SPIRE)*, volume 5721 of *Lecture Notes in Computer Science*, pages 1–6. Springer, Berlin, Heidelberg, 2009. arXiv:0903.4726. doi:10.1007/978-3-642-03784-9_1.

12 Arnab Ganguly, Daniel Gibney, Sahar Hooshmand, M. Oguzhan Külekci, and Sharma V. Thankachan. Fm-index reveals the reverse suffix array. In Inge Li Gørtz and Oren Weimann, editors, *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020, Copenhagen, Denmark, June 17-19, 2020*, volume 161 of *LIPIcs*, pages 13:1–13:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. URL: https://doi.org/10.4230/LIPIcs.CPM.2020.13, doi:10.4230/LIPICS.CPM.2020.13.

13 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 841–850. SIAM, 2003.

14 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. doi:10.1137/S0097539702402354.

15 Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Ultra-succinct representation of ordered trees with applications. *J. Comput. Syst. Sci.*, 78(2):619–631, 2012. URL: https://doi.org/10.1016/j.jcss.2011.09.002, doi:10.1016/J.JCSS.2011.09.002.

16 Dominik Kempa and Tomasz Kociumaka. Collapsing the hierarchy of compressed data structures: Suffix arrays in optimal compressed space. In *64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023, Santa Cruz, CA, USA, November 6-9, 2023*, pages 1877–1886. IEEE, 2023. doi:10.1109/FOCS57990.2023.00114.

17 Tak Wah Lam, Ruiqiang Li, Alan Tam, Simon C. K. Wong, Edward Wu, and Siu-Ming Yiu. High throughput short read alignment via bi-directional BWT. In *2009 IEEE International*

Conference on Bioinformatics and Biomedicine, BIBM 2009, Washington, DC, USA, November 1-4, 2009, Proceedings, pages 31–36. IEEE Computer Society, 2009. doi:10.1109/BIBM.2009.42.

18 Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L. Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology*, 10(3):R25, 2009.

19 Moshe Lewenstein. Orthogonal range searching for text indexing. In Andrej Brodnik, Alejandro López-Ortiz, Venkatesh Raman, and Alfredo Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms - Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, volume 8066 of *Lecture Notes in Computer Science*, pages 267–302. Springer, 2013. doi:10.1007/978-3-642-40273-9_18.

20 Heng Li and Richard Durbin. Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

21 Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

22 Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

23 J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, 2001. doi:10.1137/S0097539799364092.

24 Gonzalo Navarro. Wavelet trees for all. *J. Discrete Algorithms*, 25:2–20, 2014. URL: https://doi.org/10.1016/j.jda.2013.07.004, doi:10.1016/J.JDA.2013.07.004.

25 Gonzalo Navarro. Contextual pattern matching. In *International Symposium on String Processing and Information Retrieval*, pages 3–10. Springer, 2020.

26 Gonzalo Navarro. Indexing highly repetitive string collections, part I: repetitiveness measures. *ACM Comput. Surv.*, 54(2):29:1–29:31, 2022. doi:10.1145/3434399.

27 Gonzalo Navarro. Indexing highly repetitive string collections, part II: compressed indexes. *ACM Comput. Surv.*, 54(2):26:1–26:32, 2022. doi:10.1145/3432999.

28 Mihai Pătraşcu. Succincter. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*, pages 305–313. IEEE Computer Society, 2008. doi:10.1109/FOCS.2008.83.

29 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4):43, 2007. doi:10.1145/1290672.1290680.

30 Luís MS Russo, Gonzalo Navarro, and Arlindo L Oliveira. Fully-compressed suffix trees. In *LATIN 2008: Theoretical Informatics: 8th Latin American Symposium, Búzios, Brazil, April 7-11, 2008. Proceedings 8*, pages 362–373. Springer, 2008.

31 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007. URL: https://doi.org/10.1007/s00224-006-1198-x, doi:10.1007/S00224-006-1198-X.

32 Dirk Strothmann. The affix array data structure and its applications to RNA secondary structure analysis. *Theor. Comput. Sci.*, 389(1-2):278–294, 2007. URL: https://doi.org/10.1016/j.tcs.2007.09.029, doi:10.1016/J.TCS.2007.09.029.

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

34 Peter Weiner. Linear pattern matching algorithms. *14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.