A Textbook Solution for Dynamic Strings

Zsuzsanna Lipták
Dipartimento di Informatica, University of Verona, Italy

Francesco Masillo
Dipartimento di Informatica, University of Verona, Italy

Gonzalo Navarro
Center for Biotechnology and Bioengineering (CeBiB)
Department of Computer Science, University of Chile, Chile

Abstract

We consider the problem of maintaining a collection of strings while efficiently supporting splits and concatenations on them, as well as comparing two substrings, and computing the longest common prefix between two suffixes. This problem can be solved in optimal time $\mathcal{O}(\log N)$ whp for the updates and $\mathcal{O}(1)$ worst-case time for the queries, where N is the total collection size [Gawrychowski et al., SODA 2018]. We present here a much simpler solution based on a forest of enhanced splay trees (FeST), where both the updates and the substring comparison take $\mathcal{O}(\log n)$ amortized time, n being the lengths of the strings involved. The longest common prefix of length ℓ is computed in $\mathcal{O}(\log n + \log^2 \ell)$ amortized time. Our query results are correct whp. Our simpler solution enables other more general updates in $\mathcal{O}(\log n)$ amortized time, such as reversing a substring and/or mapping its symbols. We can also regard substrings as circular or as their omega extension.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases dynamic strings, splay trees, dynamic data structures, LCP, circular strings

Digital Object Identifier 10.4230/LIPIcs..2024.

Funding *Zsuzsanna Lipták*: Partially funded by the MUR PRIN project Nr. 2022YRB97K 'PINC' (Pangenome INformatiCs. From Theory to Applications) and by the INdAM-GNCS Project CUP_E53C23001670001 (Compressione, indicizzazione, analisi e confronto di dati biologici).
Gonzalo Navarro: Funded by Basal Funds FB0001, Mideplan, Chile, and Fondecyt Grant 1-230755, Chile.

1 Introduction

Consider the problem in which we have to maintain a collection of *dynamic strings*, that is, strings we want to modify over time. The modifications may be edit operations such as insertion, deletion, or substitution of a single character; inserting or deleting an entire substring (possibly creating a new string from the deleted substring); adding a fresh string to the collection; etc. In terms of queries, we may want to retrieve a symbol or substring of a dynamic string, determine whether two substrings from anywhere in the collection are equal, or even determine the longest prefix shared by two suffixes in the collection (LCP). The collection must be maintained in such a way that both updates and queries have little cost.

This setup is known in general as the *dynamic strings* problem. A partial and fairly straightforward solution are the so-called ropes, or cords [7]. These are binary trees[1] where the leaves store short substrings, whose left-to-right concatenation forms the string. Ropes were introduced for the Cedar programming language to speed up handling very long strings; a C implementation (termed cords) was also given in the same paper [7]. As the motivating application of ropes/cords was that of implementing a text editor, they support edit operations and extraction/insertion of substrings to enable fast typing and cut&paste, as well as retrieving substrings, but do not support queries like substring equality or LCPs. The trees must be periodically rebalanced to maintain logarithmic times. Recently, a modified version of ropes was implemented for the Ruby language as a basic data type [39]. This variant supports the same updates but does not give any theoretical guarantee.

The first solution we know of that enables equality tests, by Sundar and Tarjan [47], supports splitting and concatenating whole sequences, and whole-string equality in constant time, with updates taking $\mathcal{O}(\sqrt{N \log m} + \log m)$ amortized time, where N is the total length of all the strings in the collection and m is the number of updates so far. It is easy to see that these three primitives encompass all the operations and queries above, except for LCP (substring retrieval is often implicit). The update complexity was soon improved by Mehlhorn et al. [38] to $\mathcal{O}(\log^2 N)$ expected time with a randomized data structure, and $\mathcal{O}(\log N(\log m \log^* m + \log N))$ worst-case time with a deterministic one. The deterministic time complexity was later improved by Alstrup et al. [1] to $\mathcal{O}(\log N \log^* N)$ (which holds with high probability, whp), also computing LCPs in $\mathcal{O}(\log N)$ worst-case time. Recently, Gawrychowski et al. [23, 24] obtained $\mathcal{O}(\log N)$ update time whp, retaining constant time to compare substrings, and also decreasing the LCP time to constant, among many other results. They also showed that the problem is essentially closed because just updates and substring equality require $\Omega(\log N)$ time even if allowing amortization. Nishimoto et al. [41, 42] showed how to compute LCPs in worst-case time $\mathcal{O}(\log N + \log \ell \log^* N)$, where ℓ is the LCP length, while inserting/deleting substrings of length ℓ in worst-case time $\mathcal{O}((\ell + \log N \log^* N)\frac{(\log\log N)^2}{\log\log\log N})$.

All these results build on the idea of parsing a string hierarchically by consistently cutting it into blocks, giving unique names to the blocks, and passing the sequence of names to the next level of parsing. The string is then represented by a parse tree of logarithmic height, whose root consists of a single name, which can be compared to the name at the root of another substring to determine string equality. While there is a general consensus on the fact that those solutions are overly complicated, Gawrychowski et al. [24] mention that

"We note that it is very simple to achieve $\mathcal{O}(\log n)$ update time [...], if we allow the

[1] The authors [7] actually state that they are DAGs and referring to them as binary trees is just a simplification. The reason is that the nodes can have more than one parent, so subtrees may be shared.

equality queries to give an incorrect result with polynomially small probability. We represent every string by a balanced search tree with characters in the leaves and every node storing a fingerprint of the sequence represented by its descendant leaves. However, it is not clear how to make the answers always correct in this approach [...]. Furthermore, it seems that both computing the longest common prefix of two strings of length n and comparing them lexicographically requires $\Omega(\log^2 n)$ *time in this approach."*

This suggestion, indeed, connects to the original idea of ropes [7]. Cardinal and Iacono [12] built on the suggestion to develop a kind of tree dubbed "Data Dependent Tree (DDT)", which enables updates and LCP computation in $\mathcal{O}(\log N)$ *expected amortized* time, yet with no errors. DDTs eliminate the chance of errors by ensuring that the fingerprints have no collisions—they simply rebuild all DDTs for all strings in the collection, using a new hash function, when this low-probability event occurs—and reduce the LCP complexity to $\mathcal{O}(\log N)$ by ensuring that subtrees representing the same string have the same shape (so one can descend in the subtrees of both strings synchronously).

In this paper we build on the same suggestion [24], but explore the use of another kind of tree—an enhanced splay tree—which yields a beautifully simple yet powerful data structure for maintaining dynamic string collections. We obtain logarithmic *amortized* update times for most operations (our cost to compute LCPs lies between logarithmic and squared-logarithmic, see later) and our queries return correct answers whp. The ease of implementation of splay trees makes our solution attractive to be included in a textbook for undergraduate students.

An important consequence of using simpler data structures is that our space usage is $\mathcal{O}(N)$, whereas the solutions based on parsings require in addition $\mathcal{O}(\log N)$ space per update performed, as each one adds a new path to the parse tree. Since the previous parse tree is still available, those structures are *persistent*: one can access any previous version. Our solution is not persistent in principle, but we can make it persistent using $\mathcal{O}(\log n)$ extra space per update or query made so far (we cannot make direct use of the techniques of Driscoll et al. [19]). This adds only $\mathcal{O}(1)$ amortized time to the operations.

It would not be hard to obtain *worst-case* times instead of amortized ones, by choosing AVL, α-balanced, or other trees that guarantee logarithmic height. One can indeed find the use of such binary trees for representing strings in the literature [44, 16, 22]. Our solution using splay trees has the key advantage of being very simple and easy to understand. The basic operations of splitting and concatenating strings, using worst-case balanced trees, imply attaching and detaching many subtrees, plus careful rebalancing, which is a nightmare to explain and implement.[2] Knuth, for example, considered them too complicated to include in his book [34, p. 473] *"Deletion, concatenation, etc. It is possible to do many other things to balanced trees and maintain the balance, but the algorithms are sufficiently lengthly that the details are beyond the scope of this book."* Instead, he says [34, p. 478] *"A much simpler self-adjusting data structure called a splay tree was developed subsequently [...] Splay trees, like the other kinds of balanced trees already mentioned, support the operations of concatenation and splitting as well as insertion and deletion, and in a particularly simple way."*

Our contribution. We use a splay tree [45], enhanced with additional information, to represent each string in the collection, where all the nodes contain string symbols and Karp-Rabin-like fingerprints [30, 40] of the symbols in their subtree. We refer to our data structure as a *forest of enhanced splay trees*, or FeST. As we will see, we can create new

[2] As an example, an efficient implementation [33] of Rytter's AVL grammar [44] has over 10,000 lines of C++ code considering only their "basic" variant.

strings in $\mathcal{O}(n)$ time, extract substrings of length ℓ in $\mathcal{O}(\ell + \log n)$ time, perform updates and (correctly whp) compare substrings in $\mathcal{O}(\log n)$ time, where n is the length of the strings involved—as opposed to the total length N of all the strings—and the times are amortized (the linear terms are also worst-case). Further, we can compute LCPs correctly whp in amortized time $\mathcal{O}(\log n + \log^2 \ell)$, where ℓ is the length of the returned LCP.

While our LCP time is $\mathcal{O}(\log^2 n)$ for long enough ℓ, LCPs are usually much shorter than the suffixes. For example, in considerably general probabilistic models [48], the maximum LCP value between *any* distinct suffixes of two strings of length n is almost surely $\mathcal{O}(\log n)$, in which case our algorithm runs in $\mathcal{O}(\log n)$ amortized time.

The versatility of our FeST data structure allows us to easily support other kinds of operations, such as reversing or complementing substrings, or both. We can thus implement the reverse complementation of a substring in a DNA or RNA sequence, whereby the substring is reversed and each character is replaced by its Watson-Crick complement. Substring reversal alone is used in classic problems on genome rearrangements where genomes are represented as sequences of genes, and have to be sorted by reversals (see, e.g., [50, 6, 10, 11, 43, 13], to cite just a few). Note that chromosomes can be viewed either as permutations or as strings, when gene duplication is taken into account, see Fertin et al. [20]; our FeST data structure accommodates both. We can also implement signed reversals [28, 27], another model of evolutionary operation used in genome rearrangements. In general, we can combine reversals with any involution on the alphabet, of which signed or Watson-Crick complementation are only examples. In order to support these operations in $\mathcal{O}(\log n)$ amortized time, we only need to add new constant-space annotations, further enhancing our splay trees while retaining the running times for the other operations. The obvious solution of maintaining modified copies of the strings (e.g., reversed, complemented, etc.) is less attractive in practice due to the extra space and time needed to store and update all the copies.

Operations supported. We maintain a collection of strings of total length N in $\mathcal{O}(N)$ space, and support the following operations, where we distinguish the basic string data type from dynamic strings (all times are amortized). We have not chosen a minimal set of primitives because reducing to primitives entails considerable performance overheads in practice, even if the asymptotic time complexities are not altered.

- `make-string`(w) creates a dynamic string s from a basic string w, in $\mathcal{O}(|s|)$ time.
- `access`(s, i) returns the symbol $s[i]$ in $\mathcal{O}(\log |s|)$ time.
- `retrieve`(s, i, j) returns the basic string $w[1..j - i + 1] = s[i..j]$, in $\mathcal{O}(|w| + \log |s|)$ time.
- `substitute`(s, i, c), `insert`(s, i, c), and `delete`(s, i) perform the basic edit operations on s: substituting $s[i]$ by character c, inserting c at $s[i]$, and deleting $s[i]$, respectively, all in $\mathcal{O}(\log |s|)$ time. For appending c at the end of s one can use `insert`$(s, |s| + 1, c)$.
- `introduce`(s_1, i, s_2) inserts s_2 at position i of s_1 (for $1 \leq i \leq |s_1| + 1$), converting s_1 to $s_1[..i - 1] \cdot s_2 \cdot s_1[i..]$ and destroying s_2, in $\mathcal{O}(\log |s_1 s_2|)$ time.
- `extract`(s, i, j) creates dynamic string $s' = s[i..j]$, removing it from s, in $\mathcal{O}(\log |s|)$ time.
- `equal`$(s_1, i_1, s_2, i_2, \ell)$ determines the equality of substrings $s_1[i_1..i_1 + \ell - 1]$ and $s_2[i_2..i_2 + \ell - 1]$ in $\mathcal{O}(\log |s_1 s_2|)$ time, correctly whp.
- `lcp`(s_1, i_1, s_2, i_2) computes the length ℓ of the longest common prefix between suffixes $s_1[i_1..]$ and $s_2[i_2..]$, in $\mathcal{O}(\log |s_1 s_2| + \log^2 \ell)$ time, correctly whp, and also tells which suffix is lexicographically smaller.
- `reverse`(s, i, j) reverses the substring $s[i..j]$ of s, in $\mathcal{O}(\log |s|)$ time.
- `map`(s, i, j) applies a fixed involution (a symbol mapping that is its own inverse) to all the symbols of $s[i..j]$, in $\mathcal{O}(\log |s|)$ time.

Our data structure also enables easy implementation of other features, such as handling circular strings. This is an important and emerging topic [5, 15, 25, 26, 29], as many current sequence collections, in particular in computational biology, consist of circular rather than linear strings. Recent data structures built for circular strings [8, 9], based on the extended Burrows-Wheeler Transform (eBWT) [37], avoid the detour via the linearization and handle the circular input strings directly. Finally, FeST also allows queries on the omega extensions of strings, that is, on the infinite concatenation $s^\omega = s \cdot s \cdot s \cdots$. These occur, for example, in the context of the eBWT, which is based on the so-called omega-order. In Section 5 we will sketch how to handle circular strings and the omega extension of strings; a detailed description will be given in the full version of the paper.

2 Basic concepts

Strings. We use array-based notation for strings, indexing from 1, so a string s is a finite sequence over a finite ordered alphabet Σ, written $s = s[1..n] = s[1]s[2]\cdots s[n]$, for some $n \geq 0$. We assume that the alphabet Σ is integer. The length of s is denoted $|s|$, and ε denotes the *empty string*, the unique string of length 0. For $1 \leq i, j \leq |s|$, we write $s[i..j] = s[i]s[i+1]\cdots s[j]$ for the *substring* from i to j, where $s[i..j] = \varepsilon$ if $i > j$. We write *prefixes* as $s[..i] = s[1..i]$ and *suffixes* as $s[i..] = s[i..|s|]$. Given two strings s, t, their concatenation is written $s \cdot t$ or simply st, and s^k denotes the k-fold concatenation of s, with $s^0 = \varepsilon$. A substring (prefix, suffix) of s is called *proper* if it does not equal s.

The *longest common prefix* (LCP) of two strings s and t is defined as the longest string u that is both a prefix of s and t, and $\mathrm{lcp}(s, t) = |u|$ as its length. One can define the lexicographic order based on the lcp: $s <_{\mathrm{lex}} t$ if either s is a proper prefix of t, or otherwise if $s[\ell + 1] < t[\ell + 1]$, where $\ell = \mathrm{lcp}(s, t)$.

Splay trees. The *splay tree* [45] is a binary search tree that guarantees that a sequence of insertions, deletions, and node accesses costs $\mathcal{O}(\log n)$ amortized time per operation on a tree of n nodes that starts initially empty. In addition, splay trees support splitting and joining trees, both in $\mathcal{O}(\log n)$ amortized time, where n is the total number of nodes involved in the operation.

The basic operation of the splay tree is called *splay*(x), which moves a tree node x to the root by a sequence of primitive rotations called zig, zig-zig, zig-zag, and their symmetric versions. Let $x(A, B)$ denote a tree rooted at x with left and right subtrees A and B, then the rotation zig-zig converts $z(y(x(A, B), C), D)$ into $x(A, y(B, z(C, D))$, while the rotation zig-zag converts $z(y(A, x(B, C)), D)$ into $x(y(A, B), z(C, D))$. Whether zig-zig or zig-zag (or their symmetric variant) is applied to x depends on its relative position w.r.t. its grandparent. Note that both of these operations are composed by two edge rotations. Finally, operation zig, which is only applied if x is a child of the root, converts $y(x(A, B), C)$ into $x(A, y(B, C))$.

Every access or update on the tree is followed by a *splay* on the deepest reached node. In particular, after finding a node x in a downward traversal, we do *splay*(x) to make x the tree root. The goal is that the costs of all the operations are proportional to the cost of all the related *splay* operations performed, so we can focus on analyzing only the splays. Many of the splay tree properties can be derived from a general "access lemma" [45, Lem. 1].

▶ **Lemma 1** (Access Lemma [45]). *Let us assign any positive weight $w(x)$ to the nodes x of a splay tree T, and define $sw(x)$ as the sum of the weights of all the nodes in the subtree rooted at x. Then, the amortized time to splay x is $\mathcal{O}(\log(W/sw(x))) \subseteq \mathcal{O}(\log(W/w(x)))$, where $W = \sum_{x \in T} w(x)$.*

The result is obtained by defining $r(x) = \log sw(x)$ (all our logarithms are in base 2) and $\Phi(T) = \sum_{x\in T} r(x)$ as the potential function for the splay tree T. If we choose $w(x) = 1$ for all x, then $W = n$ on a splay tree of n nodes, and thus we obtain $\mathcal{O}(\log n)$ amortized cost for each operation. By choosing other functions $w(x)$, one can prove other properties of splay trees like static optimality, the static finger property, and the working set property [45].

The update operations supported by splay trees include inserting new nodes, deleting nodes, joining two trees (where all the nodes in the second tree go to the right of the nodes in the first tree), and splitting a tree into two at some node (where all the nodes to its right become a second tree). The times of those operations are ruled by the "balance theorem with updates" [45, Thm. 6].

▶ **Lemma 2** (Balance Theorem with Updates [45]). *Any sequence of access, insert, delete, join and split operations on a collection of initially empty splay trees has an amortized cost of $\mathcal{O}(\log n)$ per operation, where n is the size of the tree(s) where the operation is carried out.*

This theorem is proved with the potential function that assigns $w(x) = 1$ to every node x. Note the theorem considers a forest of splay trees, whose potential function is the sum of the functions $\Phi(T)$ over the trees T in the forest. For details, see the original paper [45].

Karp-Rabin fingerprinting. Our queries will be correct "with high probability" (whp), meaning a probability of at least $1 - 1/N^c$ for an arbitrarily large constant c, where N is the total size of the collection. This will come from the use of a variant of the original Karp-Rabin fingerprint [30] (cf. [40]) defined as follows. Let $[1..a]$ be the alphabet of our strings and $p \geq a$ a prime number. We choose a random base b uniformly from $[1..p-1]$. The fingerprint κ of string $s[1..n]$ is defined as $\kappa(s) = \left(\sum_{i=0}^{n-1} s[n-i]\cdot b^i\right) \bmod p$. We say that two strings $s \neq s'$ of the same length n collide through κ if $\kappa(s) = \kappa(s')$, that is, $\kappa(s'') = 0$ where $s'' = s - s'$ is the string defined by $s''[i] = (s[i] - s'[i]) \bmod p$. Since $\kappa(s'')$ is a polynomial, in the variable b, of degree at most $n-1$ over the field $\mathbb{Z}_p$, it has at most $n-1$ roots. The probability of a collision between two strings of length n is then bounded by $(n-1)/(p-1)$ because b is uniformly chosen in $[1..p-1]$. By choosing $p \in \Theta(N^{c+1})$ for any desired constant c, we obtain that κ is collision-free on any $s \neq s'$ whp. We will actually choose $p \in \Theta(N^{c+2})$ because some of our operations perform $\mathcal{O}(\operatorname{polylog} N)$ string comparisons, not just one. Since N varies over time, we can use instead a fixed upper bound, like the total amount of main memory. We use the RAM machine model where logical and arithmetic operations on $\Theta(\log N)$ machine words take constant time.

Two fingerprints $\kappa(s)$ and $\kappa(s')$ can then be composed in constant time to form $\kappa(s'\cdot s) = (\kappa(s')\cdot b^{|s|} + \kappa(s)) \bmod p$. To avoid the $\mathcal{O}(\log |s|)$ time for modular exponentiation, we will maintain the value $b^{|s|} \bmod p$ together with $\kappa(s)$. The corresponding value for $s'\cdot s$ is $(b^{|s'|}\cdot b^{|s|}) \bmod p$, so we can maintain those powers in constant time upon concatenations.

3 Our data structure and standard operations

In this section we describe our data structure called FeST (for Forest of enhanced Splay Trees), composed of a collection of (enhanced) splay trees, and then show how the traditional operations on dynamic strings are carried out on it.

3.1 The data structure

We will use a FeST for maintaining the collection of strings, one splay tree per string. A dynamic string $s[1..n]$ is encoded in a splay tree with n nodes such that $s[k]$ is stored in the node x with in-order k (the in-order of a node is the position in which it is listed if we recursively traverse first the left subtree, then the node, and finally the right subtree). We will say that node x *represents* the substring $s[i..j]$, where $[i..j]$ is the range of the in-orders of all the nodes in the subtree rooted at x. Let T be the splay tree representing string s, then for $1 \le i \le |s|$, we call $\mathit{node}(i)$ the node with in-order i, and for a node x of T, we call $\mathit{pos}(x)$ the in-order of node x. The root of T is denoted $\mathit{root}(T)$.

For the amortized analysis of our FeST, our potential function Φ will be the sum of the potential functions $\Phi(T)$ over all the splay trees T representing our string collection. The collection starts initially empty, with $\Phi = 0$. New strings are added to the collection with `make-string`; then edited with `substitute`, `insert`, and `delete`, and redistributed with `introduce` and `extract`.

Information stored at nodes. A node x of the splay tree representing $s[i..j]$ will contain pointers to its left and right children, called x.left and x.right, its symbol $x.\text{char} = s[\mathit{pos}(x)]$, its subtree size $x.\text{size} = j - i + 1$, its fingerprint $x.\text{fp} = \kappa(s[i..j])$, and the value $x.\text{power} = b^{j-i+1} \bmod p$. These fields are recomputed in constant time whenever a node x acquires new children x.left and/or x.right (e.g., during the splay rotations) with the following formulas: (1) $x.\text{size} = x.\text{left.size} + 1 + x.\text{right.size}$, (2) $x.\text{fp} = ((x.\text{left.fp} \cdot b + x.\text{char}) \cdot x.\text{right.power} + x.\text{right.fp}) \bmod p$, and (3) $x.\text{power} = (x.\text{left.power} \cdot b \cdot x.\text{right.power}) \bmod p$, as explained in Section 2. For the formula to be complete when the left and/or right child is *null*, we assume $\mathit{null}.\text{size} = 0$, $\mathit{null}.\text{fp} = 0$, and $\mathit{null}.\text{power} = 1$. We will later incorporate other fields.

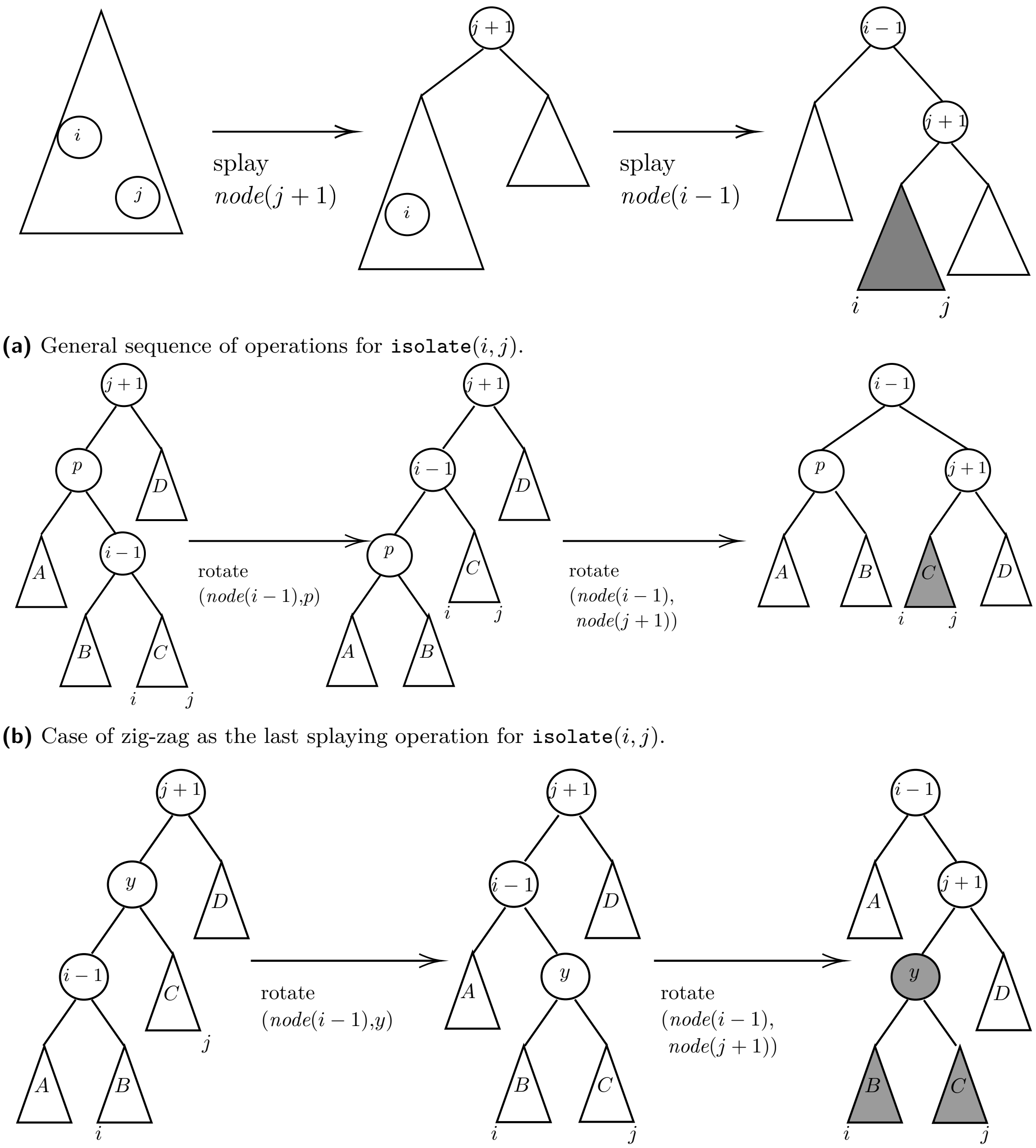
Subtree sizes allow us identify $\mathit{node}(i)$ given i, in the splay tree T representing string s, in $\mathcal{O}(\log |s|)$ amortized time. This means we can answer $\texttt{access}(s, i)$ in $\mathcal{O}(\log |s|)$ amortized time, since $s[i] = \mathit{node}(i).\text{char}$. Finding $\mathit{node}(i)$ is done in the usual way, with the recursive function $\texttt{find}(i) = \texttt{find}(\mathit{root}(T), i)$ that returns the ith smallest element in the subtree rooted at the given node. More precisely, $\texttt{find}(x, i) = x$ if $i = x.\text{left.size} + 1$, $\texttt{find}(x, i) = \texttt{find}(x.\text{left}, i)$ if $i < x.\text{left.size} + 1$, and $\texttt{find}(x, i) = \texttt{find}(x.\text{right}, i - (x.\text{left.size} + 1))$ if $i > x.\text{left.size} + 1$. To obtain logarithmic amortized time, `find` splays the node it returns, thus $\mathit{pos}(\mathit{root}(T)) = i$ holds after calling $\texttt{find}(\mathit{root}(T), i)$.

Isolating substrings. We will make use of another primitive we call $\texttt{isolate}(i, j)$, for $1 \le i, j \le |s|$ and $i \le j + 1$, on a tree T representing string s. This operation rearranges T in such a way that $s[i..j]$ becomes represented by one subtree, and returns this subtree's root y.

If $i = 1$ and $j = n$, then $y = \mathit{root}(T)$ and we are done. If $i = 1$ and $j < n$, then we find (and splay) $\mathit{node}(j+1)$ using $\texttt{find}(j+1)$; this will move $\mathit{node}(j+1)$ to the root, and $s[i..j]$ will be represented by the left subtree of the root, so $y = \mathit{root}(T).\text{left}$. Similarly, if $1 < i$ and $j = n$, then we perform $\texttt{find}(i-1)$, so $\mathit{node}(i-1)$ is splayed to the root and $s[i..j]$ is represented by the right subtree of the root, thus $y = \mathit{root}(T).\text{right}$.

Finally, if $1 < i, j < n$, then splaying first $\mathit{node}(j+1)$ and then $\mathit{node}(i-1)$ will typically result in $\mathit{node}(i-1)$ being the root and $\mathit{node}(j+1)$ its right child, thus the left subtree of $\mathit{node}(j+1)$ contains $s[i..j]$, that is, $y = \mathit{root}(T).\text{right.left}$. The only exception arises if the last splay operation on $\mathit{node}(i-1)$ is a zig-zig, as in this case $\mathit{node}(j+1)$ would become a grandchild, not a child, of the root. Therefore, in this case, we modify the last splay operation: if $\mathit{node}(i-1)$ is a grandchild of the root and a zig-zig must be applied, we perform instead two consecutive zig operations on $\mathit{node}(i-1)$ in a bottom-up manner, that is, we first rotate the edge between $\mathit{node}(i-1)$ and its parent, and then the edge between $\mathit{node}(i-1)$ and its new parent (former grandparent), see Fig. 1.

We now consider the effect of the modified zig-zig operation on the potential. In the proof of Lemma 1 [45, Lem. 1], Sleator and Tarjan show that the zig-zig and the zig-zag cases contribute $3(r'(x)-r(x))$ to the amortized cost, where $r'(x)$ is the new value of $r(x)$ after the operation. The sum then telescopes to $3(r(t)-r(x)) = 3\log(sw(t)/sw(x))$ along an upward path towards a root node t. The zig rotation, instead, contributes $1+r'(x)-r(x)$, where the 1 would be problematic if it was not applied only once in the path. Our new zig-zig may, at most one time in the path, cost like two zig's, $2+2(r'(x)-r(x))$, which raises the cost bound of the whole splay operation from $1+3\log(sw(t)/sw(x))$ to $2+3\log(sw(t)/sw(x))$. This retains the amortized complexity, that is, the amortized time for `isolate` is $\mathcal{O}(\log|s|)$.



(a) General sequence of operations for `isolate`(i,j).

(b) Case of zig-zag as the last splaying operation for `isolate`(i,j).

(c) Case of the modified zig-zig as the last splaying operation for `isolate`(i,j).

Figure 1 Scheme of the `isolate`(i,j) operation applied on a splay tree. Subfigures 1b and 1c show two cases of the last splay operation of `isolate`(i,j), yielding a single (shaded) subtree that represents the substring $s[i..j]$.

3.2 Creating a new dynamic string

Given a basic string $w[1..n]$, operation `make-string`(w) creates a new dynamic string $s[1..n]$ with the same content as w, which is added to the FeST. While this can be accomplished in $\mathcal{O}(n \log n)$ amortized time via successive `insert` operations on an initially empty string, we describe a "bulk-loading" technique that achieves linear worst-case (and amortized) time.

The idea is to create, in $\mathcal{O}(n)$ time, a perfectly balanced splay tree using the standard recursive procedure. As we show in the next lemma, this shape of the tree adds only $\mathcal{O}(n)$ to the potential function, and therefore the amortized time of this procedure is also $\mathcal{O}(n)$.

▶ **Lemma 3.** *The potential $\Phi(T)$ of a perfectly balanced splay tree T with n nodes is at most $2n + \mathcal{O}(\log^2 n) \subseteq \mathcal{O}(n)$.*

Proof. Let d be the depth of the deepest leaves in a perfectly balanced binary tree, and call $l = d - d' + 1$ the *level* of any node of depth d'. It is easy to see that there are at most $1 + n/2^l$ subtrees of level l. Those subtrees have at most $2^l - 1$ nodes. Separating the sum $\Phi(T) = \sum_{x \in T} r(x)$ by levels l and using the bound $sw(x) < 2^l$ if x is of level l, we get $\Phi(T) < \sum_{l=1}^{\log n} \left(1 + \frac{n}{2^l}\right) \log 2^l = 2n + \mathcal{O}(\log^2 n)$. ◀

Once the tree is created and the fields x.char are assigned in in-order, we perform a post-order traversal to compute the other fields. This is done in constant time per node using the formulas given in Section 3.1.

3.3 Retrieving a substring

Given a string s in the FeST and two indices $1 \leq i \leq j \leq |s|$, operation `retrieve`(s, i, j) extracts the substring $s[i..j]$ and returns it as a basic string. The special case $i = j$ is given by `access`(s, i), which finds *node*(i), splays it, and returns *root*(T).char, recall Section 3.1. If $i < j$, we perform $y =$ `isolate`(i, j) and then we return $s[i..j]$ with an in-order traversal of the subtree rooted at y. Overall, the operation `retrieve`(s, i, j) takes $\mathcal{O}(\log |s|)$ amortized time for `isolate`, and then $\mathcal{O}(j - i + 1)$ worst case time for the traversal of the subtree.

3.4 Edit operations

Let s be a string in the FeST, i an index of s, and c a character. The simplest edit operation, `substitute`(s, i, c) writes c at $s[i]$, that is, s becomes $s' = s[..i-1] \cdot c \cdot s[i+1..]$. It is implemented by doing `find`(i) in the splay tree T of s, in $\mathcal{O}(\log |s|)$ amortized time. After the operation, *node*(i) is the root, so we set *root*(T).char $= c$ and recompute (only) its fingerprint as explained in Section 3.1.

Now consider operation `insert`(s, i, c), which converts s into $s' = s[..i-1] \cdot c \cdot s[i..]$. This corresponds to the standard insertion of a node in the splay tree, at in-order position i. We first use `find`(i) in order to make $x =$ *node*(i) the tree root, and then create a new root node y, with y.left $= x$.left and y.right $= x$. We then set x.left $=$ *null* and recompute the other fields of x as shown in Section 3.1. Finally, we set y.char $= c$ and also compute its other fields. By Lemma 2, the amortized cost for an insertion is $\mathcal{O}(\log |s|)$.

Finally, the operation `delete`(s, i) converts s into $s' = s[..i-1] \cdot s[i+1..]$. This corresponds to standard deletion in the splay tree: we first do `find`(i) in the tree T of s, so that $x =$ *node*(i) becomes the root, and then join the splay trees of x.left and x.right, isolating the root node x and freeing it. The joined tree now represents s'; the amortized cost is $\mathcal{O}(\log |s|)$.

3.5 Introducing and extracting substrings

Given two strings s_1 and s_2 represented by trees T_1 and T_2 in the FeST, and an insertion position i in s_1, operation $\texttt{introduce}(s_1, i, s_2)$ generates a new string $s = s_1[..i-1] \cdot s_2 \cdot s_1[i..]$ (the original strings are not anymore available). We implement this operation by first doing $y = \texttt{isolate}(i, i-1)$ on the tree T_1. Note that in this case y will be a *null* node, whose in-order position is between $i-1$ and i. We then replace this null node by (the root of) the tree T_2. As shown in Section 3.1, the node y that we replace has at most two ancestors in T_1, say x_1 (the root) and x_2. We must then recompute the fields of x_2 and then of x_1.

Apart from the $\mathcal{O}(\log |s_1|)$ amortized time for isolate, the other operations take constant time. We must consider the change in the potential introduced by connecting T_2 to T_1. In the potential Φ, the summands $\log sw(x_1)$ and $\log sw(x_2)$ will increase to $\log(sw(x_1) + |s_2|)$ and $\log(sw(x_2) + |s_2|)$, thus the increase is $\mathcal{O}(\log |s_2|)$. The total amortized time is thus $\mathcal{O}(\log |s_1| + \log |s_2|) = \mathcal{O}(\log |s_1 s_2|)$.

Let s be a string represented by tree T in the FeST and $i \leq j$ indices in s. Function $\texttt{extract}(s, i, j)$ removes $s[i..j]$ from s and creates a new dynamic string s' from it. This can be carried out by first doing $y = \texttt{isolate}(i, j)$ on T, then detaching y from its parent in T to make it the root of the tree that will represent s', and finally recomputing the fields of the (former) ancestors x_2 and x_1 of y. The change in potential is negative, as $\log sw(x_1)$ and $\log sw(x_2)$ decrease by up to $\mathcal{O}(\log(j - i + 1))$. The total amortized time is then $\mathcal{O}(\log |s|)$.

3.6 Substring equality

Let $s_1[i_1..i_1 + \ell - 1]$ and $s_2[i_2..i_2 + \ell - 1]$ be two substrings, where possibly $s_1 = s_2$. Per Section 2, we can compute equal whp by comparing $\kappa(s_1[i_1..i_1+\ell-1])$ and $\kappa(s_2[i_2..i_2+\ell-1])$. We compute $y_1 = \texttt{isolate}(i_1, i_1 + \ell - 1)$ on the tree of s_1 and $y_2 = \texttt{isolate}(i_2, i_2 + \ell - 1)$ on the tree of s_2. Once node y_1 represents $s_1[i_1..i_1 + \ell - 1]$ and y_2 represents $s_2[i_2..i_2 + \ell - 1]$, we compare $y_1.\mathrm{fp} = \kappa(s_1[i_1..i_1 + \ell - 1])$ with $y_2.\mathrm{fp} = \kappa(s_2[i_2..i_2 + \ell - 1])$.

The splay operations take $\mathcal{O}(\log |s_1 s_2|)$ amortized time, while the comparison of the fingerprints takes constant time and returns the correct answer whp. Note this is a one-sided error; if the method answers negatively, the strings are distinct.

4 Extended operations

In this section we consider less standard operations of dynamic strings, including the computation of LCPs and others we have not seen addressed before.

4.1 Longest common prefixes

Operation $\texttt{lcp}(s_1, i_1, s_2, i_2)$ computes $\mathrm{lcp}(s_1[i_1..], s_2[i_2..])$ correctly whp, by exponentially searching for the maximum value ℓ such that $s_1[i_1..i_1 + \ell - 1] = s_2[i_2..i_2 + \ell - 1]$. The exponential search requires $\mathcal{O}(\log \ell)$ equality tests, which are done using equal operations. The amortized cost of this basic solution is then $\mathcal{O}(\log |s_1 s_2| \log \ell)$; we now improve it.

We note that all the accesses the exponential search performs in s_1 and s_2 are at distance $\mathcal{O}(\ell)$ from $s_1[i_1]$ and $s_2[i_2]$. We could then use the dynamic finger property [18] to show, with some care, that the amortized time is $\mathcal{O}(\log |s_1 s_2| + \log^2 \ell)$. This property, however, uses a different mechanism of potential functions where trees cannot be joined or split.[3] We

[3] The static finger property cannot be used either, because we need new fingers every time an LCP is

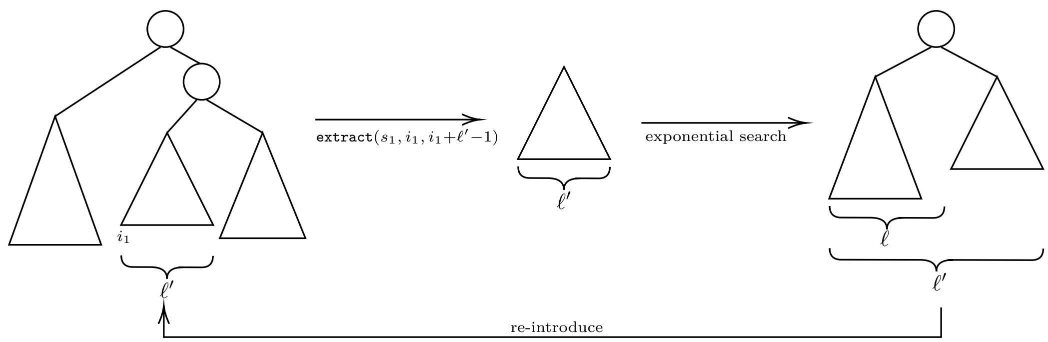


Figure 2 Scheme of operations for `lcp` shown on one of the two strings.

then use an alternative approach. The main idea is that, if we could bound ℓ beforehand, we could isolate those areas so that the accesses inside them would cost $\mathcal{O}(\log \ell)$ and then we could reach the desired amortized time. Bounding ℓ in less than $\mathcal{O}(\log \ell)$ accesses (i.e., $\mathcal{O}(\log |s_1 s_2| \log \ell)$ time) is challenging, however. Assuming for now that $s_1 \neq s_2$ (we later handle the case $s_1 = s_2$), our plan is as follows (see Fig. 2):

1. Find a (crude) upper bound $\ell' \geq \ell$.
2. Extract substrings $s_1' = s_1[i_1..i_1 + \ell' - 1]$ and $s_2' = s_2[i_2..s_2 + \ell' - 1]$.
3. Run the basic exponential search for ℓ between $s_1'[1..]$ and $s_2'[1..]$.
4. Reinsert substrings s_1' and s_2' into s_1 and s_2.

Steps 2 and 4 are carried out in $\mathcal{O}(\log |s_1 s_2|)$ amortized time using the operations `extract` and `introduce`, respectively. Step 3 will still require $\mathcal{O}(\log \ell)$ substring comparisons, but since they will be carried out on the shorter substrings s_1' and s_2', they will take $\mathcal{O}(\log \ell \log \ell')$ amortized time. The main challenge is to balance the cost to find ℓ' in Step 1 with the quality of the approximation of ℓ' so that $\log \ell'$ is not much larger than $\log \ell$.

Consider the following strategy for Step 1. Let $n = |s_1 s_2|$ and $n' = \min(|s_1| - i_1 + 1, |s_2| - i_2 + 1)$. We first check a few border cases that we handle in $\mathcal{O}(\log n)$ amortized time: if $s_1[i_1..i_1 + n' - 1] = s_2[i_2..i_2 + n' - 1]$ we finish with the answer $\ell = n'$, or else if $s_1[i_1..i_1 + 1] \neq s_2[i_2..i_2 + 1]$ we finish with the answer $\ell = 0$ or $\ell = 1$. Otherwise, we define the sequence $\ell_0 = 2$ and $\ell_i = \min(n', \ell_{i-1}^2)$ and try out the values ℓ_i for $i = 1, 2, \ldots$, until we obtain $s_1[i_1..i_1 + \ell_i - 1] \neq s_2[i_2..i_2 + \ell_i - 1]$. This implies that $\ell_{i-1} \leq \ell < \ell_i$, so we can use $\ell' = \ell_i \leq \ell^2$. This yields $\mathcal{O}(\log \ell \log \ell') = \mathcal{O}(\log^2 \ell)$ amortized time for Step 3. On the other hand, since $\ell \geq \ell_{i-1} = 2^{2^{i-1}}$, it holds $i \leq 1 + \log\log \ell$. Since each of the i values is tried out in $\mathcal{O}(\log n)$ time with `equal`, the amortized cost of Step 1 is $\mathcal{O}(\log n \log\log \ell)$ and the total cost to compute `lcp` is $\mathcal{O}(\log n \log\log \ell + \log^2 \ell)$. In particular, this is $\mathcal{O}(\log^2 \ell)$ when ℓ is large enough, $\log \ell = \Omega(\sqrt{\log n \log\log n})$.

Hitting twice. To obtain our desired time $\mathcal{O}(\log n + \log^2 \ell)$ for every value of $\log \ell$, we will apply our general strategy twice. First, we will set $\ell'' = 2^{\log^{2/3} n}$ and determine whether $s_1[i_1..i_1 + \ell'' - 1] = s_2[i_2..i_2 + \ell'' - 1]$. If they are equal, then $\log \ell = \Omega(\log^{2/3} n)$ and we can apply the strategy of the previous paragraph verbatim, obtaining amortized time $\mathcal{O}(\log^2 \ell)$. If they are not equal, then we know that $\ell'' > \ell$, so we `extract` $s_1'' = s_1[i_1..i_1 + \ell'' - 1]$ and

computed. Extending the "unified theorem" [45, Thm. 5] to m fingers (to support m LCP operations in the sequence) introduces an $\mathcal{O}(\log m)$ additive amortized time in the operations, since now $W = \Theta(m)$.

$s_2'' = s_2[i_2..i_2+\ell''-1]$ to complete the search for ℓ' inside those (note we are still in Step 1). We use the same sequence ℓ_i of the previous paragraph, with the only difference that the accesses are done on trees of size ℓ'' and not n; therefore each step costs $\mathcal{O}(\log \ell'') = \mathcal{O}(\log^{2/3} n)$ instead of $\mathcal{O}(\log n)$. After finally finding ℓ', we `introduce` back s_1'' and s_2'' into s_1 and s_2. Step 1 then completes in amortized time $\mathcal{O}(\log n + \log^{2/3} n \log\log \ell) = \mathcal{O}(\log n)$. Having found $\ell' \le \ell^2$, we proceed with Step 2 onwards as above, taking $\mathcal{O}(\log^2 \ell)$ additional time.

When the strings are the same. In the case $s_1 = s_2$, assume w.l.o.g. $i_1 < i_2$. We can still carry out Step 1 and, if $i_1 + \ell' \le i_2$, proceed with the plan in the same way, extracting s_1' and s_2' from the same string and later reintroducing them. In case $i_1 + \ell' > i_2$, however, both substrings overlap. In this case we extract just one substring, $s' = s_1[i_1..i_2 + \ell' - 1]$, which is of length at most $2\ell'$, and run the basic exponential search between $s'[1..]$ and $s'[i_2 - i_1 + 1..]$ still in amortized time $\mathcal{O}(\log \ell \log \ell')$. We finally reintroduce s' in s_1. The same is done if we need to extract s_1'' and s_2'': if both come from the same string and $i_1 + \ell'' > i_2$, then we extract just one single string $s'' = s[i_1..i_2 + \ell'' - 1]$ and obtain the same asymptotic times.

Lexicographic comparisons. Once we know that (whp) the LCP of the suffixes is of length ℓ, we can determine which is smaller by accessing (using `access`) the symbols at positions $s_1[i_1 + \ell]$ and $s_2[i_2 + \ell]$ and comparing them, in $\mathcal{O}(\log |s_1 s_2|)$ additional amortized time.

4.2 Substring reversals

Operation $\texttt{reverse}(s, i, j)$ changes s to $s[..i-1]s[j]s[j-1]\cdots s[i+1]s[i]s[j+1..]$. Reflecting it directly in our current structure requires $\Omega(j - i + 1)$ time, which is potentially $\Omega(|s|)$. Our strategy, instead, is to just "mark" the subtrees where the reversal should be carried out, and de-amortize its cost across future operations, materializing it progressively as we traverse the marked subtrees. To this end, we extend our FeST data structure with a new Boolean field x.rev in each node x, which indicates that its whole subtree should be regarded as reversed, that is, its descending nodes should be read right-to-left, but that this update has not yet been carried out. This field is set to *false* on newly created nodes. We also add a field x.fprev, so that if x represents $s[i..j]$, then $x.\text{fprev} = \kappa(s[j]s[j-1]\cdots s[i+1]s[i])$ is the fingerprint of the reversed string. When x.rev is *true*, the fields of x (including x.fp and x.fprev) still do not reflect the reversal.

The fields x.fprev must be maintained in the same way as the fields x.fp. Concretely, upon every update where the children of node x change, we not only recompute x.fp as shown in Section 3.1, but also $x.\text{fprev} = ((x.\text{right.fprev} \cdot b + x.\text{char}) \cdot x.\text{left.power} + x.\text{left.fprev}) \bmod p$.

In order to apply the described reversal to a substring $s[i..j]$, we first compute $y = \texttt{isolate}(i, j)$ on the tree of s, and then toggle the Boolean value $y.\text{rev} = \neg\, y.\text{rev}$ (note that, if y had already an unprocessed reversal, this is undone without ever materializing it). The operation `reverse` then takes $\mathcal{O}(\log |s|)$ amortized time, dominated by the cost of $\texttt{isolate}(i, j)$. We must, however, handle potentially reversed nodes.

Fixing marked nodes. Every time we access a tree node, if it is marked as reversed, we *fix* it, after which it can be treated as a regular node because its fields will already reflect the reversal of its represented string (though some descendant nodes may still need fixing).

Fixing a node involves exchanging its left and right children, toggling their reverse marks, and updating the node fingerprint. More precisely, we define the primitive $\texttt{fix}(x)$ as follows: if x.rev is *true*, then (i) set x.rev = *false*, $x.\text{left.rev} = \neg\, x.\text{left.rev}$, $x.\text{right.rev} = \neg\, x.\text{right.rev}$,

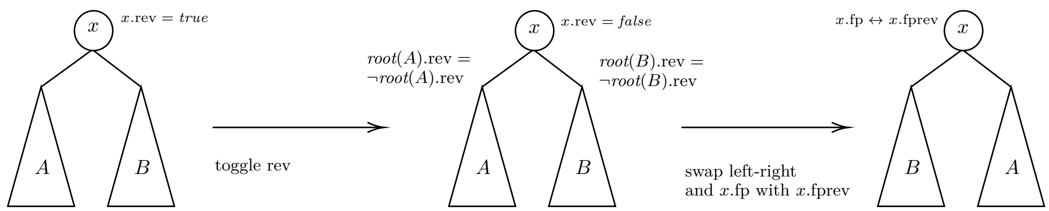


Figure 3 Scheme of the `fix` operation on node x.

(ii) swap x.left with x.right, and (iii) swap x.fp with x.fprev. See Fig. 3 for an example. It is easy to see that `fix` maintains the invariants about the meaning of the reverse fields.

Because all the operations in splay trees, including the *splay*, are done along paths that are first traversed downwards from the root, it suffices that we run `fix`(x) on every node x we find as we descend from the root (for example, on every node x where we perform `find`(x, i)), before taking any other action on the node. This ensures that all the accesses and structural changes to the splay tree are performed over fixed nodes, and therefore no algorithm needs further changes. For example, when we perform *splay*(x), all the ancestors of x are already fixed. As another example, if we run `equal` as in Section 3.6, the nodes y_1 and y_2 will already be fixed by the time we read their fingerprint fields. As a third example, if we run `retrieve`(s, i, j) as in Section 3.3 and the subtree of y has reversed nodes inside, we will progressively fix all those nodes as we traverse the subtree, therefore correctly retrieving $s[i..j]$ within $\mathcal{O}(j - i + 1)$ time.

Note that `fix` takes constant time per node and does not change the potential function Φ, so no time complexities change due to our adjustments. The new fields also enable other queries, for example to decide whether a string is a palindrome.

4.3 Involutions

We support the operation `map`(s, i, j) analogously to substring reversals, that is, isolating $s[i..j]$ in a node $y =$ `isolate`(i, j) and then marking that the substring covered by node y is mapped using a new Boolean field y.map, which is set to *true*. This will indicate that every symbol $s[k]$, for $i \leq k \leq j$, must be interpreted as $f(s[k])$, but that the change has not yet been materialized. Similarly to `reverse`, this information will be propagated downwards as we descend into a subtree, otherwise it is maintained in the subtree's root only. The operation will then take $\mathcal{O}(\log |s|)$ amortized time.

To manage the mapping and deamortize its linear cost across subsequent operations, we will also store fields $x.\text{mfp} = \kappa(f(s[i])f(s[i+1]) \cdots f(s[j]))$ and $x.\text{mfprev} = \kappa(f(s[j])f(s[j-1]) \cdots f(s[i]))$, which maintain the fingerprint of the mapped string, and its reverse, represented by x. Those are maintained analogously as the previous fingerprints: (1) $x.\text{mfp} = ((x.\text{left.mfp} \cdot b + f(x.\text{char})) \cdot x.\text{right.power} + x.\text{right.mfp}) \bmod p$, and (2) $x.\text{mfprev} = ((x.\text{right.mfprev} \cdot b + f(x.\text{char})) \cdot x.\text{left.power} + x.\text{left.mfprev}) \bmod p$.

As for string reversals, every time we access a tree node, if it is marked as mapped, we unmark it and toggle the mapped mark of its children, before proceeding with any other action. Precisely, we define the primitive `fixm`(x) as follows: if x.map is *true*, then (i) set x.map = *false*, x.left.map = ¬ x.left.map, x.right.map = ¬ x.right.map, (ii) set x.char = $f(x.\text{char})$, and (iii) swap x.fp with x.mfp, and x.fprev with x.mfprev. We note that, in addition, the `fix` operation defined in Section 4.2 must also exchange x.mfp with x.mfprev if we also support involutions. Note how, as for reversals, two applications of f cancel each other, which is correct because f is an involution. Operation `fixm` is applied in the same way as `fix` along tree traversals.

Reverse complementation. By combining string reversals and involutions, we can for example support the application of *reverse complementation* of substrings in DNA sequences, where a substring $s[i..j]$ is reversed and in addition its symbols are replaced by their Watson-Crick complement, applying the involution $f(\texttt{A}) = \texttt{T}$, $f(\texttt{T}) = \texttt{A}$, $f(\texttt{C}) = \texttt{G}$, and $f(\texttt{G}) = \texttt{C}$. In case we *only* want to perform reverse complementation (and not reversals and involutions independently), we can simplify our fields and maintain only a Boolean field x.rc and the fingerprint x.mfprev in addition to x.fp. Fixing a node consists of: if x.rc is *true*, then (i) set $x.\text{rc} = \textit{false}$, $x.\text{left.rc} = \neg\, x.\text{left.rc}$, $x.\text{right.rc} = \neg\, x.\text{right.rc}$, (ii) set $x.\text{char} = f(x.\text{char})$, (iii) swap x.left with x.right, (iv) swap x.fp with x.mfprev.

5 Circular strings and omega extension

Our data structure can be easily extended to handle circular strings. We do this by introducing a new routine, called `rotate`, which allows us linearize the circular string starting at any of its indices. By carefully using this primitive, along with a slight modification for the computation of fingerprints, we can support every operation that we presented on linear strings with the same time bounds, as well as signed reversals, in $\mathcal{O}(\log|\hat{s}|)$ amortized time.

By supporting operations on circular strings, we can also handle the omega extension of strings, which is the infinite concatenation of a string: $s^\omega = s \cdot s \cdots$. Again, we are able to meet the same time bounds on every operation on linear strings. We also define two ways to implement the equality between omega-extended substrings. Full details will be contained in the full version of the paper.

6 Conclusion

We presented a new data structure, a forest of enhanced splay trees (FeST), to handle collections of dynamic strings. Our solution is much simpler than those offering the best theoretical results, while still offering logarithmic amortized times for most update and query operations. We answer queries correctly whp, and updates are always correct.

To build our data structure, we employ an approach that differs from theoretical solutions: we use a splay tree for representing each string, enhancing it with additional annotations. The use of binary trees to represent dynamic strings is not new, but exploiting the simplicity of splay trees for attaching and detaching subtrees is. As our FeST is easy to understand, explain, and implement, we believe that it offers the opportunity of wide usability and can become a textbook implementation of dynamic strings. Further, we have found nontrivial—yet perfectly implementable—solutions to relevant queries, like computing the length ℓ of the longest common prefix of two suffixes in time $\mathcal{O}(\log n + \log^2 \ell)$ instead of the trivial $\mathcal{O}(\log^2 n)$. The simplicity of our solution enables new features, like the possibility of reversing a substring, or reverse-complementing it, to be easily implemented in logarithmic amortized time. Our data structure also allows handling circular strings, as well as omega-extensions of strings—features competing solutions have not explored.

References

1 Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. Pattern matching in dynamic texts. In *Proc. 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 819–828, 2000.

2 Amihood Amir, Itai Boneh, Panagiotis Charalampopoulos, and Eitan Kondratovsky. Repetition detection in a dynamic string. In *Proc. 27th Annual European Symposium on Algorithms (ESA)*, pages 5:1–5:18, 2019.

3 Amihood Amir, Panagiotis Charalampopoulos, Costas S. Iliopoulos, Solon P. Pissis, and Jakub Radoszewski. Longest common factor after one edit operation. In *Proc. 24th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 14–26, 2017.

4 Amihood Amir, Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. Dynamic and internal longest common substring. *Algorithmica*, 82(12):3707–3743, 2020.

5 Lorraine A.K. Ayad and Solon P. Pissis. MARS: Improving multiple circular sequence alignment using refined sequences. *BMC Genomics*, 18(1):1–10, 2017.

6 Vineet Bafna and Pavel A. Pevzner. Genome rearrangements and sorting by reversals. In *Proc. 34th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 148–157, 1993.

7 Hans-Juergen Boehm, Russell R. Atkinson, and Michael F. Plass. Ropes: An alternative to strings. *Software Practice and Experience*, 25(12):1315–1330, 1995.

8 Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. Computing the original eBWT faster, simpler, and with less memory. In *Proc. 28th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 129–142, 2021.

9 Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. r-indexing the eBWT. *Information and Computation*, 298:105155, 2024. doi:10.1016/j.ic.2024.105155.

10 Alberto Caprara. Sorting by reversals is difficult. In *Proc. 1st Annual International Conference on Research in Computational Molecular Biology (RECOMB)*, pages 75–83, 1997.

11 Alberto Caprara and Romeo Rizzi. Improved approximation for breakpoint graph decomposition and sorting by reversals. *Journal of Combinatorial Optimization*, 6(2):157–182, 2002.

12 Jean Cardinal and John Iacono. Modular subset sum, dynamic strings, and zero-sum sets. In *Proc. 4th Symposium on Simplicity in Algorithms (SOSA)*, pages 45–56. SIAM, 2021.

13 Giulio Cerbai and Luca S. Ferrari. Permutation patterns in genome rearrangement problems: The reversal model. *Discrete Applied Mathematics*, 279:34–48, 2020.

14 Panagiotis Charalampopoulos, Tomasz Kociumaka, and Shay Mozes. Dynamic string alignment. In *Proc. 31st Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 9:1–9:13, 2020.

15 Panagiotis Charalampopoulos, Tomasz Kociumaka, Jakub Radoszewski, Solon P. Pissis, Wojciech Rytter, Tomasz Walen, and Wiktor Zuba. Approximate circular pattern matching. In *Proc. 30th Annual European Symposium on Algorithms (ESA)*, pages 35:1–35:19, 2022.

16 M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

17 Raphaël Clifford, Allan Grønlund, Kasper Green Larsen, and Tatiana Starikovskaya. Upper and lower bounds for dynamic data structures on strings. In *Proc. 35th Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 22:1–22:14, 2018.

18 Richard Cole. On the dynamic finger conjecture for splay trees. Part II: The proof. *SIAM Journal on Computing*, 30(1):44–85, 2000.

19 James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. In *Proc. 18th Annual ACM Symposium on Theory of Computing (STOC)*, pages 109–121, 1986.

20 Guillaume Fertin, Anthony Labarre, Irena Rusu, Eric Tannier, and Stéphane Vialette. *Combinatorics of Genome Rearrangements.* MIT Press, 2009.

21 Mitsuru Funakoshi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Longest substring palindrome after edit. In *Proc. 29th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 12:1–12:14, 2018.

22 Pawel Gawrychowski. Pattern matching in Lempel-Ziv compressed strings: Fast, simple, and deterministic. In *Proc. 19th Annual European Symposium on Algorithms (ESA)*, pages 421–432, 2011.

23 Pawel Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Jakub Lacki, and Piotr Sankowski. Optimal dynamic strings. *CoRR*, abs/1511.02612, 2015.

24 Pawel Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Jakub Lacki, and Piotr Sankowski. Optimal dynamic strings. In *Proc. 29th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1509–1528, 2018.

25 Roberto Grossi, Costas S. Iliopoulos, Jesper Jansson, Zara Lim, Wing-Kin Sung, and Wiktor Zuba. Finding the cyclic covers of a string. In *Proc. 17th International Conference and Workshops on Algorithms and Computation (WALCOM)*, pages 139–150, 2023.

26 Roberto Grossi, Costas S Iliopoulos, Robert Mercas, Nadia Pisanti, Solon P Pissis, Ahmad Retha, and Fatima Vayani. Circular sequence comparison: algorithms and applications. *Algorithms for Molecular Biology*, 11(1):1–14, 2016.

27 Yijie Han. Improving the efficiency of sorting by reversals. In *Proc. International Conference on Bioinformatics & Computational Biology (BIOCOMP)*, pages 406–409, 2006.

28 Sridhar Hannenhalli and Pavel A. Pevzner. Transforming cabbage into turnip: Polynomial algorithm for sorting signed permutations by reversals. In *Proc. 27th Annual ACM Symposium on Theory of Computing (STOC)*, pages 178–189, 1995.

29 Costas S. Iliopoulos, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Tomasz Walen, and Wiktor Zuba. Linear-time computation of cyclic roots and cyclic covers of a string. In *Proc. 34th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 15:1–15:15, 2023.

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

31 Dominik Kempa and Tomasz Kociumaka. Dynamic suffix array with polylogarithmic queries and updates. In *Proc. 54th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 1657–1670. ACM, 2022.

32 Dominik Kempa and Tomasz Kociumaka. Dynamic suffix array with polylogarithmic queries and updates. *CoRR*, abs/2201.01285, 2022. URL: `https://arxiv.org/abs/2201.01285`, `arXiv:2201.01285`.

33 Dominik Kempa and Ben Langmead. Fast and space-efficient construction of AVL grammars from the LZ77 parsing. *CoRR*, 2105.11052, 2021.

34 D. E. Knuth. *The Art of Computer Programming, volume 3: Sorting and Searching*. Addison-Wesley, 2nd edition, 1998.

35 Tomasz Kociumaka, Anish Mukherjee, and Barna Saha. Approximating edit distance in the fully dynamic model. In *Proc. 64th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1628–1638. IEEE, 2023.

36 M. Lothaire. *Applied Combinatorics on Words*. Cambridge University Press, 2005.

37 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the Burrows-Wheeler transform. *Theoretical Computer Science*, 387(3):298–312, 2007.

38 Kurt Mehlhorn, Rajamani Sundar, and Christian Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997.

39 Kevin Menard, Chris Seaton, and Benoit Daloze. Specializing ropes for ruby. In *Proc. 15th International Conference on Managed Languages & Runtimes (ManLang)*, pages 10:1–10:7, 2018.

40 Gonzalo Navarro and Nicola Prezza. Universal compressed text indexing. *Theoretical Computer Science*, 762:41–50, 2019.

41 Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Fully dynamic data structure for LCE queries in compressed space. In *Proc. 41st International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 72:1–72:14, 2016.

42 Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Dynamic index and LZ factorization in compressed space. *Discrete Applied Mathematics*, 274:116–129, 2020.

43 Andre Rodrigues Oliveira, Ulisses Dias, and Zanoni Dias. On the sorting by reversals and transpositions problem. *Journal of Universal Computer Science*, 23(9):868–906, 2017.

44 W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1-3):211–222, 2003.

45 Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

46 Damien Stehlé and Paul Zimmermann. A binary recursive gcd algorithm. In *Proc. 6th International Symposium on Algorithmic Number Theory (ANTS)*, pages 411–425, 2004.

47 Rajamani Sundar and Robert E. Tarjan. Unique binary-search-tree representations and equality testing of sets and sequences. *SIAM Journal on Computing*, 23(1):24–44, 1994.

48 Wojciech Szpankowski. A generalized suffix tree and its (un)expected asymptotic behaviors. *SIAM Journal on Computing*, 22(6):1176–1198, 1993.

49 Yuki Urabe, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Longest Lyndon substring after edit. In *Proc. 29th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 19:1–19:10, 2018.

50 G.A. Watterson, W.J. Ewens, T.E. Hall, and A. Morgan. The chromosome inversion problem. *Journal of Theoretical Biology*, 99:1–7, 1982.

APPENDIX

A Figures

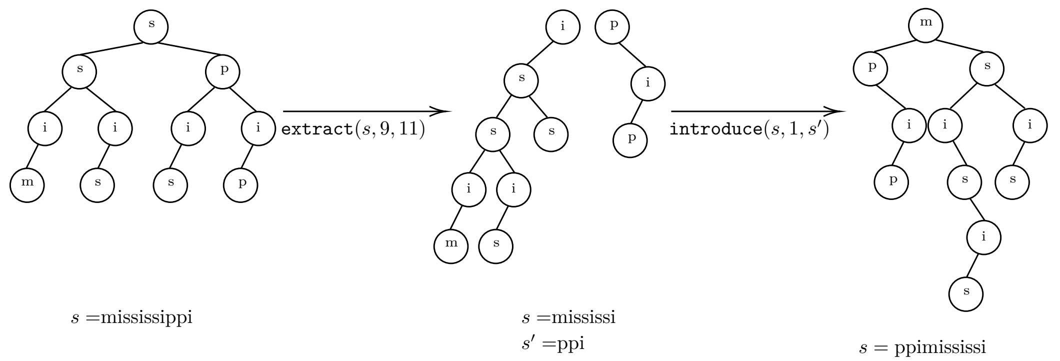


Figure 4 Cycle-rotation operation: $\texttt{rotate}(s, 9)$ moves $s[9..]$ to the left of $s[..8]$. After the rotation the string becomes $s[9..]s[..8]$.

B Other Related Work

A related line of work aims at maintaining a data structure such that the solution to some particular problem on one or two strings can be efficiently updated when these strings undergo an edit operation (deletion, insertion, or substitution). Examples are longest common factor of two strings [3, 4], optimal alignment of two strings [14], approximating the edit distance [35], longest palindromic substring [21], longest square [2], or longest Lyndon factor [49] of one string. The setup can be what is referred to as partially dynamic, when the original string or strings are returned to their state before the edit, or fully dynamic, when the edit operations are reflected on the original string or strings. Clifford et al. [17] give lower bounds on various problems of this kind when a single substitution is applied.

This setup, also referred to as *dynamic strings*, differs from ours in several ways: (a) we are not only interested in solving one specific problem on strings; (b) we have an entire collection of strings, and will want to ask queries on any one or any pair of these; and (c) we allow many different kinds of update operations.

Locally consistent parsings to maintain dynamic strings have been used to support more complex problems, such as simulating suffix arrays [31, 32].

C Circular strings and omega extensions

C.1 Additional definitions

In this section, we are going to use some further concepts regarding periodicity and conjugacy.

A string s is called *periodic* with period r if $s[i + r] = s[i]$ for all $1 \leq i \leq |s| - r$.

Two strings s, t are *conjugates* if there exist strings u, v, possibly empty, such that $s = uv$ and $t = vu$. Conjugacy is an equivalence; the equivalence classes $[s]$ are also called *circular strings*, and any $t \in [s]$ is called a *linearization* of this circular string. Abusing notation, any linear string s can be viewed as a circular string, in which case it is taken as a representative of its conjugacy class. A *substring* of a circular string s is any prefix of any $t \in [s]$, or, equivalently, a string of the form $s[i..j]$ for $1 \leq i, j \leq |s|$ (a linear substring), or $s[i..]s[..j]$, where $j < i$. A *necklace* is a string s with the property that $s \leq_{\text{lex}} t$ for all $t \in [s]$. Every conjugacy class contains exactly one necklace.

When the dynamic strings in our collection are to be interpreted as circular strings, we need to adjust some of our operations. Our model is that we will maintain a canonical representative $\hat{s}$ of the class of rotations of s. All the indices of the operations refer to positions in $\hat{s}$. Internally, we may store in the FeST another representative s of the class, not necessarily $\hat{s}$.

C.2 Circular strings

Our general approach to handle operations on $\hat{s}$ regarding it as circular is to rotate it conveniently before accessing it. The splay tree T of $\hat{s}$ will then maintain some (string) rotation $s = \hat{s}[r..]\hat{s}[..r-1]$ of $\hat{s}$, and we will maintain a field $\text{start}(\hat{s}) = r$ so that we can map any index $\hat{s}[i]$ referred to in update or query operations to $s[((|s| + i - \text{start}(\hat{s})) \bmod |s|) + 1]$.

When we want to change the rotation of $\hat{s}$ to another index r', so that we now store $s' = \hat{s}[r'..]\hat{s}[..r'-1]$, we make use of a new operation $\texttt{rotate}(s, i)$, which rotates s so that its splay tree represents $s[i..]s[..i-1]$. This is implemented as $s' = \texttt{extract}(s, i, |s|)$ followed by $\texttt{introduce}(s, 1, s')$. We then move from rotation r to r' in $\mathcal{O}(\log |s|)$ amortized time by doing $\texttt{rotate}(s, r' - r + 1)$ if $r' > r$, or $\texttt{rotate}(s, |s| + r' - r + 1)$ if $r' < r$. We then set $\text{start}(\hat{s}) = r'$.

Operation $s = \texttt{make-string}(w)$ stays as before, in the understanding that $\hat{s} = w$ will be seen as the canonical representation of the class, so we set $\text{start}(\hat{s}) = 1$; this can be changed later with a string rotation if desired. All the operations that address a single position $\hat{s}[i]$, like `access` and the edit operations, are implemented verbatim by just shifting the index i using $\text{start}(\hat{s})$ as explained. Instead, the operations `retrieve`, `extract`, `equal`, `reverse`, and `map`, which act on a range $\hat{s}[i..j]$, may give trouble when $i > j$, as in this case the substring is $\hat{s}[i..]\hat{s}[..j]$ by circularity. In this case, those operations will be preceded by a change of rotation from the current one, $r = \text{start}(\hat{s})$, to $r' = 1$, using `rotate` as explained. This guard will get rid of those cases. Note that, in the case of `equal`, we may need to rotate both s_1 and s_2, independently, to compute each of the two signatures.

The two remaining operations deserve some consideration. Operation $\texttt{introduce}(s_1, i, s_2)$ could be implemented verbatim (with the shifting of i), but in this case it would introduce in $\hat{s_1}[i]$ the current rotation of s_2, instead of $\hat{s_2}$ as one would expect. Therefore, we precede the operation by a change of rotation in s_2 to $r' = 1$, which makes the splay tree store $\hat{s_2}$ with $\text{start}(\hat{s_2}) = 1$.

Finally, in operation $\texttt{lcp}(s_1, i_1, s_2, i_2)$ we do not know for how long the LCP will extend, so we precede it by changes of rotations in both s_1 and s_2 that make them start at position 1 of $\hat{s_1}$ and $\hat{s_2}$. In case $s_1 = s_2$, however, this trick cannot be used. One simple solution is to rotate the string every time we call `equal` during Step 1; recall Section 4.1. This will be needed as long as the accesses are done on s_1 and s_2; as soon as we extract the substrings of length ℓ'' (and, later, ℓ' for Step 3), we work only on the extracted strings. While the complexity is preserved, rotating the string every time can be too cumbersome. We can use an alternative way to compute signatures of circular substrings, $\kappa(s[i..]s[..j])$: we compute as in Section 3.6 $\sigma = \kappa(s[i..])$ and $\tau = \kappa(s[..j])$, as well as $b^j \bmod p$, which comes for free with the computation of τ; then $\kappa(s[i..]s[..j]) = (\sigma \cdot b^j + \tau) \bmod p$.

Overall, we maintain for all the operations the same asymptotic running times given in the Introduction when the strings are interpreted as circular.

Signed reversals on circular strings. By combining reversals and involutions, we can support signed reversals on circular strings, too. We do this in the same way as for linear strings, namely by doubling the alphabet Σ of gene identifiers such that each gene i has a negated version $-i$, and using the involution $f(i) = -i$ (and $f(-i) = i$). Note that the original paper in which reversals were introduced [50] used circular chromosomes.

C.3 Omega extensions

Circular dynamic strings allow us to implement operations that act on the omega extensions of the underlying strings. Recall that for a (linear) string s, the infinite string s^ω is defined as the infinite concatenation $s^\omega = s \cdot s \cdot s \cdots$. These are, for example, used in the definition of the *extended Burrows-Wheeler Transform* (eBWT) of Mantaci et al. [37], where the underlying string order is based on omega extensions. In this case, comparisons of substrings may need to be made whose length exceeds the shorter of the two strings s_1 and s_2. We therefore introduce a generalization of circular substrings as follows: t is called an *omega-substring* of s if $t = s[i..]s^k s[..j]$ for some $j < i - 1$ and $k \geq 0$. Note that the suffix $s[i..]$ and the prefix $s[..j]$ may also be empty. Thus, t is an omega-substring of s if and only if $t = v^k v[..j]$ for some $k \geq 1$ and some conjugate v of s.

An important tool in this section will be the famous Fine and Wilf Lemma [36], which states that if a string w has two periods r, q and $|w| \geq r + q - \gcd(r, q)$, then w is also periodic with period $\gcd(r, q)$ (a string s is called periodic with period r if $s[i + r] = s[i]$ for all $1 \leq i \leq |s| - r$). The following is a known corollary, a different formulation of which was proven, e.g., in [37]; we reprove it here for completeness.

▶ **Lemma 4.** *Let u, v be two strings. If $\mathrm{lcp}(u^\omega, v^\omega) \geq |u| + |v| - \gcd(|u|, |v|)$, then $u^\omega = v^\omega$.*

Proof. Let $\ell = \mathrm{lcp}(u^\omega, v^\omega) \geq |u| + |v| - \gcd(|u|, |v|)$. Then the string $t = s_1^\omega[..\ell]$ is periodic both with period $|u|$ and with period $|v|$, and thus, by the Fine and Wilf lemma, it is also periodic with period $\gcd(|u|, |v|)$. Since $\gcd(|u|, |v|) \leq |u|, |v|$, this implies that both u and v are powers of the same string x, of length $\gcd(|u|, |v|)$ and therefore, $u^\omega = x^\omega = v^\omega$. ◀

We further observe that the fingerprint of strings of the form u^k can be computed from the fingerprint of string u. More precisely, let u be a string, $\pi = \kappa(u)$ its fingerprint, and $k \geq 1$. Then, calling $d = b^{|u|} \bmod p$ (which we also obtain in the field y.power when computing $\kappa(u)$), it holds

$$\begin{aligned} \kappa(u^k) &= (\pi \cdot d^{k-1} + \pi \cdot d^{k-2} + \cdots + \pi \cdot d + \pi) \bmod p \\ &= (\pi \cdot (d^{k-1} + d^{k-2} + \cdots + 1)) \bmod p, \end{aligned} \tag{1}$$

where $geomsum(d, k-1) = (d^{k-1} + d^{k-2} + \cdots + 1) \bmod p$ can be computed in $\mathcal{O}(\log k)$ time using the identity $d^{2k+1} + d^{2k} + \cdots + 1 = (d+1) \cdot ((d^2)^k + (d^2)^{k-1} + \cdots + 1)$, as follows[4] (all modulo p):

$$\begin{aligned} geomsum(d, 0) &= 1 \\ geomsum(d, 2k+1) &= (d+1) \cdot geomsum(d^2, k) \\ geomsum(d, 2k) &= d \cdot geomsum(d, 2k-1) + 1 \end{aligned} \tag{2}$$

[4] This technique seems to be folklore. Note that the better known formula $geomsum(d, k) = ((d^{k+1} - 1) \cdot (d-1)^{-1}) \bmod p$ requires computing multiplicative inverses, which takes $\mathcal{O}(\log N)$ time using the extended Euclid's algorithm, or $\mathcal{O}(\log \log N)$ with faster algorithms [46]; those terms would not be absorbed by others in our cost formula.

Extended substring equality. We devise at least two ways in which our `equal` query can be extended to omega extensions. First, consider the query $\mathtt{equal}_\omega(s_1, i_1, s_2, i_2, \ell) = \mathtt{equal}(s_1^\omega, i_1, s_2^\omega, i_2, \ell)$, that is, the normal substring equality interpreted on the omega extensions of s_1 and s_2. We let $v_1 = \mathtt{rotate}(s_1, i_1)$ and $v_2 = \mathtt{rotate}(s_2, i_2)$. Then we have $s_1^\omega[i_1..i_1 + \ell - 1] = v_1^{k_1} v_1[..j_1]$, where $k_1 = \lfloor \ell/|s_1| \rfloor$ and $j_1 = \ell \bmod |s_1|$. If $k_1 = 0$, we simply compute $\kappa_1 = \kappa(s_1^\omega[i_1..i_1+\ell-1]) = \kappa(v_1[..j_1])$. Otherwise, we compute $\kappa_1 = \kappa(s_1^\omega[i_1..i_1+\ell-1])$ by applying Eq. (1) as follows:

$$\kappa_1 = (\kappa(v_1) \cdot (d^{k_1-1} + \cdots + 1) \cdot b^{j_1} + \kappa(v_1[..j_1])) \bmod p. \tag{3}$$

There are various components to compute in this formula apart from the fingerprints themselves. First, note that $d = b^{|s_1|} \bmod p = b^{|v_1|} \bmod p = \mathit{root}(T_1).\mathrm{power}$ for the tree T_1 of s_1 (or v_1), so we have it in constant time. Second, $b^{j_1} \bmod p$ is the field y.power after we compute $\kappa(v_1[..j_1])$ via $y = \mathtt{isolate}(v_1, 1, j_1)$ after completion of $\mathtt{rotate}(s_1, i_1)$, thus we also have it in constant time. Third, $d^{k_1-1} + \cdots + 1 = \mathit{geomsum}(d, k_1 - 1)$ is computed with Eq. (2) in time $\mathcal{O}(\log k_1) \subseteq \mathcal{O}(\log \ell)$.

By Lemma 4 we can define $\ell_\omega = |s_1| + |s_2|$ and, if $\ell \geq \ell_\omega$, run the $\mathtt{equal}_\omega$ query with ℓ_ω instead of ℓ. The lemma shows that $s_1[i_1..i_1 + \ell - 1] = s_2[i_2..i_2 + \ell - 1]$ iff $s_1[i_1..i_1 + \ell_\omega - 1] = s_2[i_2..i_2 + \ell_\omega - 1]$. This limits ℓ to $|s_1| + |s_2|$ in our query and therefore the cost $\mathcal{O}(\log \ell)$ is in $\mathcal{O}(\log |s_1 s_2|)$.

We compute κ_2 analogously, and return *true* if and only if $\kappa_1 = \kappa_2$, after undoing the rotations to get back the original strings s_1 and s_2. The total amortized time for operation $\mathtt{equal}_\omega$ is then $\mathcal{O}(\log |s_1 s_2|)$. Note that our results still hold whp because we are deciding on fingerprints of strings of length $\mathcal{O}(N)$, not $\mathcal{O}(\ell)$ (which is in principle unbounded).

A second extension of `equal` is $\mathtt{equal}_\omega^\omega(s_1, i_1, \ell_1, s_2, i_2, \ell_2)$, interpreted as $(s_1^\omega[i_1..i_1 + \ell_1 - 1])^\omega = (s_2^\omega[i_2..i_2 + \ell_2 - 1])^\omega$, that is, the omega extension of $s_1^\omega[i_1..i_1 + \ell_1 - 1]$ is equal to the omega extension of $s_2^\omega[i_2..i_2 + \ell_2 - 1]$. By Lemma 4, this is equivalent to $(s_1^\omega[i_1..i_1 + \ell_1 - 1])^{\ell_2} = (s_2^\omega[i_2..i_2 + \ell_2 - 1])^{\ell_1}$. So we first compute $\kappa_1 = \kappa(s_1^\omega[i_1..i_1 + \ell_1 - 1])$ and $\kappa_2 = \kappa(s_2^\omega[i_2..i_2 + \ell_2 - 1])$ as above, compute $d_1 = b^{\ell_1} \bmod p$ and $d_2 = b^{\ell_2} \bmod p$, and then return whether $(\kappa_1 \cdot (d_1^{\ell_2-1} + \cdots + 1)) \bmod p = (\kappa_2 \cdot (d_2^{\ell_1-1} + \cdots + 1)) \bmod p$. Operation $\mathtt{equal}_\omega^\omega$ is then also computed in amortized time $\mathcal{O}(\log |s_1 s_2|)$.

Extended longest common prefix. We are also able to extend LCPs to omega extensions: operation $\mathtt{lcp}_\omega(s_1, i_1, s_2, i_2)$ computes, for the corresponding rotations $v_1 = \mathtt{rotate}(s_1, i_1)$ and $v_2 = \mathtt{rotate}(s_2, i_2)$, the longest common prefix length $\mathrm{lcp}(v_1^\omega, v_2^\omega)$, as well as the lexicographic order of v_1^ω and v_2^ω. That this can be done efficiently follows again from Lemma 4. We first compare their omega-substrings of length $\ell_\omega = |s_1| + |s_2|$. If $\mathtt{equal}_\omega(s_1, i_1, s_2, i_2, \ell_\omega)$ answers *true*, then it follows that $\mathtt{lcp}(s_1, i_1, s_2, i_2)$ is ∞. Otherwise, we run a close variant of the algorithm described in Section 4.1; note that ℓ_ω can be considerably larger than one of s_1 or s_2. For Step 1, we define $n' = n = |s_1 s_2|$; the other formulas do not change. We run the $\mathtt{equal}_\omega$ computations on s_1 and s_2 using Eq. (3) to compute the fingerprints. We extract the substrings of length ℓ' in Step 3 (analogously, ℓ'' in Step 1) using the `extract` for circular strings, but do so only if $\ell' \leq |s_1|$ (resp., $\ell' \leq |s_2|$); otherwise we keep accessing the original string using Eq. (3). The total amortized time to compute LCPs on omega extensions is thus $\mathcal{O}(\log |s_1 s_2|)$.

C.4 Future work

One feature that we would like to add to our data structure is allowing identification of conjugates. The rationale behind this is that a circular string can be represented by any of its linearizations, so these should all be regarded as equivalent. Furthermore, when the collection contains several conjugates of the same string, then this may be just an artifact caused by the data acquisition process.

This could be solved by replacing each circular string with its necklace representative, that is, the unique conjugate that is lexicographically minimal in the conjugacy class, before applying `make-string`; this representative is computable in linear time in the string length [36]. However, updates can change the lexicographic relationship of the rotations, and thus the necklace representative of the conjugacy class. Recomputing the necklace rotation of s after each update would add worst-case $\mathcal{O}(|s|)$ time to our running times, which is not acceptable. Computing the necklace rotation after an edit operation, or more in general, after any one of our update operations, is an interesting research question, which to the best of our knowledge has not yet been addressed.