LF Successor: Compact Space Indexing for Order-Isomorphic Pattern Matching

Arnab Ganguly ✉
Dept. of CS, University of Wisconsin - Whitewater, USA.

Dhrumil Patel ✉
School of EECS, Louisiana State University, Baton Rouge, USA.

Rahul Shah ✉
School of EECS, Louisiana State University, Baton Rouge, USA.

Sharma V. Thankachan ✉
Dept. of CS, University of Central Florida, Orlando, USA.

Abstract

Two strings are order isomorphic iff the relative ordering of their characters is the same at all positions. For a given text $T[1, n]$ over an ordered alphabet of size σ, we can maintain an order-isomorphic suffix tree/array in $O(n \log n)$ bits and support (order-isomorphic) pattern/substring matching queries efficiently. It is interesting to know if we can encode these structures in space close to the text's size of $n \log \sigma$ bits. We answer this positively by presenting an $O(n \log \sigma)$-bit index that allows access to any entry in order-isomorphic suffix array (and its inverse array) in $t_{\mathsf{SA}} = O(\log^2 n / \log \sigma)$ time. For any pattern P given as a query, this index can count the number of substrings of T that are order-isomorphic to P (denoted by occ) in $O((|P| \log \sigma + t_{\mathsf{SA}}) \log n)$ time using standard techniques. Also, it can report the locations of those substrings in additional $O(occ \cdot t_{\mathsf{SA}})$ time.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Succinct data structures, Pattern Matching

Digital Object Identifier 10.4230/LIPIcs.ICALP.2021.66

1 Introduction

An index of a text $T[1, n]$ is a data structure that is capable of counting/reporting all those *substrings* of T that "*match*" (as per the problem specific definition of match) with any given pattern P. We use Σ to denote the alphabet set (of size σ) from which the characters in T are drawn from. WLOG, we assume that $T[n] = \$$, a special character that does not appear anywhere else in T. Two fundamental indexes for exact pattern matching are the suffix tree (ST) [21] and the suffix array (SA) [16]. Both takes $\Theta(n \log n)$ bits of space, which could be much larger than the $n\lceil \log \sigma \rceil$ bits needed to store T optimally. The first succinct indexes that use close to $n \log \sigma$ bits are the Compressed Suffix Array (CSA) [12] and the FM-index [6]. The crucial component of FM Index is Burrows-Wheeler Transform (BWT) [2] and its associated operation called the *Last-to-Front* (LF) *mapping*. The subsequent work lead to fully functional suffix trees in succinct space [20]. See [18] for further reading.

The parameterized ST [1, 17] and the order-isomorphic ST [4] are two popular ST variants under the class known as *suffix trees with missing suffix links* [3]. As they do not hold some critical structural properties of the original ST, their compression is challenging. Recently, Ganguly *et al.* showed that it is indeed possible to compress the parameterized suffix arrays. They implemented LF mapping using a BWT-like transformation called the parameterized BWT [9]. However, such a transformation is hard to define for order-isomorphic ST because LF mapping could lead to multiple changes in the (encoding of) associated suffixes. To that end, we present a novel technique for implementing the LF mapping (named LF *Successor*), leading to the first compact space index for order-isomorphic pattern matching.

48th International Colloquium on Automata, Languages, and Programming (ICALP 2021).
Editors: Nikhil Bansal, Emanuela Merelli, and James Worrell; Article No. 66; pp. 66:1–66:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1.1 Generalizing the Philosophy of BWT and LF Mapping

We present an overview of our approach using three problems: (i) traditional/exact matching, (ii) parameterized matching, and (iii) order-isomorphic matching, in that order, to show gradation and successive generalization of the LF mapping approach.

Indexing for Traditional Matching: The classic solution is the suffix tree (ST), a compact trie over all the suffixes of T. In a ST, each edge is labeled by some substring of T such that the concatenation of the edge labels on each root to leaf path represents a particular suffix of T. Based on the lexicographic order of the suffixes, a suffix array $\mathsf{SA}[1, n]$ (whose entries correspond to each leaf in the suffix tree in left to right order) marks the starting index (in T) of the suffix corresponding to the i^{th} leftmost leaf ℓ_i. Thus, $\mathsf{SA}[i] = t$ and the inverse suffix array entry $\mathsf{SA}^{-1}[t] = i$ iff the suffix corresponding to ℓ_i is $T[t, n]$. Inverse suffix array associates each position i in the text with leaf position (rank) of suffix $T[i..n]$ in the suffix tree. Also, for $t > 1$, $\mathsf{LF}(i) = j$ iff the leaf ℓ_j corresponds to the suffix $T[t-1, n]$, i.e., $\mathsf{SA}[j] = t - 1$. Formally, $\mathsf{LF}(i) = \mathsf{SA}^{-1}[\mathsf{SA}[i] - 1]$ (for the special case of $\mathsf{SA}[i] = 1$, we take $\mathsf{SA}^{-1}[0] = \mathsf{SA}^{-1}[n]$). The Burrows-Wheeler Transform is an array $\mathsf{BWT}[1, n]$, such that $\mathsf{BWT}[i] = T[\mathsf{SA}[i] - 1]$. Computing LF mapping is central to BWT based pattern matching, and in some sense, the BWT enables efficient computation of LF mapping. A fundamental result is that once we store the BWT, and its associated counting structures, we can replace the costly (space-wise) suffix array by a (cheaper) sampled suffix array [6].

Indexing for Parameterized Matching: Here, P matches with T at position i iff there is one-to-one correspondence between the characters of P and $T[i, i + |P| - 1]$. For example, $xwyx$ can match with $abca$ as x can be mapped to a, b to w, and c to y. However, $abca$ does not match with $xyxw$ because both a and c cannot be mapped to x. Baker [1] presented an encoding called $\mathsf{prev}(S)$ which encodes every character in the string by replacing it by its distance to the previous occurrence of the same character and using 0 if the character has not occurred before. For example, $\mathsf{prev}(xwxyywx) = 0020144$. It is not hard to see that two strings X and Y are a parameterized match iff $\mathsf{prev}(X) = \mathsf{prev}(Y)$. The parameterized suffix tree is a compact trie over all strings in $\{\mathsf{prev}(T[i, n-1]) \circ \$ \mid 1 \le i < n\}$, where $\circ$ denotes concatenation. Then, the parameterized matching of P in T can be performed via traditional matching of $\mathsf{prev}(P)$ in this suffix tree. The same notion of LF-mapping can be defined and implemented in succinct space using a BWT-like transform [9].

Indexing for Order-isomorphic Matching: This problem has received significant attention since its inception [4, 13, 15], not only due to its simple and elegant formulation, but also due its to ability to model string matching problems in other domains (e.g., music retrieval, analysis of time series data, etc) where the relative ordering of characters has to be matched rather than the string itself. Here, there is a total ordering between the symbols in Σ. The pattern P matches with text $T[1, n]$ at position i if for any j, k in $[1, |P|]$, $P[j] < P[k]$ iff $T[i + j - 1] < T[i + k - 1]$. Similar constraints apply for $P[j] > P[k]$ and $P[j] = P[k]$. For example, 1423 can match with 2957 but not with 2657 because $6 < 7$ and $4 > 3$. A new encoding "pred" works in this case. This is a slight modification of the scheme in [4].

▶ **Definition 1** (pred encoding). *Given a character $S[i]$ in string S, its predecessor is a character q which occurs in $S[1, i-1]$ such that $q \le S[i]$ and there is no other character r in $S[1, i-1]$ such that $q < r \le S[i]$. Given a string S, $\mathsf{pred}(S)[i]$ is defined as follows: let alphabet symbol q be the predecessor of $S[i]$ in $S[1, i-1]$ and let position j be the rightmost occurrence of q in $S[1, i-1]$. Then, $\mathsf{pred}(S)[i] = (i-j)$ if $q \ne S[i]$, $(i-j)'$ if $q = S[i]$, and 0 if q does not exist. Thus $\mathsf{pred}(S)$ is a string over the alphabet $\{0, 1, 1', 2, 2', \ldots, |S| - 1, (|S| - 1)'\}$.*

Thus, in pred encoding, every position (character) in T points to its closest predecessor on the left. For e.g., $\mathsf{pred}(0869514371) = 0\ 1\ 2\ 2\ 4\ 5\ 1\ 2\ 6\ 4'$. We refer to *primed* characters as an *equality* version of their non-primed counterparts. For example, $2'$ is equality variant of 2. It is easy to see that two strings X and Y are order-isomorphic iff $\mathsf{pred}(X) = \mathsf{pred}(Y)$.

The *order-isomorphic suffix tree* [4] of T is the compacted trie over all strings in $\{\mathsf{pred}(T[i, n-1]) \circ \$ \mid 1 \leq i < n\}$. We order the encoded characters as: $0 < 1 < 1' < 2 < 2' < \cdots < n-1 < (n-1)' < \$$. The *order-isomorphic suffix array* is such that its ith entry denotes the starting location of the suffix corresponding to ith leaf ℓ_i. Again, as in earlier cases, the LF mapping operation for an order isomorphic suffix tree where $j = \mathsf{LF}(i)$ maps leaf ℓ_i to leaf ℓ_j. The suffix j is obtained by prepending to suffix i the character which occurs just before the starting location of suffix i in T.

1.2 Challenges in Implementing (Generalised) LF Mapping Compactly

The challenge here is in deciding what needs to be precomputed and stored, so that $\mathsf{LF}(i)$ for any i can be computed efficiently. At its root, we need to solve the following: given two leaves ℓ_i and ℓ_j with $i < j$, how quickly can we decide whether $\mathsf{LF}(i) < \mathsf{LF}(j)$ or $\mathsf{LF}(i) > \mathsf{LF}(j)$.

In the case of **traditional matching**, the order between $\mathsf{LF}(i)$ and $\mathsf{LF}(j)$ will stay the same if the corresponding suffixes have the same *previous character* (which are $\mathsf{BWT}[i]$ and $\mathsf{BWT}[j]$). It will flip iff the previous character of the suffix corresponding to j is smaller than that of i in the lexicographic order. Therefore pair-wise comparison between such i and j can be computed in "bulk" for i against all j's, enabling "quick" computation of $\mathsf{LF}(i)$ [6].

In the case of **parameterized matching**, this order determination is more sophisticated [9]. Here, it becomes essential to see how prepending the previous character changes the canonical encoding of a suffix and how can this information be stored compactly. For example, consider $T[1, n] = abcabbadcb$ and the suffix $T[4, n] = abbadcb$. Its previous character $T[3]$ is c. When we prepend this character, the suffix (in traditional ST) becomes $cabbadcb$. The string corresponding to $T[4, n]$ in the parameterized suffix tree is $\mathsf{prev}(T[4, n]) = 0013004$. When $T[4, n]$ is prepended with c and prev is applied, apart from the insertion (of 0) at the beginning, there is one change within prev of $T[4, n]$, which is at the first occurrence of c in $T[4, n]$. Thus, the second last character in the encoding switches from 0 to 6, i.e., $\mathsf{prev}(T[3, n]) = 00013064$. Ganguly *et al.* [9] show how to record this change-location for each suffix succinctly using the paramaterized-BWT, which supports LF mapping. Again, as in the case of traditional pattern matching, we can compare two suffixes in terms of their LF mapping by comparing which suffix changes first – in case at least one of them changes before their longest common prefix (LCP). See [10, 14, 8] for some related results.

We now illustrate **order-isomorphic matching** using an example $T[1, n] = 20869514371$. Then, $T[2, n] = 0869514371$ and $\mathsf{pred}(T[2, n]) = 0\ 1\ 2\ 2\ 4\ 5\ 1\ 2\ 6\ 4'$. However, pred after prepending $T[1] = 2$, i.e., $\mathsf{pred}(T[1, n])$ is $0\ 0\ 2\ 3\ 2\ 5\ 5\ \ 7\ 8\ \ 6\ 4'$. Observe how the encoding changes when we go from $T[2, n]$ to $T[1, n]$. Apart from the obvious 0 in front, there are "five" other entries whose predecessor changed due to the newly inserted 2. Both earlier problems, traditional and parameterized, incurred only a constant (1 or 2) number of changes per suffix, and hence it was possible to record this information compactly. However, the number of changes here can be as large as σ, which makes it challenging and the existing techniques do not seem adequate.

Our approach: Even though many positions change, and they cannot be explicitly stored, the structural properties of this problem show that the *last point of change* (the rightmost value which changes) during LF is what matters. In the example above, the rightmost character which changes its encoding is 3 and its encoding changes from 2 to 8.

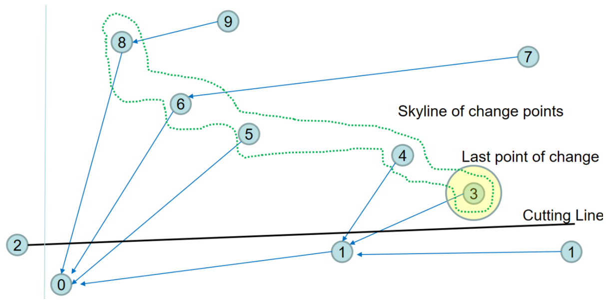

Figure 1 Geometric interpretation of the change in pred encoding of 0869514371 when prepended with 2.

The good part is that once we know this, we can deterministically pinpoint which other previous (to the left) locations changed their encoding. Thus, we can register/store one particular value and all previous changes can be captured based on that. Yet this only gives us existential dependency and not an algorithmic tool.

1.3 Our Contribution

The existing results on this topic are partial and conditional. For example, the $O(n \log\log n)$-bit by Gagie et al. [7] can answer only counting queries, that too for short patterns of size $O(\log^{O(1)} n)$. Another result by Decaroli et al. [5] is based on heuristics. We show:

▶ **Theorem 2.** *Let $T[1, n]$ be any text over an ordered alphabet of size σ. By maintaining an $O(n \log \sigma)$-bit index, we can decode any entry in the order-isomorphic suffix array of T, as well as in its inverse array, in $O(\log^2 n/ \log \sigma)$ time.*

At the heart of proving Theorem 2 lies a novel way of implementing LF mapping. We call this as LF Successor. It goes one step beyond the current approach of simulating *Suffix Array using LF mapping.*

2 Structural Properties of the Order Isomorphic Suffixes

In this section we introduce two key lemmas explaining the structural properties of the pred encoding. In other words, we see where the changes occur when a new character is prepended to the suffix. Firstly, we formally define a *change point* as follows,

▶ **Definition 3** (Change Point). *Given a string $T[r, z]$ along with its* pred *encoding* $\mathsf{pred}(T[r, z])$, *point $i \in [r, z]$ is a change point if* $\mathsf{pred}(T[r-1, z])[i-r+2] \neq \mathsf{pred}(T[r, z])[i-r+1]$.

In other words, when a character is prepended to $T[r, z]$ (making it $T[r-1, z]$) the encoding of the character $T[i]$ changes. Here point i means position in the text.

▶ **Definition 4** (Skyline). *A point i in text substring $T[r, z]$ covers a point j iff $i < j$ and $T[i] \leq T[j]$. γ-skyline of $T[r, z]$ is set of all points $i \in [r, z]$ such that $T[i] \geq \gamma$ and it is not covered by any point $j \in [r, i-1]$ such that $T[i] \geq T[j] \geq \gamma$. When $\gamma = T[r-1]$, we simply refer to this as skyline of $T[r, z]$. Given a point $d \in T[r, z]$, the skyline induced by d is same as $T[d]$-skyline of $T[r, z]$ (i.e., the one obtained by setting $\gamma = T[d]$).*

Lemma 5 proves that all the change points of $T[r, z]$ are exactly the ones that are on the *skyline* (See Figure 1 for geometric interpretation). Secondly, as mentioned earlier, although there are many change points in the order isomorphic setting, given the rightmost or last change point we can uniquely determine all the previous change points (see Figure 1). More formally, it can be stated as follows.

▶ **Lemma 5** (Skyline Lemma). *Given a text substring $T[r, z]$ and it rightmost change point d of the substring, all the change points in $T[r, z]$ can be determined based on d. These are precisely the points in $T[d]$-skyline of $T[r, z]$.*

Proof. Firstly, let's consider any change point $i \in T[r, z]$. Since its pred-encoding changes due to prepending of $T[r-1]$ the new predecessor of point i in $T[r-1, z]$ must be $r-1$ (i.e., $\mathsf{pred}(i) = i - r + 1$). This means $T[i] \geq T[r-1]$. Also if point i was covered by point j such that $j < i$ and $T[j] \geq$T[r-1], then predecessor of i in $T[r-1, z]$ would still be j.

For the other way around, consider any point i on the skyline of $T[r, z]$. The predecessor of i in $T[r, z]$ cannot be any point j such that $T[j] \geq T[r-1]$ (by definition of cover), Thus, when $T[r-1]$ gets prepended, this will become the new predecessor of i. Hence, i is a change point.

◀

Next, given two suffixes and their last common change points, all their previous change points will be the same. We state this as a lemma below. Here we define $\mathsf{rank}(x, T[r, z])$ as the number of values in $T[r, z]$ that are less than or equal to x.

▶ **Lemma 6** (Last Common Point of Change (LCPC) Lemma). *Given two text substrings $T[r, r+l-1]$ and $T[s, s+l-1]$ such that $\mathsf{pred}(T[r, r+l-1]) = \mathsf{pred}(T[s, s+l-1])$, let d be the greatest value such that $r+d-1$ and $s+d-1$ are the change points in $T[r, r+l-1]$ and $T[s, s+l-1]$ respectively. Thus, the dth point is the last common change point of substrings $T[r, r+l-1]$ and $T[s, s+l-1]$. Then for every $e \in [1, d-1]$, $r+e-1$ is a change point in $T[r, r+l-1]$ if and only if $s+e-1$ is a change point in $T[s, s+l-1]$.*

Proof. Firstly, w.l.o.g, let $\mathsf{rank}(T[r-1], T[r, r+l-1]) < \mathsf{rank}(T[s-1], T[s, s+l-1])$. Now, there is no point p such that $r < p < d$ and $\mathsf{rank}(T[r-1], T[r, r+l-1]) < \mathsf{rank}(T[p], T[r, r+l-1]) < \mathsf{rank}(T[s-1], T[s, s+l-1])$. This is because if there was such a point p, then d cannot be a change point of $T[r, r+l-1]$, because d will be covered by point p. Secondly, if $e \in [1, d-1]$ is a change point of $T[r, r+l-1]$ and suppose q was the predecessor of e before prepending of the new point, then $\mathsf{rank}(T[r+q+1], T[r, r+l-1]) < \mathsf{rank}(T[r-1], T[r, r+l-1]) < \mathsf{rank}(T[r+e+1], T[r, r+l-1])$. Therefore, we can say that $\mathsf{rank}(T[r+q+1], T[r, r+l-1]) < \mathsf{rank}(T[r-1], T[r, r+l-1]) < \mathsf{rank}(T[s-1], T[s, s+l-1]) < \mathsf{rank}(T[r+e+1], T[r, r+l-1])$. Here if we just consider the ranking orders of $T[s, s+l-1]$, then $\mathsf{rank}(T[s+q+1], T[s, s+l-1]) < \mathsf{rank}(T[s-1], T[s, s+l-1]) < \mathsf{rank}(T[s+e+1], T[s, s+l-1])$ because $\mathsf{pred}(T[r, r+l-1]) = \mathsf{pred}(T[s, s+l-1])$. This implies that $T[s-1]$ is the new predecessor of $T[s+e+1]$, which means e is also a change point of $T[s, s+l-1]$.

The encoding of characters which are not change points will stay the same in $\mathsf{pred}(T[r-1, r+d-1])$ and $\mathsf{pred}(T[s-1, s+d-1])$. On the characters which are change points, their $\mathsf{pred}(\cdot)$ values point to $T[r-1]$ (resp. $T[s-1]$). Since pred encodes distance to the predecessor character, these pred values will be the same for corresponding change points in $T[r-1, r+d-1]$ and $T[s-1, s+d-1]$. Thus, $\mathsf{pred}(\cdot)$ encoding for both agree up to the first $d+1$ characters. ◀

3 LF Successor and Order-Isomorphic Text Indexing

Recall our encoding scheme pred (Definition 1) and the lexicographic order of encoded symbols: $0 < 1 < 1' < 2 < 2' < \cdots < n-1 < (n-1)' < \$$. We will now introduce a few more terminologies related to the order-isomorphic suffix tree (ST). We shall refer to any character on any substring representing an edge label as a "point" in ST. An edge is labeled by a substring represented by that edge in ST. For any point c in ST, let $\mathsf{path}(c)$ denote the concatenation of labels from the root until c. We shall denote $\mathsf{char}(c)$ as an (pred encoded) character represented by point c. We will also refer to nodes in ST as points. In this case, the node will be represented by the character just above it (i.e., the last character of the label of its parent edge). For any point c, $\mathsf{depth}(c)$ is length of $\mathsf{path}(c)$ and $\alpha\mathsf{Depth}(c)$ = number of distinct symbols in $T[r, r + \mathsf{depth}(c) - 1]$, where $T[r, n]$ is any suffix passing through c. Note that this $\alpha\mathsf{Depth}$ indeed refers back to the original text instead of encoded text (in terms of encoded text this would be the number of non-primed characters). We call this the alphabet depth of point c. We shall generalize this notion as alphabet length for any string S as $\alpha(S)$ = number of unique alphabet symbols in S. For any two suffixes i and j (i.e., suffixes corresponding to leaves ℓ_i and ℓ_j), let point $v = \mathsf{lca}(i, j)$ be the lowest common ancestor (LCA) of ℓ_i and ℓ_j. Then, the length of longest common prefix $\mathsf{LCP}(i, j) = \mathsf{depth}(v)$ and $\alpha\mathsf{LCP}(i, j) = \alpha\mathsf{Depth}(v)$.

The locus of a pattern P is the highest node u such that $\mathsf{pred}(P)$ is a prefix of $\mathsf{path}(u)$. Every leaf ℓ_i in the sub-tree of u corresponds to an occurrence of P at a position in T given by $\mathsf{SA}[i]$. Let $[sp, ep]$ be the suffix range of P, where ℓ_{sp} (resp. ℓ_{ep}) is the leftmost (resp. rightmost) suffix in the subtree of u. We note that in order to support pattern matching, we need to **(a)** compute the suffix range $[sp, ep]$ of P and **(b)** decode suffix array values $\mathsf{SA}[i]$, $i \in [sp, ep]$. Using a standard binary search on the suffix array along with the text, we can find the suffix range. Storing $\mathsf{SA}[i]$ for every leaf ℓ_i is too costly as it will take $\Theta(n \log n)$ bits. The goal is to encode suffix array values in compact space so that they can be decoded efficiently. We show how to achieve this using a *sampled suffix array* and *LF mapping*.

Recall that LF mapping is defined as: $j = \mathsf{LF}(i)$ iff $\mathsf{SA}[j] = \mathsf{SA}[i] - 1$. We explicitly store $\mathsf{SA}[\cdot]$ values belonging to the set $\{1, 1 + \Delta, 1 + 2\Delta, \ldots, n\}$, where Δ is a tunable parameter to be set later. For any suffix i, where $\mathsf{SA}[i]$ has not been stored, we repeatedly apply LF mapping operation (starting from i) until we reach j such that $\mathsf{SA}[j]$ has been sampled. Then, $\mathsf{SA}[i] = \mathsf{SA}[j] + k$, where k is the number of LF operations applied; note that $k \leq \Delta$. Thus, we have reduced the problem to that of computing $\mathsf{LF}(\cdot)$. To this end, we introduce *LF successor*, defined as:

$$i' \text{ is called the LF-successor of } i \text{ iff } \mathsf{LF}(i') = \mathsf{LF}(i) + 1$$

We denote it as $i' = \mathsf{LFS}(i)$. Throughout this paper, we use i' to denote $\mathsf{LFS}(i)$ for any suffix i. Thus, the leaves ℓ_i and $\ell_{i'}$ are mapped by using LF operation to leaves ℓ_j and ℓ_{j+1} respectively. To compute LF mapping, we again use a sampling technique. Specifically, we explicitly store $\mathsf{LF}(\cdot)$ values in the set $\{1, 1 + \Delta, 1 + 2\Delta, \ldots, n\}$, thereby reducing the problem of computing LF mapping to that of computing at most Δ number of LF successors. In Section 4, we show how to compute LF successor in time $t_{\mathsf{LFS}} = O(\log \sigma)$ by using an $O(n \log \sigma)$-bit index. Therefore, LF can be computed in time $t_{\mathsf{LF}} = \Delta \cdot t_{\mathsf{LFS}}$ and $t_{\mathsf{SA}} = O(\Delta \cdot t_{\mathsf{LF}})$. Theorem 2 follows immediately by fixing $\Delta = \log_\sigma n$.

4 Computing LF Successor in Time $O(\log \sigma)$ Using Compact Space

In this section, we shall describe what additional information should be augmented to each leaf of the suffix tree, so that given ith leaf ℓ_i, we can quickly identify which leaf is its LF successor $\mathsf{LFS}(i)$. We shall first describe the data structure and then the query algorithm for computing $\mathsf{LFS}(i)$. We saw earlier that we will be writing SA values and LF values only for

n/Δ positions. Thus, this takes $O(n \log \sigma)$-bit space by choosing $\Delta = \log_\sigma n$. What remains to be seen is how to compute LF successor for a given suffix associated with the leaf ℓ_i. If we explicitly write it at all the leaves, it will take $\Theta(\log n)$ bits per leaf. Since there is no sampling here, this will lead to $\Theta(n \log n)$ bits which will defeat our purpose. Thus, our approach here is to store only $O(\log \sigma)$ bits of information in each leaf and yet be able to compute the LF successor quickly.

4.1 Four Cases for Suffix and its LF Successor

For the discourse in this section, we use the following terminology. Let i' be $\mathsf{LFS}(i)$. Let the starting position in the text for suffix denoted by leaf ℓ_i be r (i.e, $r = \mathsf{SA}[i]$), and that of $\ell_{i'}$ be r'. Let d denote the length of longest common prefix (LCP) of these suffixes $\mathsf{pred}(T[r,n])$ and $\mathsf{pred}(T[r',n])$. Thus, $T[r, r+d-1]$ and $T[r', r'+d-1]$ are order isomorphic. Inevitably, we will also focus on suffixes $\mathsf{LF}(i)$ and $\mathsf{LF}(i')$ which are encodings of text suffixes $T[r-1,n]$ and $T[r'-1,n]$ respectively.

Now, we distinguish two cases with respect to leaf ℓ_i (and its LF successor $\ell_{i'}$) – case (1) if $T[r-1, r+d-1]$ is not order isomorphic with $T[r'-1, r'+d-1]$, and case (2) $T[r-1, r+d-1]$ is order isomorphic with $T[r'-1, r'+d-1]$ i.e., prepending of character $T[r-1]$ (resp., $T[r'-1]$) to the left still maintains order-isomorphism until the LCP i.e., $\mathsf{pred}(T[r-1, r+d-1]) = \mathsf{pred}(T[r'-1, r'+d-1])$.

First, we shall talk about case (1). In this case, let us consider all the change points of $T[r, r+d-1]$ and $T[r', r'+d-1]$. Let e be their last common change point. If $T[r'-1] \neq T[r'+e-1]$ then we call it case (1a) - the *breakaway case*. Else, we call it case (1b) - the *equality case*. In case (1a), let g be the first change point after e for $T[r', r'+d-1]$.

We now define LF-image, which generalizes the concepts of Wiener links and LF mapping.

▶ **Definition 7** (LF-image). *Let c be any point in the suffix tree and point p above c be such that for at least one of the suffixes $T[r,n]$ passing through c, p is the last change point before c. The LF-image of c with respect to a change point p, denoted by $\mathsf{LF}(c, p, EQBT)$ is a point representing the position of* (pred *encoding of*) $T[r-1, r+\mathsf{depth}(c)-1]$*. $EQBT$ is called the equality bit and is set to 1 if p is an equality change point and 0 otherwise.*

For any such suffix i passing through c with change point p being the last one above c, $\mathsf{LF}(i)$ passes through $\mathsf{LF}(c, p, \mathrm{EQBT})$. So if $q = \mathsf{LF}(c, p, \mathrm{EQBT})$, $\mathsf{path}(q) = \mathsf{pred}(T[r-1, r+\mathsf{depth}(c)-1])$. Note that the same point c can have multiple LF-images based on which change point above c is taken as the last one and also if that is equality change point or not.

If leaf ℓ_i falls under case 2, we shall again break this case into cases (2a) and (2b). In case (2a) we consider $i < i'$ (we call this *ordered case*) and in case (2b) we consider $i' < i$ (we call this *inverting case*). We say that a suffix l *inverts* over suffix k iff $l < k$ and $\mathsf{LF}(l) > \mathsf{LF}(k)$.

▶ **Lemma 8.** *If suffixes i and $i' = \mathsf{LFS}(i)$ fall in case 2 then they have the same change points (and also the same type of change points - equality or not) until $\mathsf{lca}(i, i')$. Then i cannot have a change point immediately after $\mathsf{lca}(i, i')$. Moreover, if they fall in case (2b) then i' must have a change point immediately after $\mathsf{lca}(i, i')$.*

Proof. Let the point $c = \mathsf{lca}(i, i')$. For case (2) we know that $T[r-1, r+d-1]$ is order isomorphic with $T[r'-1, r'+d-1]$ i.e. $\mathsf{pred}(T[r-1, r+d-1]) = \mathsf{pred}(T[r'-1, r'+d-1])$. This means that i and i' have all the same change points until c.

Now let p be the last common change point of i and i' i.e. $p = \mathsf{LCPC}(i)$. Here, $\mathsf{LCPC}(i)$ denotes the last common point of change of i and its LFS i'. Additionally, suppose

$b = \mathsf{LF}(c, p, EQBT)$. So this means that $\mathsf{LF}(i)$ and $\mathsf{LF}(i')$ will pass through b. As per the definition of LF successor we know that, $\mathsf{LF}(i) < \mathsf{LF}(i')$. More specifically, $\mathsf{LF}(i') = \mathsf{LF}(i) + 1$.

Firstly, lets say that i has a change point right after c (note that both i and i' cannot change immediately after c). Now if we see all the branches under b, then $\mathsf{LF}(i)$ will fall under the rightmost branch (or just previous branch depending on whether that change point is of equality type or not). This leads to $\mathsf{LF}(i') < \mathsf{LF}(i)$ which is not possible as per the definition of LF successor. Thus, i cannot have a change point immediately after c.

Now, if we take the case (2b), then i' inverts over i because $\mathsf{LF}(i')$ must be greater than $\mathsf{LF}(i)$. For this to happen i' must have a change point immediately after c. ◀

The proof of the lemma above also leads us to the following fact.

▶ **Fact 1.** *Let c be a point immediately above any node v. Let $b = \mathsf{LF}(c, p, EQBT)$ where p is a point on $\mathsf{path}(c)$ and is the last common change point (of type equality or non-equality) for two suffixes i and $i' = \mathsf{LFS}(i)$, passing through c and lying in case 2b. Then, i' has a change point immediately after v. Moreover, there cannot be another pair of case (2b) suffixes j and $j' = \mathsf{LFS}(j)$, which have the same last common point of change p, and j' changes immediately after v.*

Proof. If any two of the suffixes i' and j', where $i' = \mathsf{LFS}(i)$ and $j' = \mathsf{LFS}(j)$, passing through v have a change point right after the node v and their last common change point is p, then under the point $b = \mathsf{LF}(c, p, EQBT)$ only one of their LF values (either $\mathsf{LF}(i')$ or $\mathsf{LF}(j')$) can be next to their respective $\mathsf{LF}(i)$ or $\mathsf{LF}(j)$. That implies only one of either $\mathsf{LF}(i') = \mathsf{LF}(i) + 1$ or $\mathsf{LF}(j') = \mathsf{LF}(j) + 1$ can be true. This is a contradiction, implying the fact is true. ◀

4.2 Storing Augmenting Information for each Leaf

We shall describe this section in terms of augmenting information stored with each leaf. However, one can easily see them as arrays that run parallel to the suffix array. We shall show that each of these augmenting fields in all the cases can be stored in $O(\log \sigma)$ bits. For each leaf ℓ_i, we can write in 2 bits which of the above 4 cases it belongs to. We denote this by $\mathsf{CASE}[i]$. We also store the same value with i' and in this case we shall call it $\overline{\mathsf{CASE}}[i']$.

If ℓ_i belongs to case (1b), then we intend to store e which we will denote as $\mathsf{LCPC}[i] = e$. Recall that e is defined as the rightmost (maximum value) common change point for $T[r, r+d-1]$ and $T[r', r'+d-1]$, and LCPC stands for *last common point of change*. Thus, LCPC is an array whose ith entry corresponds to leaf ℓ_i. However, storing the value e directly will require $\log n$ bits. Therefore, instead of e, we store number of distinct alphabet symbols in $T[r, r+e-1]$) (i.e., $\alpha(T[r, r+e-1])$). We will call this value $\alpha\mathsf{LCPC}[i]$. It is worth noting that since change points only occur at new (first occurence) alphabets in the string, e can be uniquely decoded from $\alpha\mathsf{LCPC}$. We also store a complementary array of $\alpha\mathsf{LCPC}$ denoted as $\overline{\alpha\mathsf{LCPC}}$ such that $\overline{\alpha\mathsf{LCPC}}[i'] = \alpha\mathsf{LCPC}[i]$. Thus, this value is not only stored with leaf i but also replicated in leaf $i' = \mathsf{LFS}(i)$ - albeit under a differently named field.

Recall that for case (1a), g is the first change point after e for $T[r', r'+d-1]$. For the case (1a), we store g which we call the first point of break $\mathsf{FPB}[i]$. Again, we will not store the value g directly but an encoding $\alpha(T[r, r+g-1])$ which takes $\log \sigma$ bits. We will call this value $\alpha\mathsf{FPB}[i]$. Similarly, we store this value with i' as $\overline{\alpha\mathsf{FPB}}[i'] = \alpha\mathsf{FPB}[i]$.

For the case (2a), we maintain $\alpha\mathsf{LCPC}$ and $\overline{\alpha\mathsf{LCPC}}$ as in case (1b). We also maintain an extra-bit EQBT indicating which type of change point LCPC is - whether equality change point (indicated by EQBT $= 1$) or not. Similarly, we also store $\overline{EQBT}$. We also store $\alpha(T[r, r+d-1])$ that is the number of distinct alphabet symbol occurring until $\mathsf{LCP}(i, i')$.

We shall call it αLCP$[i]$. Again, we store the same value at leaf $\ell_{i'}$ so that $\overline{\alpha\mathsf{LCP}}[i'] = \alpha\mathsf{LCP}[i]$. Additionally to this, we store FPC$[i]$ (read as *first point of change post LCA*) which in this case will be defined as the first change point of $T[r,n]$ after $T[r+d-1]$. Note that this point of change cannot be right after LCA at $T[r+d]$ because otherwise i will invert over i' (this would then be case (2b) Lemma 8) during LF mapping operation and LF(i) will be greater than LF(i'). Once again we define $\overline{\mathsf{FPC}}[i'] = \mathsf{FPC}[i]$ and define αFPC$[i]$ and $\overline{\alpha\mathsf{FPC}}[i']$ in similar vein. In summary, we maintain αLCPC, EQBT, αLCP and αFPC for each such leaf which falls in case (2a). We also store these values at their corresponding LF successors. One point to note here is that FPC, LCPC, FPB are all uniquely decodable from αFPC, αLCPC, αFPB since they necessarily fall on the new alphabet which is yet unseen in the suffix. However, the same is not true of αLCP.

As an example, let us look at $T[r-1,n] =$ `caghhfbab`... and $T[r'-1,n] =$ `cagjjebae`.... Then, pred$(T[r,n]) =$ `0111′456′2′`... and pred$(T[r',n]) =$ `0111′456′3′`.... Their LCPC is at depth 5 which is encoded as `4` in the encodings of both the suffixes. Their αLCPC $= 4$, since there are 4 distinct alphabets in both the strings until that point (4 non-prime characters in their pred encoding). Length of their LCP $= 7$, however the character `a` which occurs their as encoded character `6′` is not a new character. Hence, αLCP $= 5$ which points to character `b` in both the original strings. If we try to decode αLCP, it will lead us to position 6 rather than 7. Finally, after the LF mapping, the encoded strings are `00211′556′2′` and `00211′556′3′`.

For case (2b), our solution is more intricate so we only give a brief overview and defer details to Case (2b) section of the proof of correctness. In this case, i' inverts over i. Thus, i' has a change point right after the lca(i,i') at $T[r'+d]$. Just storing additional augmenting values to the leaves of the suffix tree is not sufficient. Like before, we shall store αLCPC and αLCP values. But we shall construct additional data structures called mini-trees and search for i' in an appropriate mini-tree identified by αLCPC and αLCP values of i. We will denote this mini-tree as $\tau_{\alpha\mathsf{LCPC}[i],\alpha\mathsf{LCP}[i]}$.

4.3 Query Algorithm

Now, we outline the pseudo-code for our query algorithm.

Computing LFS(i)

- If ℓ_i falls in case (1a),
  $\ell_{i'}$ is the unique leaf under u s.t. $\overline{\mathsf{CASE}}[i'] = \mathsf{CASE}[i]$ and $\overline{\alpha\mathsf{FPB}}[i'] = \alpha\mathsf{FPB}[i]$, where u is the highest ancestor of ℓ_i with $\alpha\mathsf{Depth}(u) \geq \alpha\mathsf{FPB}[i]$
- ElseIf ℓ_i falls in case (1b)
  $\ell_{i'}$ is unique leaf under u s.t. $\overline{\mathsf{CASE}}[i'] = \mathsf{CASE}[i]$ and $\overline{\alpha\mathsf{LCPC}}[i'] = \alpha\mathsf{LCPC}[i]$, where u is the highest ancestor of ℓ_i with $\alpha\mathsf{Depth}(u) \geq \alpha\mathsf{LCPC}[i]$
- ElseIf ℓ_i falls in case (2a)
  Let $c =$ point above FPC$[i]$ on suffix $T[r,n]$ in the suffix tree. Then $\ell_{i'}$ is leftmost leaf after ℓ_i in the (subtree of αLCP$[i]$) $\setminus$ (subtree of c) s.t. $\overline{\mathsf{CASE}}[i'] = \mathsf{CASE}[i]$, $\alpha\mathsf{LCP}[i] = \overline{\alpha\mathsf{LCP}}[i']$, $\alpha\mathsf{LCPC}[i] = \overline{\alpha\mathsf{LCPC}}[i']$ and EQBT$[i] = \overline{\mathrm{EQBT}}[i']$
- Else
  $i' = \mathsf{findSucc}(i, \alpha\mathsf{LCPC}[i], \alpha\mathsf{LCP}[i])$, which is to be defined later

Note that all the arrays mentioned above can be represented in $O(n \log \sigma)$ bits, and the implementation uses standard succinct-data-structure techniques (see Section 4.5); the difficulty lies in proving the correctness of the algorithm, which is our focus next.

4.4 Proofs of Correctness

We shall show correctness of each case. In each case, we need to ensure that we would not end up with a wrong answer. This could happen if there is another pair j, j' such that $j' = \mathsf{LFS}(j)$ and this pair shares the same characteristics with the pair i, i'. In this case, pair j, j' may interfere in the search for i' leading to false answer j'.

4.4.1 Case (1a)

Let c be the first point (the character within an edge of ST) on $\mathsf{path}(\ell_i)$ such that $T[r, r + \mathsf{depth}(c) - 1]$ has exactly $\alpha\mathsf{FPB}[i]$ distinct characters. Thus, this is the first (encoded) character where $\mathsf{pred}(T[r-1, n])$ and $\mathsf{pred}(T[r'-1, n])$ differ; in other words, $\mathsf{path}(\ell_{\mathsf{LF}(i)})$ and $\mathsf{path}(\ell_{\mathsf{LF}(i')})$ bifurcate at the position given by $\mathsf{depth}(c) + 1$. Let $\hat{c}$ be the point in ST such that $\mathsf{path}(\hat{c}) = \mathsf{pred}(T[r-1, r + \mathsf{depth}(c) - 1])$ and $\hat{c'}$ be such that $\mathsf{path}(\hat{c'}) = \mathsf{pred}(T[r'-1, r' + \mathsf{depth}(c) - 1])$. These points are on sibling edges going down from the same node. Let v be the node just above $\hat{c}$ and $\hat{c'}$. For example, consider $T[r-1, n] =$ `jeabdh`... and $T[r'-1, n] =$ `gfabdh`.... Then, $\mathsf{path}(c) = \mathsf{pred}(\texttt{eabdh}) = \mathsf{pred}(\texttt{fabdh}) = 00114$. This makes $\mathsf{path}(\hat{c}) = \mathsf{pred}(\texttt{jeabdh}) = 000114$. However, $\mathsf{path}(\hat{c'}) = \mathsf{pred}(\texttt{gfabdh}) = 000115$. Note that 5 is the highest encoded character (with an exception of 5′) which branches out of the node v.

▶ **Lemma 9.** *There is only one pair of leaves i, i' in the subtree of c, such that $\alpha\mathsf{FPB}[i] = \overline{\alpha\mathsf{FPB}}[i'] = \alpha(T[r, r + \mathsf{depth}(c) - 1])$.*

Proof. Consider LF mapping of i and i'. $\mathsf{path}(\ell_{\mathsf{LF}(i)})$ and $\mathsf{path}(\ell_{\mathsf{LF}(i')})$ first bifurcate at points $\hat{c}$ and $\hat{c'}$ respectively. Since $i' = \mathsf{LFS}(i)$, $\mathsf{char}(\hat{c}) < \mathsf{char}(\hat{c'})$. Moreover, $\mathsf{char}(\hat{c'})$ is precisely $\mathsf{depth}(c)$ or its equality version i.e. $(\mathsf{depth}(c))'$. This is the highest (encoded) character, and thus the branch with $\hat{c'}$ will be one of the two rightmost branches among branches (depending on whether the change point c for suffix i' was based on "equality" or not). However, the point $\hat{c}$ will certainly be before the two rightmost branches at v. If there was any other pair j and j' of case (1a) under the subtree of c such that $j' = \mathsf{LFS}(j)$ and $\mathsf{FPB}(j) = \mathsf{FPB}(i)$, then both $\mathsf{LF}(i')$ and $\mathsf{LF}(j')$ will fall under the subtree of $\hat{c'}$ because as per the LCPC lemma all the change points of i' and j' are the same until c (including c). On the contrary, $\mathsf{LF}(i)$ and $\mathsf{LF}(j)$ cannot fall under this subtree as they are under the subtree of $\hat{c}$. Thus, depending on whether $\mathsf{LF}(i') < \mathsf{LF}(j')$ or not, only one pair out of $(\mathsf{LF}(i), \mathsf{LF}(i'))$ or $(\mathsf{LF}(j), \mathsf{LF}(j'))$ can be adjacent. Since, i' is indeed the LF successor of i, such a pair j, j' cannot exist. ◀

4.4.2 Case (1b)

Let c be the first point in ST on $\mathsf{path}(\ell_i)$ such that $T[r, r + \mathsf{depth}(c) - 1]$ has $\alpha\mathsf{LCPC}[i]$ distinct characters. In this case, c is a change point for both i and i'. For i', it is the equality change point while for i it is not (i.e., $T[r'-1] = T[r' + \mathsf{depth}(c) - 1]$ and $T[r-1] \neq T[r + \mathsf{depth}(c) - 1]$). Let point $\hat{c}$ correspond to $\mathsf{path}(T[r-1, r + \mathsf{depth}(c) - 1])$ and $\hat{c'}$ correspond to $\mathsf{path}(T[r'-1, r' + \mathsf{depth}(c) - 1])$. Let v be the node right above $\hat{c}$ (and also $\hat{c'}$) which can be identified by $\mathsf{path}(v) = T[r-1, r + \mathsf{depth}(c) - 2]$. In this case, $\hat{c'}$ will fall in the rightmost branch at node v and $\hat{c}$ will fall in the branch previous to that. The character at point $\hat{c'}$ is precisely

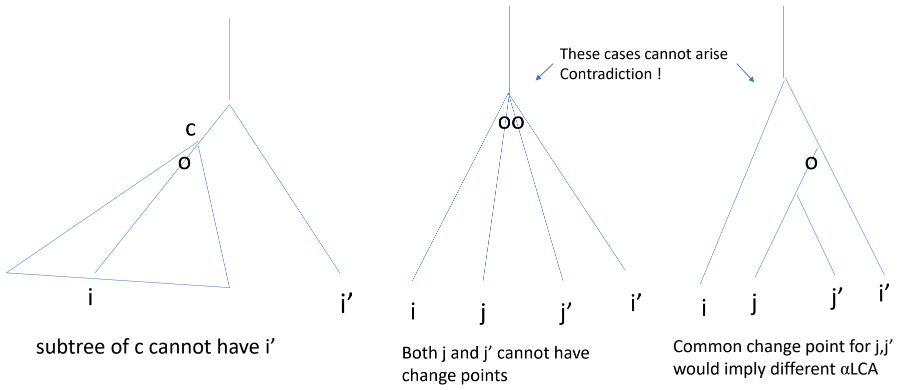

Figure 2 Illustration of case (2a)

the equality (prime) version of the character at $\hat{c}$. For example, consider $T[r-1,n] = \texttt{geabdh}\ldots$ and $T[r'-1,n] = \texttt{hfabdh}\ldots$. Then, $\mathsf{path}(c) = \mathsf{pred}(\texttt{eabdh}) = \mathsf{pred}(\texttt{fabdh}) = \mathsf{00114}$. This makes $\mathsf{path}(\hat{c}) = \mathsf{pred}(\texttt{geabdh}) = \mathsf{000115}$. However, $\mathsf{path}(\hat{c'}) = \mathsf{pred}(\texttt{hfabdh}) = \mathsf{000115'}$. Here $\mathsf{5'}$ is the highest encoded character. Again, as in the case (1a), if there were any other pair j, j' falling in case (1b) under subtree of c such that $\mathsf{LCPC}(j) = \mathsf{LCPC}(i)$, then $\mathsf{LF}(j')$ will also fall in the rightmost branch at v while $\mathsf{LF}(j)$ will fall in the previous one. Again, by applying simple interval logic as in case (1a), we can show that only one of the pairs can satisfy the LF-successor definition.

4.4.3 Case (2a)

In this case, post $\mathsf{lca}(i,i')$, branch with ℓ_i is to the left of the branch with $\ell_{i'}$. Let c be the point just above $\mathsf{FPC}[i]$. Let ℓ_k be the rightmost leaf in the subtree of c. Note that since $\mathsf{FPC}[i]$ is not immediately after the $\mathsf{lca}(i,i')$, the subtree of c does not include i'. Therefore, the order between i and i' will not be inverted after taking LF mapping. Let f be the first point in ST on suffix $T[r,n]$ such that $\alpha(\mathsf{path}(f)) = \alpha\mathsf{LCP}[i]$. The actual $\mathsf{LCP}[i]$ will be somewhere in the subtree of f because $\mathsf{LCP}[i]$ is not uniquely decodable from $\alpha\mathsf{LCP}[i]$. Here $\mathsf{LCP}[i]$ denotes the $\mathsf{lcp}(i,i')$. Let j, j' be another pair in the subtree of f such that $j' = \mathsf{LFS}(j)$ and $\alpha\mathsf{LCP}[j] = \alpha\mathsf{LCP}[i]$ and $\mathsf{LCPC}[j] = \mathsf{LCPC}[i]$. All four leaves $\mathsf{LF}(i), \mathsf{LF}(i'), \mathsf{LF}(j), \mathsf{LF}(j')$ will be in the subtree of $\hat{f}$ which is the LF-image $\mathsf{LF}(f, \mathsf{LCPC}[i], \mathrm{EQBT})$. In other words, $\hat{f}$ is the locus of $\mathsf{pred}(T[r-1, r+\mathsf{depth}(f)-1])$ in ST.

▶ **Lemma 10.** *There does not exist a pair (j, j') such that $j' = \mathsf{LFS}(j)$, $\alpha\mathsf{LCP}[j] = \alpha\mathsf{LCP}[i]$, $\alpha\mathsf{LCPC}[j] = \alpha\mathsf{LCPC}[i]$ and j' lies in between k and i'.*

Proof. Consider any other pair j, j' in the subtree of f and with the same $\alpha\mathsf{LCPC}$, EQBT and $\alpha\mathsf{LCP}$ values such that $k < j' < i'$. We will show by contradiction that such a j' cannot exist. Firstly, since $i < k < j'$ and ℓ_k being the rightmost leaf in the subtree of c, i cannot invert over j' after taking LF mapping. This is because c is the point just above $\mathsf{FPC}[i]$. Hence $\mathsf{LF}(i) < \mathsf{LF}(j')$. Also, since $\mathsf{LF}(i') = \mathsf{LF}(i) + 1$, $\mathsf{LF}(j')$ must be greater than $\mathsf{LF}(i')$. Secondly, the pair j, j' falls under case (2a) where $j < j'$ and $\mathsf{LF}(j) < \mathsf{LF}(j')$. Thus, $\mathsf{LF}(i') \le \mathsf{LF}(j) < \mathsf{LF}(j')$ which means both j and j' invert over i' after LF operation.

Next, $j < j' < i'$ means $\mathsf{lca}(j,i')$ is equal to or above $\mathsf{lca}(j',i')$. Since j and j' invert over i', it must be at $\mathsf{lca}(j,i')$ and $\mathsf{lca}(j',i')$ respectively. If $\mathsf{lca}(j,i')$ is above $\mathsf{lca}(j',i')$, then j inverts

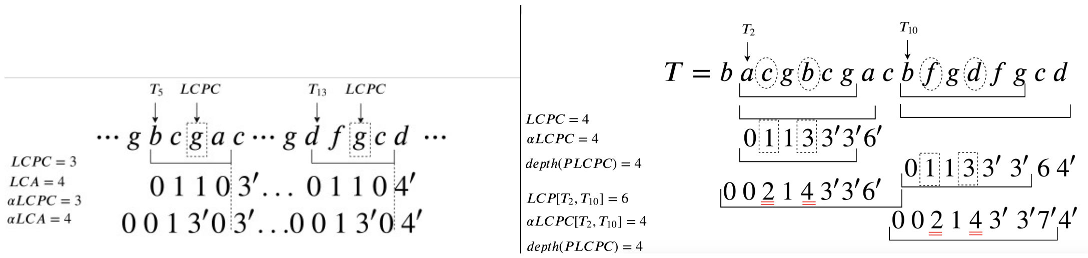

Figure 3 Illustration of case (2a) (left) and case (2b) (right). Red underline shows the character encoding that changes after taking LF.

above j' and it implies $\mathsf{LF}(j) > \mathsf{LF}(j')$ which is a contradiction. Now if $\mathsf{lca}(j, i') = \mathsf{lca}(j', i')$, then there are two cases. The first case is where j and j' invert from a common branch connecting path of i'. Here, j and j' will have a common change point at this branch which is post $\mathsf{lca}(j', i')$. It implies that there is another common change point for j, j' which leads to $\mathsf{LCPC}[j] > \mathsf{LCPC}[i]$ (a contradiction). In the second case, j and j' branch out at $\mathsf{lca}(j', i')$ but fall in different branches. However, according to Lemma 8, only one of j or j' can have a change point right after the $\mathsf{lca}(i, i')$. Hence, this case also leads to contradiction. Thus, j' does not lie in between k and i' (See Figure 3). ◀

4.4.4 Case (2b)

For the case (2b), we know that suffix i' comes before suffix i in the suffix tree, i.e. $i' < i$. Additionally, for the case (2b), i' has a change point right after the node representing the $\mathsf{lca}(i, i')$. Moreover, under $\mathsf{lca}(i, i')$ the branch containing the suffix i' will be the only one that will have a change point tied with the same LCPC (See Fact 1). Since $i' = \mathsf{LFS}(i)$, after the LF mapping i' will invert over i making $\mathsf{LF}(i') = \mathsf{LF}(i) + 1$.

As mentioned in Section 4.2, for the case (2b) we store $\alpha\mathsf{LCPC}[i]$ and $\alpha\mathsf{LCP}[i]$ values for each leaf ℓ_i as augmenting information. Additionally, we store their complements $\overline{\alpha\mathsf{LCPC}}[i']$ and $\overline{\alpha\mathsf{LCP}}[i']$ for each leaf $\ell_{i'}$. Now we consider an additional data structure called mini-trees that will help us in finding i' given i. Specifically, a particular mini-tree $\tau_{a,b}$ has set of all leaves ℓ_i and their corresponding LF successors $\ell_{i'}$ from the suffix tree that has $\alpha\mathsf{LCPC}[i] = \overline{\alpha\mathsf{LCPC}}[i'] = a$ and $\alpha\mathsf{LCP}[i] = \overline{\alpha\mathsf{LCP}}[i'] = b$. A particular leaf ℓ_i will not be in any mini-tree if that leaf does not fall under the case (2b). Thus, a leaf can be present in a mini-tree if it falls under case (2b) or it is an LF-successor of some other leaf which falls under the case (2b). Therefore, each leaf in the suffix tree will be in at most two mini-trees and some mini-trees are possibly empty. In other words, a mini-tree is a compacted subtrie of the suffix tree containing only those leaves selected for that mini-tree. Hence, overall size of all the mini-trees combined is $O(n)$.

To draw a correspondence between the leaves of the suffix tree and the leaves of the mini-trees, we use a bit-vector $B[1, n]$, where $B[i] = 1$ iff leaf i falls in case (2b) or leaf i is an LF-successor of the leaf which falls in case (2b). In other words, $B[i] = 1$ if a leaf from the suffix tree is present in at least one of the mini-trees, and $B[i] = 0$ otherwise. Next, we create two character vectors C and $\overline{C}$ as follows. If $B[i] = 0$, then $C[i] = \overline{C}[i] = 0$. Otherwise,

1. $C[i]$ stores an encoding of the pair $\alpha\mathsf{LCPC}[i], \alpha\mathsf{LCP}[i]$ as a combined character from an alphabet of size σ^2; essentially $C[i] = (\sigma - 1) \cdot \alpha\mathsf{LCPC}[i] + \alpha\mathsf{LCP}[i]$
2. $\overline{C}[i] = -C[i]$ if $\alpha\mathsf{LCPC}[i] = \overline{\alpha\mathsf{LCPC}}[i]$ and $\alpha\mathsf{LCP}[i] = \overline{\alpha\mathsf{LCP}}[i]$, and $\overline{C}[i] = (\sigma - 1) \cdot \overline{\alpha\mathsf{LCPC}}[i] + \overline{\alpha\mathsf{LCP}}[i]$ otherwise.

Now given a particular leaf ℓ_i in the suffix tree, for finding the corresponding leaf in the mini-tree, we first check if $B[i] = 1$. Since $a = \alpha\mathsf{LCPC}[i]$ and $b = \alpha\mathsf{LCP}[i]$, we can quickly identify the mini-tree $\tau_{a,b}$ it belongs to as augmenting information $\alpha\mathsf{LCPC}[i]$ and $\alpha\mathsf{LCP}[i]$ is stored for the leaf ℓ_i. To find out which leaf in $\tau_{a,b}$ corresponds to ℓ_i, all we have to do is figure out the number of leaves $j \leq i$ that satisfy $a = \alpha\mathsf{LCPC}[j] = a$ and $b = \alpha\mathsf{LCP}[j]$ or $\overline{\alpha\mathsf{LCPC}}[j] = a$ and $\overline{\alpha\mathsf{LCP}}[j] = b$; this is the same as the number of entries $j \leq i$ in the character vectors C such that $C[j] = C[i]$ plus the number of entries $k \leq i$ in the character vectors $\overline{C}$ such that $\overline{C}[k] = C[i]$. This is because the mini-tree is just a compacted subtrie of the original suffix tree consisting of only those leaves present in a particular mini-tree. To map a leaf from the mini-tree back to the leaf of the original suffix tree, we need to store a character vector for each mini-tree over the leaves of the mini-tree. Let $C_{a,b}$ be the character vector for the mini-tree $\tau_{a,b}$. This character array indicates whether the leaf has $a = \alpha\mathsf{LCPC}[i]$ and $b = \alpha\mathsf{LCP}[i]$ or $a = \overline{\alpha\mathsf{LCPC}}[i]$ and $b = \overline{\alpha\mathsf{LCP}}[i]$ or both. In other words, it simply specifies how the leaf was selected for that mini-tree using techniques similar to that described above. It is to be noted that all character vectors combined need $O(n \log \sigma)$ bits.

4.4.4.1 Identifying i'

We know that $\alpha\mathsf{LCPC}[i] = a$ and $\alpha\mathsf{LCP}[i] = b$. Let p_a be the first point in suffix tree where $\alpha(T[r + \mathsf{depth}(p_a) - 1])) = a$ and p_b be the first point such that $\alpha(T[r + \mathsf{depth}(p_b) - 1]) = b$. Thus, p_a and p_b are the points in suffix tree where $\alpha\mathsf{LCPC}[i]$ and $\alpha\mathsf{LCP}[i]$ are located. Note that p_a is above or the same as p_b. Now consider the mini-tree $\tau_{a,b}$. Let another pair j, j' where $j' = \mathsf{LFS}(j)$ fall under the same mini-tree (i.e., ℓ_j and ℓ'_j are also descendants of p_b and $\alpha\mathsf{LCPC}[j] = \alpha\mathsf{LCPC}[i]$ and $\alpha\mathsf{LCP}[j] = \alpha\mathsf{LCP}[i]$). Here j' will be on the left of j because they fall under the case (2b). We will focus here on searching i' as the first qualifying leaf to the left of i. Another pair j, j' could interfere with our process of searching if j' falls between i' and i. Formally, we say

▶ **Definition 11.** *A pair j, j'* interferes *with i, i' if $i' < j' < i$ and $\alpha\mathsf{LCPC}[j] = \alpha\mathsf{LCPC}[i]$ and $\alpha\mathsf{LCP}[j] = \alpha\mathsf{LCP}[i]$. Here, $i' = \mathsf{LFS}(i)$ and $j' = \mathsf{LFS}(j)$*

There are two cases of 'interference' that can occur with respect to these two pairs – case (2b′) is where both j' and j are in between i' and i i.e. $i' < j' < j < i$ and case (2b*) where j is on the right of i i.e. $i' < j' < i < j$. As we know that $\alpha\mathsf{LCPC}[i] = \alpha\mathsf{LCPC}[j] = a$ and p_a is the first point in the suffix tree where $\alpha(T[r + \mathsf{depth}(p_a) - 1])) = a$. Suppose $x = \mathsf{LF}(\mathsf{lca}(i, i'), p_a, EQBT)$ and $y = \mathsf{LF}(\mathsf{lca}(j, j'), p_a, EQBT)$. Here $EQBT$ is set to 1 if i' has an equality change point and 0 otherwise. Now in the case (2b′), after taking LF-mapping, j' inverts over j under y and i' inverts over all three of j, j', i under x – we call this the *nested case*. In case (2b*), j' and i both together (maintaining same order) invert over j under y and then i' inverts over all of them under x – we call this the *bulk-invert case*. Additionally, we will need to augment this mini-tree further so that we can distinguish the pair i, i' from the pair j, j'.

▶ **Lemma 12.** *If a pair j, j' interferes with i, i', then $\mathsf{lca}(i', i)$ occurs above $\mathsf{lca}(j', j)$ in the suffix tree. Additionally, if $i < j$, then $\mathsf{lca}(j', i)$ is below $\mathsf{lca}(j, j')$.*

Proof. Note that in bulk invert case since j' and i both invert together over j, $\mathsf{lca}(j', i)$ must be below $\mathsf{lca}(j, j')$. Even though $\alpha\mathsf{LCP}[i] = \alpha\mathsf{LCP}[j]$, it cannot happen that LCAs of both the pairs are on the same node in the suffix tree (i.e. $\mathsf{lca}(i', i) = \mathsf{lca}(j', j)$). This is because from any node only one branch can have a change point at the next character below the node (see Fact 1). But we know that i' has a change point just below the node representing

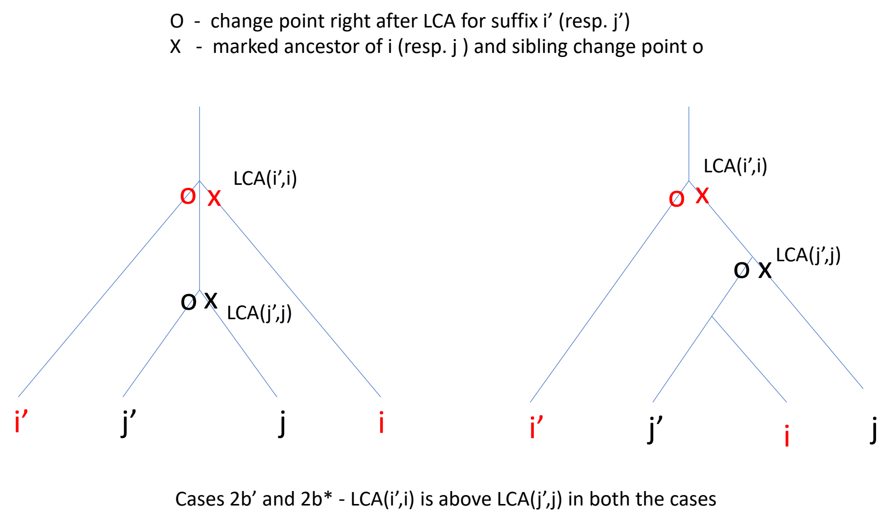

Figure 4 Mini-trees for case (2b)

$\mathsf{lca}(i, i')$. Therefore, the branch containing j' cannot have a change point just below that node. This implies $j' \neq \mathsf{LFS}(j)$ since j falls under the case (2b). This holds a contradiction.

Therefore, for the case (2b′), it must be the case that $\mathsf{lca}(j', j)$ is below $\mathsf{lca}(i', i)$, implying that suffixes j' and j belong to the subtree at $\mathsf{lca}(i', i)$. In case (2b*), it cannot happen that $\mathsf{lca}(i', i)$ is below $\mathsf{lca}(j', j)$ because that would mean j' has a change point right below $\mathsf{lca}(j', j)$ which falls above $\mathsf{lca}(i', i)$. This would make $\alpha\mathsf{LCPC}[i]$ different than $\alpha\mathsf{LCPC}[j]$ because the suffixes i and i' will have an extra change point above $\mathsf{lca}(i, i')$ and below the $\mathsf{lca}(j, j')$. Hence, for the case (2b*) this leads to a contradiction and $\mathsf{lca}(j, j')$ cannot be above the $\mathsf{lca}(i, i')$. ◀

If $\mathsf{lca}(i', i)$ and $\mathsf{lca}(j', j)$ are not on the same root-to-leaf path (neither above nor below nor same as each other), then pairs i, i' and j, j' are non-interfering. So we need not consider that case as in some sense for i, our algorithm looks at the closest suffix to the left of i that has the same $\alpha\mathsf{LCP}$ and $\alpha\mathsf{LCPC}$ as the qualifying suffix for $\mathsf{LFS}(i)$.

Finally, from Fact 1 we can say that there exists a unique suffix i' marked with case (2b) under the point at $1 + \mathsf{depth}(\mathsf{lca}(i', i))$ depth such that $\alpha\mathsf{LCP}[i] = \overline{\alpha\mathsf{LCP}}[i']$ and $\alpha\mathsf{LCPC}[i] = \overline{\alpha\mathsf{LCPC}}[i']$, with the constraint that i' has a change point at $1 + \mathsf{depth}(\mathsf{lca}(i', i))$ depth.

4.4.4.2 Searching in Minitree

For any i, if we can identify $\mathsf{lca}(i', i)$ precisely, then i' is the leaf which has the same $\overline{\alpha\mathsf{LCPC}}$ and $\overline{\alpha\mathsf{LCP}}$ values (as that of i) and i' is in the subtree of a branch of $\mathsf{lca}(i', i)$ whose leading character in that branch is a change point. For this, we mark some nodes in the tree. More precisely, for each mini-tree, we mark a node v if a point at $(\mathsf{depth}(\mathsf{parent}(v)) + 1)$ depth is a change point for a suffix i' (in case (2b)) in the subtree of v. Note that only one child of a node can get marked (refer to Fact 1). Also note that there is only one marked node in a path from the root to a leaf because if there were another marked node w for a suffix j', then $\alpha\mathsf{LCPC}[i'] \neq \alpha\mathsf{LCPC}[j']$. But we know that all the leaves in a mini-trie have the same $\alpha\mathsf{LCPC}[i], \alpha\mathsf{LCP}$ (or their complement) values.

Now lets say that a node x in the mini tree $\tau_{\alpha\mathsf{LCPC}[i],\alpha\mathsf{LCP}[i]}$ is the node corresponding to $\mathsf{lca}(i', i)$ in the suffix tree. Therefore, given i, our task simply becomes locating the leaf ℓ in the mini-tree that corresponds to i. Then, find the lowest ancestor of ℓ that has a marked child before ℓ in pre-order; observe that this lowest ancestor is precisely the node x corresponding to $\mathsf{lca}(i', i)$. Let y be the marked child of x. Within the subtree of y, we can find the unique leaf ℓ' corresponding to i', which can be mapped back to the original suffix tree. To find this unique leaf, we store a unary encoding at the marked node indicating which leaf we looking for; more precisely, if the desired leaf is the z^{th} leftmost leaf under the marked node, then store z in unary at the marked node. Since there is only one marked node from a leaf to root path in a mini-tree, the total length of all such unary encodings combined is bounded by the size of the mini-tree. The mapping to and from the suffix tree to a mini-tree can be carried out using the bit-vector and the character vectors defined earlier.

For the sake of completion, we summarize the discussion in the following findSucc method, which was used by pseudo-code in Section 4.3.

findSucc(i, a, b)

- Use the bit-vector B and the character vectors C and $\overline{C}$ to identify the leaf ℓ in $\tau_{a,b}$ that corresponds to ℓ_i
- Find the lowest ancestor x of ℓ that has a marked child y before x in pre-order
- Use the unary encoding stored at y to locate the leaf ℓ' in $\tau_{a,b}$ corresponding to $\ell_{i'}$
- Finally, use the character vector $C_{a,b}$ to map ℓ' back to i'

4.5 Implementation and Complexity Analysis

We will rely on the following well-known data structures of Fact 2 and Fact 3.

▶ **Fact 2** (Wavelet Tree [11]). *Given an array $A[1, t]$ over Σ, by using a $t \log |\Sigma| + o(t \log |\Sigma|)$-bit structure, we can compute the following in $O(\log |\Sigma|)$ time:*

- $A[i]$
- $\mathsf{rank}_A(i, x)$ = *number of occurrences of x in $A[1, i]$*
- $\mathsf{select}_A(i, x)$ = *i-th occurrence of x in A*
- $\mathsf{prevValue}_A(i, y)$ = *rightmost position $j < i$ such that $A[j] \le y$*

We drop the subscript A when the context is clear.

▶ **Fact 3** (Fully-Functional Succinct Tree [19]). *The topology of order-isomorphic suffix tree can be encoded in $O(n)$ bits to support the following operations in $O(1)$ time.*

- $\mathsf{pre\text{-}order}(u)/\mathsf{post\text{-}order}(u)$: *pre-order/post-order rank of node u*
- $\mathsf{parent}(u)$: *parent of node u*
- $\mathsf{nodeDepth}(u)$: *number of edges on the path from the root to u*
- $\mathsf{child}(u, q)$: *qth leftmost child of node u*
- $\mathsf{sibRank}(u)$: *number of children of $\mathsf{parent}(u)$ to the left of u*
- $\mathsf{lca}(u, v)$: *lowest common ancestor (LCA) of two nodes u and v*
- $\mathsf{sp}(u)/\mathsf{ep}(u)$: *leftmost/rightmost leaf in the subtree of u*
- $\mathsf{levelAncestor}(u, d)$: *ancestor of u such that $\mathsf{nodeDepth}(u) = d$*

Moving forward, we assume that any array has been pre-processed using Fact 2. We maintain the topology of the order-isomorphic suffix tree and the mini-trees (Case 2b) using Fact 3. Finally, we explicitly store $\alpha\mathsf{Depth}(u)$ for every node u in the order-isomorphic suffix tree. For the purpose of locating the node immediately below FPB or LCPC, we will rely on the following lemma.

▶ **Lemma 13.** *By maintaining 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 $\alpha\mathsf{Depth}(w) \geq W$ in $O(\log \sigma)$ time.*

Proof. Create an array A such that $A[k] = \alpha\mathsf{Depth}(w)$, where w is the node with pre-order rank k. Maintain A as a wavelet tree. Given ℓ_i, find the rightmost entry $r < \mathsf{pre\text{-}order}(\ell_i)$ in A such that $A[r] < W$ using $\mathsf{prevValue}_A(\mathsf{pre\text{-}order}(\ell_i), W-1)$. Let $v' = \mathsf{lca}(\ell_i, v)$, where v is the node with pre-order rank r. Then, $w = \mathsf{levelAncestor}(\ell_i, \mathsf{nodeDepth}(v')+1)$. To see why this is correct, observe that $\alpha\mathsf{Depth}(v') \leq \alpha\mathsf{Depth}(v) < W$. If $\alpha\mathsf{Depth}(w) < W$, the prevValue-query should have returned w instead of v (since $\mathsf{pre\text{-}order}(v) < \mathsf{pre\text{-}order}(w) \leq \mathsf{pre\text{-}order}(\ell_i)$). ◀

4.5.1 Case (1a) and Case (1b)

In case (1a), i' is the only leaf marked with case (1a) in the sub-tree of $\mathsf{FPB}(i)$ that satisfies $\overline{\alpha\mathsf{FPB}}[i'] = \alpha\mathsf{FPB}[i]$. The first task is to find the subtree of $\mathsf{FPB}(i)$, i.e., the node just below $\mathsf{FPB}(i)$. This node, say v, can be found in $O(\log \sigma)$ time using Lemma 13 and by using $\alpha\mathsf{FPB}[i]$. Within the subtree of v, we simply find the only leaf i' marked with 1a such that $\overline{\mathsf{FPB}}[i'] = \mathsf{FPB}[i]$ using Fact 2. Since $\alpha\mathsf{FPB}$ and $\overline{\alpha\mathsf{FPB}}$ entries for case (1a) suffixes are at least one, in order to identify a valid case (1a) suffix, we simply set the $\alpha\mathsf{FPB}$ and $\overline{\alpha\mathsf{FPB}}$ entries for non case (1a) suffixes to zero.

In case (1b), the idea is the same, with the difference that we use $\alpha\mathsf{LCPC}$ and $\overline{\alpha\mathsf{LCPC}}$ arrays (instead of FPB and $\alpha\mathsf{FPB}$ arrays) for finding the node v and then i'. As in the previous case, we set the $\alpha\mathsf{LCPC}$ and $\overline{\alpha\mathsf{LCPC}}$ entries for non case (1b) suffixes to zero.

Note that the wavelet trees for the four arrays need $O(n \log \sigma)$ bits, and a wavelet tree query needs $O(\log \sigma)$ time.

4.5.2 Case (2a)

Let c be the point just above $\mathsf{FPC}[i]$. Let ℓ_k be the rightmost leaf in the subtree of c. By Lemma 10, it is evident that i' is the leftmost leaf such that $i' > k$, $\overline{\alpha\mathsf{LCP}}[i'] = \alpha\mathsf{LCP}[i]$, $\overline{\alpha\mathsf{LCPC}}[i'] = \alpha\mathsf{LCPC}[i]$, and $\overline{\mathrm{EQBT}}[i'] = \mathrm{EQBT}[i]$. To properly identify a case (2a) suffix, we maintain a summary vector X defined as follows. For any suffix i lying in case (2a), $X[i] = (\sigma-1)\cdot\alpha\mathsf{LCP}[i] + \alpha\mathsf{LCPC}[i]$ if $\mathrm{EQBT}[i] = 1$, and $X[i] = -(\sigma-1)\cdot\alpha\mathsf{LCP}[i] - \alpha\mathsf{LCPC}[i]$ if $\mathrm{EQBT}[i] = 0$. For any suffix j not in case (2a), we let $X[i] = 0$. Likewise, we define $\overline{X}$ based on $\overline{\alpha\mathsf{LCP}}$, $\overline{\alpha\mathsf{LCPC}}$, and $\overline{\mathrm{EQBT}}$.

Note that any entry in X and $\overline{X}$ is from the set $[0, 2\sigma^2]$; hence, a wavelet over them needs $O(n \log \sigma)$ bits and supports queries in $O(\log \sigma)$ time. Thus, if we can find out the leaf ℓ_k, we can locate i' by using the wavelet-tree over the two summary vectors X and $\overline{X}$ in additional $O(\log \sigma)$ time.

To find ℓ_k, we use Lemma 13 and $\alpha\mathsf{FPB}$ to first find the highest node v such that $\alpha\mathsf{Depth}(v) \geq \alpha\mathsf{FPB}[i]$. Note that ℓ_k is the rightmost leaf in the subtree of $\mathsf{parent}(v)$ if $\mathsf{FPB}[i]$ is the first character of the edge on which it lies, and is the rightmost leaf in the subtree of v otherwise. We explicitly store a bit-vector to distinguish between the cases. Using these, ℓ_k is located in $O(\log \sigma)$ time.

4.5.3 Case (2b)

In our previous discussion, we have already addressed how to map a case (2b) leaf i in the suffix tree to its corresponding leaf in the mini-tree. (Refer to Section 4.4.4.) We have also addressed that given the desired marked node (corresponding to i) in the mini-tree, how we can find the leaf in the mini-tree corresponding to the LF-successor i'. Finally, we also know how to map-back to i' from the mini-tree. Note that all of these can be achieved by storing the character vectors and the bit vector as a wavelet tree, and by using a succinct encoding of the mini trees. What is left to discuss is how to find the marked node. To this end, we present Lemma 14. Using this we can find the desired marked node in $O(1)$ time given the leaf corresponding to i in the mini-tree.

▶ **Lemma 14.** *Consider a tree having t nodes, where each non-leaf node has at least two children. Also, each node is marked or unmarked. By using an $O(t)$-bit data structure, given a leaf x, in $O(1)$ time, we can find the rightmost leaf $y < x$ such that the child of* $\mathsf{lca}(y, x)$ *on the path to y is marked.*

Proof. Let u be a node. We *associate* 1 with u iff $\mathsf{parent}(u)$ has a child v before u in pre-order, where v is marked. Pre-process the tree with Lemmas 15 and 16.

Given the query x, use Lemma 15 to locate the lowest ancestor u of x associated with a 1. We find the marked sibling v of u to its left using Lemma 16. The time needed is $O(1)$. ◀

▶ **Lemma 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 lca 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. Since each node in τ_u is associated with a 0 or 1, the number of possible trees is at most 2^g (because τ_u has fewer than g non-leaf nodes). We store a pointer from u to τ_u. The total space needed for storing all pointers is $O(\frac{t}{g} \log 2^g) = O(t)$ 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 $O(g \log g) = O(g \log \log t)$ bits. Therefore, the total space for all such trees is $O(2^g g \log \log t)$. By choosing $c = 1/2$, 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 $u^* = \mathsf{lca}(1+g\lfloor k/g \rfloor, \max\{t, g(1+\lfloor k/g \rfloor)\})$ of ℓ_k. 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 lowest ancestor of ℓ_k associated with a 1 is given by $\mathsf{levelAncestor}(\ell_k, D)$. ◀

▶ **Lemma 16.** *Consider a tree of t nodes, where some nodes are marked. 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 marked 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 the i^{th} leftmost child of w, given by $\mathsf{child}(w, i)$, is marked. The total 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 $r = \mathsf{rank}_{B_{v'}}(\mathsf{sibRank}(v), 1)$. If $r = 0$, then u does not exist; otherwise, $u = \mathsf{child}(v', \mathsf{select}_{B_{v'}}(r, 1))$. ◀

References

1. 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. doi:10.1145/167088.167115.
2. M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.
3. Richard Cole and Ramesh Hariharan. Faster suffix tree construction with missing suffix links. *SIAM J. Comput.*, 33(1):26–42, 2003. *An extended abstract appeared in STOC 2000.* doi:10.1137/S0097539701424465.
4. Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Marcin Kubica, Alessio Langiu, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Order-preserving indexing. *Theor. Comput. Sci.*, 638:122–135, 2016. doi:10.1016/j.tcs.2015.06.050.
5. Gianni Decaroli, Travis Gagie, and Giovanni Manzini. A compact index for order-preserving pattern matching. In *2017 Data Compression Conference, DCC 2017, Snowbird, UT, USA, April 4-7, 2017*, pages 72–81, 2017. doi:10.1109/DCC.2017.35.
6. Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005. *An extended abstract appeared in FOCS 2000 under the title "Opportunistic Data Structures with Applications".* doi:10.1145/1082036.1082039.
7. Travis Gagie, Giovanni Manzini, and Rossano Venturini. An encoding for order-preserving matching. In *25th Annual European Symposium on Algorithms, ESA 2017, September 4-6, 2017, Vienna, Austria*, pages 38:1–38:15, 2017. doi:10.4230/LIPIcs.ESA.2017.38.
8. Arnab Ganguly, Wing-Kai Hon, Kunihiko Sadakane, Rahul Shah, Sharma V. Thankachan, and Yilin Yang. A framework for designing space-efficient dictionaries for parameterized and order-preserving matching. *Theor. Comput. Sci.*, 854:52–62, 2021. doi:10.1016/j.tcs.2020.11.036.
9. 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.
10. Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. Structural pattern matching - succinctly. In Yoshio Okamoto and Takeshi Tokuyama, editors, *28th International Symposium on Algorithms and Computation, ISAAC 2017, December 9-12, 2017, Phuket, Thailand*, volume 92 of *LIPIcs*, pages 35:1–35:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ISAAC.2017.35.
11. 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.
12. 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. *An extended abstract appeared in STOC 2000.* doi:10.1137/S0097539702402354.
13. 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. doi:10.1016/j.tcs.2013.10.006.
14. Sung-Hwan Kim and Hwan-Gue Cho. Simpler fm-index for parameterized string matching. *Inf. Process. Lett.*, 165:106026, 2021. doi:10.1016/j.ipl.2020.106026.
15. Marcin Kubica, Tomasz Kulczyński, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. A linear time algorithm for consecutive permutation pattern matching. *Information Processing Letters*, 113(12):430–433, 2013.
16. Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. volume 22, pages 935–948, 1993. doi:10.1137/0222058.

17 Juan Mendivelso, Sharma V. Thankachan, and Yoan J. Pinzón. A brief history of parameterized matching problems. *Discret. Appl. Math.*, 274:103–115, 2020. doi:10.1016/j.dam.2018.07.017.

18 Gonzalo Navarro. *Compact data structures: A practical approach*. Cambridge University Press, 2016.

19 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.* doi:10.1145/2601073.

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

21 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. doi:10.1109/SWAT.1973.13.