Optimal Dynamic Strings*

Paweł Gawrychowski† Adam Karczmarz‡ Tomasz Kociumaka§ Jakub Łącki¶
Piotr Sankowski‖

Abstract

In this paper, we study the fundamental problem of maintaining a dynamic collection of strings under the following operations:

- `make_string` – add a string of constant length,
- `concat` – concatenate two strings,
- `split` – split a string into two at a given position,
- `compare` – find the lexicographical order (less, equal, greater) between two strings,
- `LCP` – calculate the longest common prefix of two strings.

We develop a generic framework for dynamizing the recompression method recently introduced by Jeż [J. ACM, 2016]. It allows us to present an efficient data structure for the above problem, where an update requires only $O(\log n)$ worst-case time with high probability, with n being the total length of all strings in the collection, and a query takes constant worst-case time. On the lower bound side, we prove that even if the only possible query is checking equality of two strings, either updates or queries must take amortized $\Omega(\log n)$ time; hence our implementation is optimal.

1 Introduction

In this paper, we develop a set of general and efficient tools that can be used as basic building blocks in algorithms on dynamic texts. As the prime application, we present an optimal data structure for dynamic strings equality. In this problem, we want to maintain a collection of strings that can be concatenated, split, and tested for equality. Our algorithm supports $O(1)$-time queries; an update requires $O(\log n)$ worst-case time with high probability[1], where n is the total length of the strings in the collection. We also provide a matching $\Omega(\log n)$ lower bound for the amortized complexity of update or query, showing that the solution obtained through our framework is the final answer. Additionally, we extend the supported queries to lexicographical comparison and computing the longest common prefix of two strings in $O(1)$ worst-case time.

These time complexities remain unchanged if the strings are persistent, that is, if concatenation and splitting do not destroy their arguments.[2] This extension lets us efficiently add *grammar-compressed strings* to the dynamic collection, i.e., strings generated by context-free grammars that produce exactly one string. Such grammars, called *straight-line grammars*, are a very convenient abstraction of text compression, and there has been a rich body of work devoted to algorithms processing grammar-compressed strings without decompressing them; see [27] for a survey of the area. Even the seemingly simple task of checking whether two grammar-compressed string are equal is non-trivial, as such strings might have exponential length in terms of the size g of underlying grammars g; see [26] for a survey devoted solely to this problem. Currently, the most efficient algorithm, given by Jeż [19], works in $O(g \log n)$ time, where n is the length of the derived string. This can be matched by simply applying our persistent data structure. The resulting solution is randomized, but has the advantage of being dynamic: we can interchangeably add new non-terminals to the grammar and test equality of already existing non-terminals.

In fact, our solution can be seen as a dynamization of Jeż's recompression technique, originally introduced in [22]. Inspired by the previous results of Mehlhorn et al. [29] for dynamic strings equality, the algorithm by Jeż modifies the input in phases: in each phase a set of non-overlapping pairs of consecutive letters is chosen and each such pair is replaced with a fresh symbol. Jeż successfully applied this idea to solve a wide range of problems, such as compressed membership for finite automata [17], fully compressed pattern matching [19], approximating the smallest grammar [18, 21] and the smallest tree grammar [12, 23], context unification [16], and finally word equations [7,8,20,22]. However, all these problems are static, and hence one can to choose the pairs to be replaced based on the analysis of the whole input, in order to guarantee that the string shrinks by a

*Work done while Paweł Gawrychowski held a post-doctoral position at Warsaw Center of Mathematics and Computer Science. Piotr Sankowski is supported by the Polish National Science Center, grant no 2014/13/B/ST6/00770.

†Institute of Computer Science, University of Wrocław, Poland
‡Institute of Informatics, University of Warsaw, Poland
§Institute of Informatics, University of Warsaw, Poland
¶Google Research, New York
‖Institute of Informatics, University of Warsaw, Poland

[1]The algorithm always returns correct answers; the randomization only impacts the running time.

[2]Consistently with the previous works, here we assume $O(1)$-time arithmetic on the lengths of the strings.

constant factor in each phase. In the dynamic setting, we do not know the future operations, so this is impossible.

Introducing randomness is a very natural idea to deal with the unknown future, and this is exactly how Mehlhorn et al. [29] obtained their polylogarithmic solution to the problem considered in this paper. They use blocks of expected length $O(1)$ instead of pairs of letters. However, with high probability this approach can only guarantee block size $O(\log n)$, which often leads to poor running time. In fact, the subsequent solution for maintaining a collection of dynamic strings, given by Alstrup et al. [1], chooses the blocks deterministically based on their $O(\log^* n)$-size neighborhood, using the deterministic coin tossing of Cole and Vishkin [6].

Our approach is conceptually simpler and allows for faster algorithms: we work with blocks of length two and randomly choose the non-overlapping pairs of letters. This way, each phase is expected to reduce the input size by a constant fraction, and consequently the number of phases can be bounded by $O(\log n)$ with high probability. However, it is likely that many of these phases do not make a significant progress, and this leads to major technical difficulties (see Section 3.4 for details). We manage to face these challenges using a careful implementation based on several state-of-the-art abstract data structures. We believe that these issues were the main reason why over 20 years have passed from the paper by Mehlhorn [29], that gave the first polylogarithmic time data structure for dynamic string equality, to our work, that gives the optimal solution.

In order to provide a generic framework for making recompression dynamic, we make sure that most of the technical content is formulated using reusable subroutines. At the heart of our framework lies a certain *context-insensitive* representation of strings using $O(\log n)$ symbols. A similar representation of size $O(\log n \log^* n)$ can be derived from the approach of Alstrup [1]. In fact, it has been explicitly introduced in a recent independent work by Nishimoto et al. [31, 32].

We believe that our framework has many applications yet to be discovered. In particular, whenever a static problem can be solved with the recompression method, its dynamic version is likely to be efficiently solvable using our techniques. In fact, in the arXiv version of the paper [13] we already apply them to solve dynamic text indexing. Arguably, text indexing is the main problem in algorithms on strings, and its dynamic version seems particularly challenging. Only in the simpler case, where the text is updated by either prepending or appending characters, the problem appears well-understood [3,4,11,24,25]. A number of nontrivial results are also known for the more complex general case, where a single update might dramatically change the structure of the possible occurrences [1, 9, 10, 14, 36, 39]. Using our framework, we further decrease the time complexities and generalize the solution to the persistent setting.

1.1 Our results We obtain a data structure that maintains a dynamic collection $\mathcal{W}$ of non-empty persistent strings. The collection is initially empty and then can be extended using the following updates:

- `make_string`(w), for $w \in \Sigma^+$, results in $\mathcal{W} := \mathcal{W} \cup \{w\}$;[3]
- `concat`(w_1, w_2), for $w_1, w_2 \in \mathcal{W}$, results in $\mathcal{W} := \mathcal{W} \cup \{w_1 w_2\}$;
- `split`(w, k), for $w \in \mathcal{W}$ and $1 \leq k < |w|$, results in $\mathcal{W} := \mathcal{W} \cup \{w[..k], w[(k+1)..]\}$;
- `compare`(w_1, w_2), for $w_1, w_2 \in \mathcal{W}$, checks whether $w_1 < w_2$, $w_1 = w_2$, or $w_1 > w_2$;
- `LCP`(w_1, w_2), for $w_1, w_2 \in \mathcal{W}$, returns the length of the longest common prefix of w_1 and w_2.

Each string in $\mathcal{W}$ has an integer *handle* assigned by the update which created it. This way the arguments to `concat` and `split` operations have constant size. Moreover, we make sure that if an update creates a string which is already present in $\mathcal{W}$, the original handle is returned. Otherwise, a fresh handle is assigned to the string. We assume that handles are consecutive integers starting from 0 to avoid non-deterministic output. Note that in this setting there is no need for an extra `equal` operation, as it can be implemented by testing the equality of handles.

The first result on dynamic string equality was given by Sundar and Tarjan [38]. They achieved amortized $O(\sqrt{n \log m} + \log m)$ time for an update, where m is the number of operations executed so far and n is the total length of the strings. This was later improved to expected logarithmic time for the special case of repetition-free sequences [35]. The first improvement for the general case was due to Mehlhorn et al. [29], who decreased the update time to $O(\log^2 n)$ in expectation. They also provided a deterministic version working in $O(\log n(\log n + \log m \log^* m))$ time. Finally, much later, the deterministic algorithm of [29] was substantially improved by Alstrup et al. [1], who achieved $O(\log n \log^* n)$ update time with high probability.[4] In all these solutions, equality of two strings can be checked in worst-case $O(1)$ time. Alstrup et al. [1] were the first to provide an $O(1)$-time `compare` implementation.

[3] Note that the `make_string` operation is slightly more general than what we mentioned in the abstract, as it may add strings of arbitrary length.

[4] While not explicitly stated in the paper, this bound holds w.h.p., as the algorithm uses hash tables.

Using dynamic lowest common ancestor queries [5], their framework can be easily adapted to also support `LCP` in $O(1)$ time. In previous implementations, these two operations could be simulated by a binary search using the `split` and `equal` operations.

We provide the final answer by further improving the update time to $O(\log n)$ with high probability and showing that either an update or an equality test requires $\Omega(\log n)$ time, even if the alphabet is binary and one allows amortization and randomization. In addition, we show that our data structure can be extended to support `compare` and `LCP` queries in constant time, without affecting the update time bound.

We note that it is very simple to achieve $O(\log n)$ update time for maintaining a non-persistent family of strings under concatenate and split operations if we allow the 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 (even if the time bounds should only hold in expectation). 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.

1.2 Related Techniques Our structure for dynamic string equality is based on maintaining a hierarchical representation of the strings, similarly to the previous works [1, 29]. In such an approach the representation should be, in a sense, locally consistent, meaning that two equal strings must have identical representations and the representations of two strings can be joined to form the representation of their concatenation at a relatively small cost. The process of creating such a representation can be imagined as parsing: breaking the string into blocks, replacing every block by a new character, and repeating the process on the new shorter string.

Deciding how to partition the string into blocks is very closely related to the list ranking problem, where we are given a linked list of n processors and every processor wants to compute its position on the list. This requires resolving contention, that is, choosing a large independent subset of the processors. A simple approach, called the random mate algorithm [30, 40], is to give a random bit to every processor and select processors having bit 0 such that their successor has bit 1. A more complicated (and slower by a factor of $O(\log^* n)$) deterministic solution, called deterministic coin tossing, was given by Cole and Vishkin [6]. Such symmetry-breaking method is the gist of all previous solutions for dynamic string equality. Mehlhorn et al. [29] used a randomized approach (different than random mate) and then applied deterministic coin tossing to develop the deterministic version. Later, Alstrup et al. [1] further optimized the deterministic solution.

To obtain an improvement on the work of Alstrup et al. [1], we take a step back and notice that while partitioning the strings into blocks is done deterministically, obtaining an efficient implementation requires hash tables, so the final solution is randomized anyway. Hence, we do not lose anything by replacing the complicated deterministic coin tossing technique with the simple random mate paradigm. As a result, we obtain a data structure that is conceptually simpler, although its efficient implementation requires facing several major technical challenges.

1.3 Organization of the Paper In Section 2 we introduce basic notations. Then, in Section 3 we present the main ideas of our dynamic string collections and we sketch how they can be used to give a data structure that supports equality tests in $O(1)$ time. We also discuss differences between our approach and the previous ones (in Section 3.4). The details of our implementation are given in the following three sections: Section 4 describes iterators for traversing parse trees of grammar-compressed strings. Then, in Section 5 we prove some combinatorial facts concerning context-insensitive decompositions. Combined with the results of previous sections, these facts let us handle equality tests and updates on the dynamic string collection in a clean and uniform way in Section 6. Finally, in Section 7 we provide a lower bound for maintaining a family of non-persistent strings. In the full version of the paper [13], we show how to extend the data structure developed in Sections 3 to 6 in order to support lexicographic comparisons and longest common prefix queries.

2 Preliminaries

2.1 Model of Computations We use the word RAM model with randomization, assuming that the machine word has B bits. Some of our algorithms rely on certain numbers fitting in a machine word, which is a weak assumption because a word RAM machine with word size B can simulate a machine with word size $O(B)$ with constant-factor slowdown.

2.2 Strings Let Σ be a countable set of *letters* that we call an *alphabet*. We denote by Σ^* a set of all finite *strings* over Σ, and by Σ^+ all the non-empty strings. Our data structure maintains a family of strings over some integer alphabet Σ. Internally, it also uses strings over a larger (infinite) alphabet $\mathcal{S} \supseteq \Sigma$. We say that

each element of $\mathcal{S}$ is a *symbol*.

Let $w = w_1 \cdots w_k$ be a string (throughout the paper we assume that the string indexing is 1-based). For $1 \le a \le b \le |w|$ a word $u = w_a \cdots w_b$ is called a *substring* of w. By $w[a..b]$ we denote the *occurrence* of u in w at position a, called a *fragment* of w. We use shorter notation $w[a]$, $w[..b]$, and $w[a..]$, to denote $w[a..a]$, $w[1..b]$, and $w[a..|w|]$, respectively. Additionally, we sometimes slightly abuse notation and use $w[a+1..a]$ for $0 \le a \le |w|$ to represent empty fragments.

The concatenation of two strings w_1 and w_2 is denoted by $w_1 \cdot w_2$ or simply $w_1 w_2$. For $k \in \mathbb{Z}_+$, the concatenation of k copies of w is denoted by w^k.

2.3 Straight-Line Grammars A straight-line grammar (or a straight-line program) is usually defined as a non-recursive context-free grammar which has exactly one production rule for each non-terminal. In our applications, it is convenient not to distinguish any start symbol and to allow infinitely many non-terminals. Hence, we abuse the naming slightly and use the following

Definition 2.1. *For an alphabet Σ, a set of* non-terminals *V (disjoint with Σ), and a function $P : V \to (V \cup \Sigma)^*$, we say that $\mathcal{G} = (V, \Sigma, P)$ is a* straight-line grammar *if the non-terminals can be well ordered so that each non-terminal S is strictly larger than all the non-terminals occurring in $P(S)$.*

Each *symbol* $S \in \Sigma \cup V$ of the straight-line grammar $\mathcal{G}$ yields a unique string $str(S) \in \Sigma^*$: we set $str(S) = S$ for terminals $S \in \Sigma$, and for non-terminals $S \in V$, we define $str(S) = str(S_1) \cdots str(S_k)$, where $P(S) = S_1 \cdots S_k$.

We associate a *parse tree* $\mathcal{T}(S)$ with each symbol $S \in V \cup \Sigma$. This is an ordered rooted tree, which represents the *derivation* $S \xrightarrow{*} str(S)$. Each node of $\mathcal{T}(S)$ represents a symbol of $\mathcal{G}$; the root represents S. If $S \in \Sigma$ is a non-terminal, then $\mathcal{T}(S)$ does not contain any other node. Otherwise, if $P(S) = S_1 \cdots S_k$, we attach to the root a copy of $\mathcal{T}(S_i)$ for each symbol S_i, preserving the order. Observe that $str(S)$ can be retrieved from $\mathcal{T}(S)$ if we traverse the tree and spell out the underlying letter whenever we enter a node representing a terminal.

A set of symbols $X \subseteq \Sigma \cup V$ is *closed* if $P(S) \in X^*$ for each $S \in X \cap V$. Note that in this case $(X \cap V, X \cap \Sigma, P|_{X \cap V})$ is also a straight-line grammar.

For a fixed alphabet Σ, we define a universal set of symbols $\mathcal{S}$ so that $\Sigma \subseteq \mathcal{S}$ and $\mathcal{S}^* \subseteq \mathcal{S}$. More precisely, $\mathcal{S}_0 = \Sigma$, $\mathcal{S}_{n+1} = \mathcal{S}_n \cup \mathcal{S}_n^*$, and $\mathcal{S} = \bigcup_{n=0}^{\infty} \mathcal{S}_n$. Observe that $\mathcal{U} = (\mathcal{S} \setminus \Sigma, \Sigma, \mathsf{id}_{\mathcal{S} \setminus \Sigma})$ is a straight-line grammar; we call it *universal* because, intuitively speaking, each straight-line grammar with alphabet $\Sigma' \subseteq \Sigma$ can be embedded in $\mathcal{U}$ and represented as a closed subset of $\mathcal{S}$.

3 Overview of Our Result

In this section, we present the main ideas behind the construction of our data structure for maintaining a family of strings that can be tested for equality. The full details are provided in Sections 4 to 6.

3.1 Locally Consistent Parsing We represent the collection $\mathcal{W} = \{w_1, \ldots, w_m\}$ using symbols $S_1, \ldots, S_m$ from the universal set $\mathcal{S}$ so that $str(S_i) = w_i$. Moreover, we store a grammar which lets us retrieve the strings w_i.

For each string $w \in \Sigma^+$, many symbols $S \in \mathcal{S}$ satisfy $str(S) = w$. In other words, the string w can be *parsed* in multiple ways. Below, we provide a randomized construction of a *parsing*, i.e., a mapping $str^{-1} : \Sigma^+ \to \mathcal{S}$ such that $str(str^{-1}(w)) = w$ for $w \in \Sigma^+$. Our parsing has small *depth* and is *locally consistent* (see Section 3.5).

The parsing is based on two functions that we now introduce. We first define the *run-length encoding* $\textsc{Rle} : \mathcal{S}^* \to \mathcal{S}^*$. To compute $\textsc{Rle}(w)$, we partition w into *runs*, i.e., maximal blocks consisting of adjacent equal symbols. Next, we replace each run of size at least two with the corresponding non-terminal; runs of size 1 are kept intact. For example, for $w = \mathtt{aabaaccaaa}$, we have $\textsc{Rle}(w) = S_1 \mathtt{b} S_1 S_2 S_3$, where $S_1 = \mathtt{aa}$, $S_2 = \mathtt{cc}$, and $S_3 = \mathtt{aaa}$. Note that the result of $\textsc{Rle}$ is *1-repetition-free*, i.e., its every two consecutive symbols are distinct.

The second function applied to parse a string is $\textsc{Compress}_i : \mathcal{S}^* \to \mathcal{S}^*$, where $i \ge 1$ is a parameter. It is designed so that $\textsc{Compress}_i(w)$ is (in expectation) constant-factor shorter than w if w is 1-repetition-free. To define it formally, we introduce a family of functions $h_i : \mathcal{S} \to \{0, 1\}$ (for $i \ge 1$), each of which uniformly at random assigns 0 or 1 to every symbol. To compute $\textsc{Compress}_i(w)$, we define blocks in the following way. If there are two adjacent symbols $a_j a_{j+1}$ such that $h_i(a_j) = 0$ and $h_i(a_{j+1}) = 1$, we mark them as a block. (Note that such blocks do not overlap.) All other symbols form single-symbol blocks. Next, each block of length two is replaced by the corresponding non-terminal. For example, consider $w = \mathtt{cbabcba}$ and assume that $h_1(\mathtt{a}) = 1$, $h_1(\mathtt{b}) = 0$, and $h_1(\mathtt{c}) = 1$. Then the division into blocks is $\mathtt{c}|\mathtt{ba}|\mathtt{bc}|\mathtt{ba}$, and $\textsc{Compress}_1(\mathtt{cbabcba}) = \mathtt{c} S_1 S_2 S_1$, where $S_1 = \mathtt{ba}$ and $S_2 = \mathtt{bc}$.

In a 1-repetition-free string, each two adjacent symbols are different, so they form a block with probability $\frac{1}{4}$. Thus, we obtain the following:

Observation 3.1. *If $w \in \mathcal{S}^*$ is 1-repetition-free, then $\mathbb{E}(|\textsc{Compress}_i(w)|) \le \frac{3}{4}|w| + \frac{1}{4}$ for every $i \ge 1$.*

In order to build a representation of a string w, we repeatedly apply $\textsc{Rle}$ and $\textsc{Compress}_i$. We define:

$$\textsc{Shrink}_i(w) := \begin{cases} \textsc{Rle}(w) & \text{if } i \text{ is odd,} \\ \textsc{Compress}_{i/2}(w) & \text{if } i \text{ is even,} \end{cases}$$

$$\overline{\textsc{Shrink}_0}(w) := w,$$

$$\overline{\textsc{Shrink}_{i+1}}(w) := \textsc{Shrink}_{i+1}(\overline{\textsc{Shrink}_i}(w)).$$

The *depth* of the representation of w, denoted $\textsc{Depth}(w)$, is the smallest i such that $|\overline{\textsc{Shrink}_i}(w)| = 1$. Observe that $\textsc{Depth}(w)$ depends on the choice of random bits $h_i(\cdot)$, so it is a random variable. Using Observation 3.1, we can bound it from above.

Lemma 3.2. *If $w \in \Sigma^+$ is a string and $r \in \mathbb{R}_{\geq 0}$, then $\mathbb{P}(\textsc{Depth}(w) \leq 8(r + \ln|w|)) \geq 1 - e^{-r}$.*

Proof. Let us define a sequence of random variables $X_i := |\overline{\textsc{Shrink}_{2i}}(w)| - 1$. Recall that functions $\textsc{Compress}_i$ use independent random bits $h_i(\cdot)$ for each index i. Consequently, Observation 3.1 yields $\mathbb{E}(X_i|(h_0, \ldots, h_{i-1})) \leq \frac{3}{4}X_{i-1}$. Note that $X_0 = |\overline{\textsc{Shrink}_0}(w)| - 1 < |w|$, and therefore

$$\mathbb{E}(X_i) < (\tfrac{3}{4})^i \cdot |w| = (1 - \tfrac{1}{4})^i \cdot |w| \leq \exp(-\tfrac{1}{4}i + \ln|w|).$$

Thus, by Markov's inequality:

$$\mathbb{P}(\textsc{Depth}(w) > 8(r + \ln|w|)) = \mathbb{P}(X_{4(r+\ln|w|)} \geq 1) \leq \mathbb{E}(X_{4(r+\ln|w|)}) < \exp(-(r+\ln|w|)+\ln|w|) = e^{-r}. \quad \square$$

By Lemma 3.2, the value $\textsc{Depth}(w)$ is finite with probability 1 for each $w \in \Sigma^+$. Hence, we define the parsing $str^{-1} : \Sigma^+ \to \mathcal{S}$ so that $str^{-1}(w)$ is the only symbol of $\overline{\textsc{Shrink}_{\textsc{Depth}(w)}}(w)$. This notion is almost surely well-defined because the domain Σ^+ is countable.

Remark 3.3. *In a collection $\mathcal{W}$ of $\mathrm{poly}(n)$ strings of length $\mathrm{poly}(n)$, we have $\textsc{Depth}(w) = O(\log n)$ for every $w \in \mathcal{W}$ with high probability (i.e., at least $1 - n^{-c}$ for any constant c).*

For a string $w \in \Sigma^+$, we define $\mathcal{G}(w)$ to be the set of symbols which occur in $\overline{\textsc{Shrink}_i}(w)$ for some $i \in \mathbb{Z}_{\geq 0}$. Moreover, we denote $\mathcal{G}(\mathcal{W}) = \bigcup_{w \in \mathcal{W}} \mathcal{G}(w)$.

Observation 3.4. *If $S \in \mathcal{G}(\mathcal{W})$ for some $\mathcal{W} \subseteq \Sigma^+$, then $S \in \Sigma$, $S = S_1^k$, where $S_1 \in \mathcal{G}(\mathcal{W})$ and $k \in \mathbb{Z}_{\geq 2}$, or $S = S_1 S_2$, where $S_1, S_2 \in \mathcal{G}(\mathcal{W})$ are distinct.*

The characterization above lets us define the *level* of each symbol $S \in \mathcal{G}(\Sigma^+)$: If $S \in \Sigma$ is a terminal, then $\mathit{level}(S) = 0$. If $S = S_1^k$ for $k \in \mathbb{Z}_{\geq 2}$, then $\mathit{level}(S) = \mathit{level}(S_1) + 1$. Otherwise, $S = S_1 S_2$ and $\mathit{level}(S)$ is the smallest even number l such that $l > \max(\mathit{level}(S_1), \mathit{level}(S_2))$, $h_{l/2}(S_1) = 0$, and $h_{l/2}(S_2) = 1$. The following observation characterizes this function.

Observation 3.5. *For each string $w \in \Sigma^+$, the string $\overline{\textsc{Shrink}_l}(w)$ consists of symbols of level at most l. Moreover, each symbol of level less than l corresponds to a length-1 block in $\overline{\textsc{Shrink}_{l-1}}(w)$.*

It turns out that $str : \mathcal{G}(\Sigma^+) \to \Sigma^+$ is a bijection. Such a property did not hold for previous approaches [1, 29]. Although its presence is not crucial for our solution, it makes the data structure conceptually simpler.

Lemma 3.6. *The functions $str : \mathcal{G}(\Sigma^+) \to \Sigma^+$ and $str^{-1} : \Sigma^+ \to \mathcal{G}(\Sigma^+)$ are mutually inverse bijections.*

Proof. Let S be a symbol of $\mathcal{G}(\Sigma^+)$ with $\mathit{level}(S) = \ell$. By definition, there exists a string $w \in \Sigma^+$ such that $S \in \mathcal{G}(w)$. By Observation 3.5, w has a fragment equal to $str(S)$ which is represented by S in $\overline{\textsc{Shrink}_l}(w)$. Consequently, for each $0 \leq i \leq l$, this fragment is represented by a fragment of $\overline{\textsc{Shrink}_i}(w)$. We shall inductively prove that these fragments are equal to $\overline{\textsc{Shrink}_i}(str(S))$. For $i = 0$ this is clear, so let us take $0 \leq i < l$ such that this property holds for i to conclude that the same property holds for $i + 1$. It suffices to prove that $\textsc{Shrink}_{i+1}$ produces the same blocks both in the fragment of $\overline{\textsc{Shrink}_i}(w)$ and in $\overline{\textsc{Shrink}_i}(str(S))$. Since the corresponding fragment of w is represented by a fragment of $\overline{\textsc{Shrink}_{i+1}}(w)$, then $\textsc{Shrink}_{i+1}$ places block boundaries before and after the fragment of $\overline{\textsc{Shrink}_i}(w)$. Moreover, as $\textsc{Shrink}_{i+1}$ places a block boundary between two positions solely based on the characters at these positions, the blocks within the fragment are placed in the analogous positions as in $\textsc{Shrink}_i(str(S))$. Thus, the blocks are the same and this concludes the inductive proof. In particular we get $\textsc{Shrink}_l(str(S)) = S$ and thus $str^{-1}(str(S)) = S$. $\square$

3.2 Storing the Grammar

We maintain a set $\mathcal{G} \subseteq \mathcal{S}$, which satisfies the following property:

Invariant 3.7. *$\mathcal{G}$ is closed and $\mathcal{G}(\mathcal{W}) \subseteq \mathcal{G} \subseteq \mathcal{G}(\Sigma^+)$.*

As a result, $\mathcal{G}$ can be interpreted as a straight-line grammar, which has symbols representing each string $w \in \mathcal{W}$. Formally, the grammar is $(\mathcal{G} \setminus \Sigma, \mathcal{G} \cap \Sigma, \mathsf{id}_{\mathcal{G}\setminus\Sigma})$.

For consistence with earlier works [1,29], we assign a unique integer identifier, a *signature*, to each symbol in $\mathcal{G}$. Whenever our data structure encounters a symbol S missing a signature (a newly created non-terminal or a terminal encountered for the first time), it assigns a fresh signature. We use $\bar{\Sigma}(\mathcal{G})$ to denote the set of signatures corresponding to the symbols that are in the grammar $\mathcal{G}$. We distinguish three types of signatures. We write $s \to a$ for $a \in \Sigma$ if s represents a terminal symbol, $s \to s_1 s_2$ to denote that s represents a non-terminal S which replaced two symbols S_1, S_2 (represented by signatures

s_1, s_2, correspondingly) in COMPRESS_i. Analogously, we write $s \to s_1^k$ to denote that s represents a non-terminal introduced by RLE when k copies of a symbol s_1 were replaced represented by s. We also set $\mathit{level}(s) = \mathit{level}(S)$ and $\mathit{str}(s) = \mathit{str}(S)$.

For each signature $s \in \bar{\Sigma}(\mathcal{G})$ representing a symbol S, we store a record containing some attributes of the associated symbol. More precisely, it contains the following information, which can be stored in $O(1)$ space and generated while we assign a signature to the symbol it represents: (a) the associated production rule $s \to s_1 s_2$, $s \to s_1^k$ (if it represents a non-terminal) or the terminal symbol $a \in \Sigma$ if $s \to a$; (b) the length $\mathit{length}(s)$ of the string generated from S; (c) the level $\mathit{level}(s)$ of s; (d) the random bits $h_i(S)$ for $0 < i \le B$.[5] Since the bits are uniformly random, we can *reveal* them by taking a random machine word.

In order to ensure that a single symbol has a single signature, we store three dictionaries which for each $s \in \bar{\Sigma}(\mathcal{G})$ map a to s whenever $s \to a$, (s_1, s_2) to s whenever $s \to s_1 s_2$, and (s_1, k) to s whenever $s \to s_1^k$. Thus, from now on we sometimes use signatures instead of the corresponding symbols.

Finally, we observe that the signature representing $\mathit{str}^{-1}(w)$ can be used to identify w represented by $\mathcal{G}$. In particular, this enables us to test for equality in a trivial way. However, in order to make sure that updates return deterministic values, we use consecutive integers as handles to elements of $\mathcal{W}$ and we explicitly store the mapping between signatures and handles. If we used signatures as handles, an adversary user would get some information about random choices made by the data structure. For the analysis we need to assume that these choices do not affect future updates.

3.3 Parse Trees We associate two trees with each symbol $S \in \mathcal{G}(\Sigma^+)$. We never build these trees explicitly, but they are very useful in the description of our data structure. The first one is the parse tree $\mathcal{T}(S)$, as defined in Section 2. If $w = \mathit{str}(S)$, we also denote this parse tree by $\mathcal{T}(w)$, which is valid due to Lemma 3.6.

The parse tree $\mathcal{T}(w)$ represents the derivation $S \xrightarrow{*} w$, but it does not provide a comprehensive description of intermediate steps of our parsing procedure. For this, we define an *uncompressed parse tree* $\overline{\mathcal{T}}(w)$, also denoted $\overline{\mathcal{T}}(S)$ for $S = \mathit{str}^{-1}(w)$. To construct $\overline{\mathcal{T}}(w)$, we subdivide the edges of $\mathcal{T}(w)$ as follows (see also Fig. 1): Consider an edge with the lower endpoint representing symbol S_1 and the higher endpoint – S_2. If the levels of these symbols are l_1 and l_2, respectively, we introduce $l_2 - l_1 - 1$ auxiliary nodes on that edge (note that $l_1 < l_2$ by Observation 3.5). We assume that these nodes represent the symbol S_1.

Observe that all root-to-leaf paths in $\overline{\mathcal{T}}(w)$ consist of $\text{DEPTH}(w)$ edges. Thus, the nodes can be partitioned into $\text{DEPTH}(w)+1$ levels, which we number starting from the leaves so that the level l represents $\overline{\text{SHRINK}_l}(w)$:

Observation 3.8. *The string $\overline{\text{SHRINK}_l}(w)$ can be retrieved from $\overline{\mathcal{T}}(w)$ by spelling out symbols represented by the subsequent nodes at level l. Moreover, two such nodes have the same parent if and only if SHRINK_{l+1} puts the corresponding characters in the same block.*

If v is a node in $\overline{\mathcal{T}}(S)$ distinct from the root, we define $\text{par}(v)$ to be the parent node of v in $\overline{\mathcal{T}}(S)$. We denote by $\text{child}(v, k)$ the k-th child of v in $\overline{\mathcal{T}}(S)$. Similarly, if v is a node in $\mathcal{T}(S)$ distinct from the root, we define $\text{par}_{\mathcal{T}(S)}(v)$ to be the parent node of v in $\mathcal{T}(S)$.

Consider a tree $\mathcal{T}(S)$ and its node u representing a symbol with signature $\text{sig}(u)$. Observe that if we know $\text{sig}(u)$, we also know the children of u, as they are determined by the production rule associated with $\text{sig}(u)$. More generally, $\text{sig}(u)$ uniquely determines the subtree rooted at u. On the other hand, $\text{sig}(u)$ in general does not determine the parent of u. Even a single tree may contain multiple nodes with the same signature.

We show how to deal with the problem of navigation in the parse trees. Namely, we prove that once we fix a tree $\mathcal{T}(S)$ or $\overline{\mathcal{T}}(S)$, we may traverse it efficiently, using the information stored in $\mathcal{G}$. We stress that we do not need to build the trees $\mathcal{T}(S)$ and $\overline{\mathcal{T}}(S)$ explicitly.

We use *pointers* to access the nodes of $\overline{\mathcal{T}}(S)$. Assume we have a pointer P to a node $v \in \overline{\mathcal{T}}(S)$ that corresponds to the i-th symbol of $\overline{\text{SHRINK}_j}(w)$. As one may expect, given P we may quickly get a pointer to the parent or child of v. More interestingly, we can also get a pointer to the node $\text{right}(v)$ ($\text{left}(v)$) that lies *right* (or *left*) of v in $\overline{\mathcal{T}}(S)$ in constant time. The node $\text{right}(v)$ is the node that corresponds to the $(i+1)$-th symbol of $\overline{\text{SHRINK}_j}(w)$, while $\text{left}(v)$ corresponds to the $(i-1)$-th symbol; these nodes may not have a common parent with v. For example, consider $\overline{\mathcal{T}}(w)$ in Fig. 1 and denote the level-0 nodes corresponding to the first three letters (b, a and n) by v_b, v_a and v_n. Then, $\text{right}(v_a) = v_n$, but also $\text{left}(v_a) = v_b$, although $\text{par}(v_a) \neq \text{par}(v_b)$. The pointers let us retrieve some information about the underlying nodes, including their signatures. A pointer to the leftmost (rightmost) leaf of $\overline{\mathcal{T}}(w)$ can also be efficiently created. The full interface is described in Section 4.

In order to implement the pointers to $\overline{\mathcal{T}}(w)$ trees, we first introduce pointers to compressed trees. The set of allowed operations on these pointers is more limited, but sufficient to implement the needed pointers to $\overline{\mathcal{T}}(w)$ trees in a black-box fashion. Each pointer to a tree

[5] Recall that B is the machine word size. The data structure never uses bits $h_i(S)$ for $i > B$. To be precise, it fails before they ever become necessary.

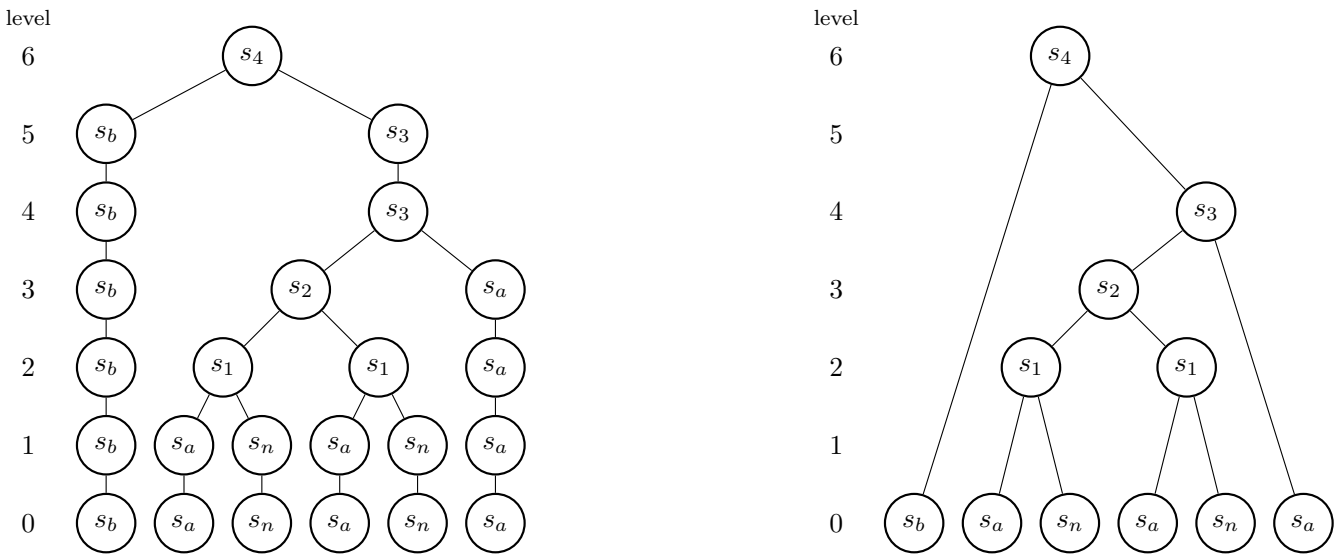

Figure 1: Let $w = \texttt{banana}$. Left picture shows the tree $\overline{\mathcal{T}}(w)$, whereas the right one depicts $\mathcal{T}(w)$. The signature stored in every node is written inside the circle representing the node. We have $str(s_b) = \texttt{b}$, $str(s_a) = \texttt{a}$, $str(a_n) = \texttt{n}$, $str(s_1) = \texttt{an}$, $str(s_2) = \texttt{anan}$, $str(s_3) = \texttt{anana}$, and $str(s_4) = \texttt{banana}$. Moreover, $s_b \to \texttt{b}$, $s_a \to \texttt{a}$, $s_n \to \texttt{n}$, $s_1 \to s_a s_n$, $s_2 \to s_1^2$, $s_3 \to s_2 s_a$, and $s_4 \to s_b s_3$. Note that the parent of the rightmost leaf has level 1 in $\overline{\mathcal{T}}(w)$, but the parent of the same leaf in $\mathcal{T}(w)$ has level 4.

$\mathcal{T}(w)$ is implemented as a stack that represents the path from the root of $\mathcal{T}(S)$ to the pointed node. To represent this path efficiently, we replace each subpath that repeatedly descends to the leftmost child by a single entry in the stack (and similarly for paths descending to rightmost children). This idea can be traced back to a work of Gąsieniec et al. [15], who implemented constant-time forward iterators on grammar-compressed strings; see also [28] for a recent generalization to grammar-compressed trees. By using a fully-persistent stack, we are able to create a pointer to a neighbor node in constant time, without destroying the original pointer. However, in order to do this efficiently, we need to quickly handle queries about the leftmost (or rightmost) descendant of a node $v \in \mathcal{T}(w)$ at a given depth, intermixed with insertions of new signatures into $\bar{\Sigma}(\mathcal{G})$. To achieve that, we use a data structure for finding level ancestors in dynamic trees [2]. This structure allows adding new leaves and querying for the k^{th} ancestor in worst-case constant-time. To use it for retrieving the leftmost descendant of a node $v \in \mathcal{T}(w)$ at a given depth, we therefore introduce another tree, where we make the left child of v (if any) its parent, so that adding new signatures translates into adding new leaves. The details of this construction are provided in Section 4.

3.4 Comparison to Previous Approaches [1, 29]

We first note that our parsing algorithm is simpler than the corresponding procedures in [1, 29]. In particular, we may determine if there is a block boundary between two symbols just by inspecting their values. In the construction of [1, 29], this requires inspecting $\Theta(\log^* n)$ surrounding symbols.

However, the use of randomness in our construction poses some serious challenges, mostly because the size of the uncompressed parse tree $\overline{\mathcal{T}}(w)$ can be $\Omega(n \log n)$ for a string w of length n with non-negligible probability. Consider, for example, the string $w = (\texttt{ab})^{n/2}$. With probability 4^{-k} we have $h_i(\texttt{a}) = h_i(\texttt{b}) = 1$ for $0 \le i < k$, and consequently $\overline{\mathcal{T}}(w)$ contains at least $|w| \cdot k$ nodes. Hence, implementing $\texttt{make\_string}(w)$ in time proportional to $|w|$ requires more care.

Another problem is that even prepending a single letter $\texttt{a}$ to a string w results in a string $\texttt{a} \cdot w$ whose uncompressed parse tree $\overline{\mathcal{T}}(\texttt{a} \cdot w)$ might differ from $\overline{\mathcal{T}}(w)$ by $\Omega(\log^2 n)$ nodes (with non-negligible probability $n^{-\varepsilon}$). In the sample string considered above, the strings $\overline{\textsc{Shrink}_i}(w)$ and $\overline{\textsc{Shrink}_i}(\texttt{a} \cdot w)$ differ by a prefix of length $\Omega(i)$ for $0 \le i < k$ with probability 4^{-k}. In the deterministic construction of [29], the corresponding strings may only differ by $O(\log^* n)$ symbols. In the randomized construction of [29], the difference is of constant size. As a result, in [1, 29], when the string $\texttt{a} \cdot w$ is added to the data structure, the modified prefixes of (the counterpart of) $\overline{\textsc{Shrink}_i}(\texttt{a} \cdot w)$ can be computed explicitly, which is not feasible in our case.

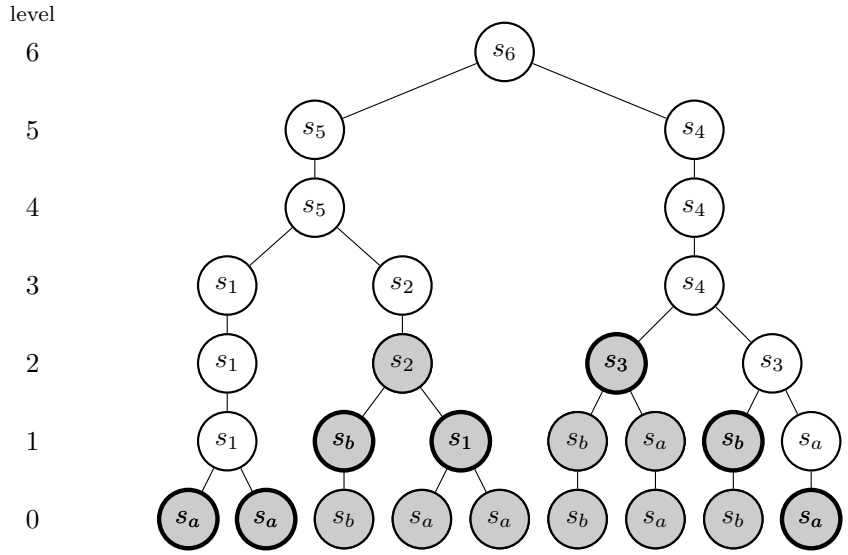

Figure 2: An uncompressed parse tree $\overline{\mathcal{T}}(w)$ for $w = \texttt{aabaababa}$. Shaded nodes are context-insensitive. For example, the leftmost node at level 1 is not context insensitive, because it would not be preserved if we prepended w with `a`. On the other hand, if we prepended w with `b`, then the leftmost node at level 1 would be preserved, but the middle node at level 3 would not. Nodes with thick boundaries represent the context-insensitive layer constructed by the algorithm following the left(par$(\cdot)$) path from the rightmost leaf and, symmetrically, the right(par$(\cdot)$) path from the leftmost leaf. The corresponding decomposition is $s_a s_a s_b s_1 s_3 s_b s_a$. Note that there is a shorter decomposition: $s_a s_a s_2 s_3 s_b s_a$.

To address the first problem, our grammar is based on the compressed parse trees $\mathcal{T}(w)$ and we operate on the uncompressed trees $\overline{\mathcal{T}}(w)$ only using constant-time operations on the pointers. In order to keep the implementation of `split` and `concat` simple despite the second issue, we additionally introduce the notion of context-insensitive decomposition, which captures the part of $\overline{\mathcal{T}}(w)$ which is preserved in the parse tree of every superstring of w (including $\texttt{a} \cdot w$). A related concept (of *common sequences* or *cores*) appeared in a report by Sahinalp and Vishkin [37] and was later used in several papers including a recent a work by Nishimoto et al. [32].

Consequently, while the parsing is simpler compared to the previous constructions, and the general idea for supporting `split` and `concat` is similar, obtaining the desired running time requires deeper insight and more advanced techniques and data-structural tools.

3.5 Context-Insensitive Nodes In this section we introduce the notion of context-insensitive nodes of $\overline{\mathcal{T}}(w)$ to express the fact that a significant part of $\overline{\mathcal{T}}(w)$ is "preserved" in the uncompressed parse trees of superstrings of w. For example, if we concatenate two strings w and y from $\mathcal{W}$, it suffices to take care of the part of $\overline{\mathcal{T}}(w \cdot y)$ which is not "present" in $\overline{\mathcal{T}}(w)$ or $\overline{\mathcal{T}}(y)$. We formalize this intuition as follows.

Consider a node v of $\overline{\mathcal{T}}(w)$, which represents a particular fragment $w[i..j]$ of w. For every fixed strings x, y this fragment can be naturally identified with a fragment of the concatenation $x{\cdot}w{\cdot}y$. If $\overline{\mathcal{T}}(xwy)$ contains a node representing that fragment, we say that v is *preserved* in the *extension* $x{\cdot}w{\cdot}y$. If v is preserved for every such extension, we call it *context-insensitive*; see Fig. 2 for an example. A weaker notion of *left* and *right* context-insensitivity is defined to impose a node to be preserved in all *left* and all *right extensions*, i.e., extensions with $y = \varepsilon$ and $x = \varepsilon$, respectively.

The following lemma captures the most important properties of context-insensitive nodes. Its proof can be found in Section 5, where a slightly stronger result appears as Corollary 5.4.

Lemma 3.9. *Let v be a node of $\overline{\mathcal{T}}(w)$. If v is right context-insensitive, so are nodes* left(v), left(par(v))*, and all children of v. Symmetrically, if v is left context-insensitive, so are nodes* right(v), right(par(v))*, and all children of v. Moreover, if v is both left context-insensitive and right context-insensitive, then it is context-insensitive.*

We say that a collection L of nodes in $\overline{\mathcal{T}}(w)$ or in $\mathcal{T}(w)$ forms a *layer* if every leaf has exactly one ancestor in L. The natural left-to-right order on $\overline{\mathcal{T}}(w)$ lets us treat

every layer as a sequence of nodes. The sequence of their signatures is called the *decomposition* corresponding to the layer; see Fig. 2 for an example. Note that a single decomposition may correspond to several layers in $\overline{\mathcal{T}}(w)$, but only one of them (the lowest one) does not contain nodes with exactly one child.

If a layer in $\overline{\mathcal{T}}(w)$ is composed of (left/right) context-insensitive nodes, we also call the corresponding decomposition (resp. left/right) context-insensitive. The following fact relates context-insensitivity with concatenation of strings and their decompositions. It is proven in Section 5.

Fact 3.10. *Let D be a right context-insensitive decomposition of w and let D' be a left-context insensitive decomposition of w'. The concatenation $D \cdot D'$ is a decomposition of ww'. Moreover, if D and D' are context-insensitive, then $D \cdot D'$ is also context-insensitive.*

The context-insensitive decomposition of a string may need to be linear in its size. For example, consider a string a^k for $k > 1$. We have $|\overline{\text{SHRINK}_1}(a^k)| = 1$. Thus, $\overline{\mathcal{T}}(a^k)$ contains a root node with k children. At the same time, the root node is not preserved in the tree $\overline{\mathcal{T}}(a^k \cdot a) = \overline{\mathcal{T}}(a^{k+1})$, as the roots of $\overline{\mathcal{T}}(a^k)$ and $\overline{\mathcal{T}}(a^{k+1})$ contain distinct signatures. Thus, the shortest context-insensitive decomposition of a^k consists of k signatures.

However, as we show, each string w has a context-insensitive decomposition, whose run-length encoding has length $O(\text{DEPTH}(w))$, which is $O(\log n)$ with high probability. This fact is proved in Section 6.1.

Let us briefly discuss, how to obtain such a decomposition. Consider a tree $\overline{\mathcal{T}}(w)$. We start at the leftmost leaf u_0 and the rightmost leaf v_0 of this tree. Then, we repeatedly move to $u_{i+1} = \text{right}(\text{par}(u_i))$ and $v_{i+1} = \text{left}(\text{par}(v_i))$. We continue as long as u_{i+1} is to the left of v_{i+1} (or $u_{i+1} = v_{i+1}$). This procedure maintains an invariant that all nodes between u_i and v_i are context-insensitive. For $i = 0$ this is trivial, since all leaves are context-insensitive. The inductive step follows from Lemma 3.9: u_i and every node to the right is left context insensitive, while v_i and every node to the left is right context insensitive. The constructed context insensitive layer contains, for each level i, all right siblings of u_i and all left siblings of v_i (including u_i and v_i); see Fig. 2 for an example. In the last level, it may happen that u_i and v_i are siblings, in which case we include all nodes between u_i and v_i (inclusive). Since the children of every node have at most two distinct signatures, the run-length encoding of the corresponding decomposition has length up to $4 \cdot \text{DEPTH}(w)$. It is easy to construct it in $O(\text{DEPTH}(w))$ time while we traverse the tree $\overline{\mathcal{T}}(w)$ using pointers. This construction can be extended to computing context-insensitive decompositions of a given substring of w, which is used in the `split` operation.

3.6 Updating the Collection In this section we show how to use context-insensitive decompositions in order to add new strings to the collection. It turns out that the toolbox which we have developed allows us to handle `make_string`, `concat` and `split` operations in a very clean and uniform way.

Consider a `concat`(w_1, w_2) operation. We first compute the run-length-encoded the context-insensitive decompositions of w_1 and w_2. By Fact 3.10, their concatenation D is a decomposition of $w_1 \cdot w_2$. Moreover, each signature in D belongs to $\bar{\Sigma}(\mathcal{G})$. Let L be the layer in $\mathcal{T}(w_1 \cdot w_2)$ that corresponds to D. In order to ensure that $\mathcal{G}$ represents $w_1 \cdot w_2$, it suffices to add to $\mathcal{G}$ the signatures of all nodes of $\mathcal{T}(w_1 \cdot w_2)$ located above L. For this, we repeatedly find all lowest-level nodes of $\mathcal{T}(w_1 \cdot w_2)$ located above L and modify L so that it contains these nodes instead of their children. For each two consecutive nodes in L, we suppose that they are siblings and determine the level of their hypothetical parent (which depends only on the signatures of the two nodes). It turns out that pairs minimizing the level are indeed siblings and their parents are the sought nodes. We actually simulate this algorithm using the run-length encoding of the decomposition D corresponding to L instead of L itself. A careful implementation allows for running time linear in the run-length encoding of D and the depth of the resulting tree $\mathcal{T}(w_1 \cdot w_2)$.

For details, see Lemma 6.7.

Thus, given the run-length encoding of decomposition of $w_1 \cdot w_2$ we can use the above algorithm to add to $\mathcal{G}$ the signatures of all nodes of $\mathcal{T}(w_1 \cdot w_2)$ that lie above L. The same procedure can be used to handle a `split` operation (we compute a context-insensitive decomposition of the prefix and suffix and run the algorithm on each of them) or even a `make_string` operation. In the case of `make_string` operation, the sequence of letters of w is a trivial decomposition of w, so by using the algorithm, we can add it to the data structure in $O(|w| + \text{DEPTH}(w))$ time. Combined with the efficient computation of the run-length encodings of context-insensitive decompositions, this gives an efficient way of handling all updates. See Section 6.3 for precise theorem statements.

4 Navigating through the Parse Trees

In this section we describe the notion of *pointers* to trees $\mathcal{T}(S)$ and $\overline{\mathcal{T}}(S)$ that has been introduced in Section 3.3 in detail. Although in the following sections we mostly make use of the pointers to $\overline{\mathcal{T}}(S)$ trees, the pointers to compressed parse trees are essential to obtain an efficient implementation of the pointers to uncompressed trees.

Recall that a pointer is a handle that can be used

to access some node of a tree $\mathcal{T}(S)$ or $\overline{\mathcal{T}}(S)$ in constant time. The pointers for trees $\mathcal{T}(S)$ and $\overline{\mathcal{T}}(S)$ are slightly different and we describe them separately.

The pointer to a node of $\mathcal{T}(S)$ or $\overline{\mathcal{T}}(S)$ can be created for any string represented by $\mathcal{G}$ and, in fact, for any existing signature s. Once created, the pointer points to some node v and cannot be moved. The information on the parents of v (and, in particular, the root s) is maintained as a part of the internal state of the pointer. The state can possibly be shared with other pointers.

In the following part of this section, we first describe the pointer interface. Then, we move to the implementation of pointers to $\mathcal{T}(S)$ trees. Finally, we show that by using the pointers to $\mathcal{T}(S)$ trees in a black-box fashion, we may obtain pointers to $\overline{\mathcal{T}}(S)$ trees.

4.1 The Interface of Tree Pointers We now describe the pointer functions we want to support. Let us begin with pointers to the uncompressed parse trees. Fix a signature $s \in \bar{\Sigma}(\mathcal{G})$ and let $w = str(S)$, where s corresponds to the symbol S. First of all, we have three primitives for creating pointers:

- $\texttt{root}(s)$ – a pointer to the root of $\overline{\mathcal{T}}(S)$.
- $\texttt{begin}(s)$ ($\texttt{end}(s)$) – a pointer to the leftmost (rightmost) leaf of $\overline{\mathcal{T}}(S)$.

We also have six functions for navigating through the tree. These functions return appropriate pointers or `nil` if the respective nodes do not exist. Let P be a pointer to $v \in \overline{\mathcal{T}}(S)$.

- $\texttt{parent}(P)$ – a pointer to $\mathrm{par}(v)$.
- $\texttt{child}(P, k)$ – a pointer to the k-th child of v in $\overline{\mathcal{T}}(S)$.
- $\texttt{right}(P)$ ($\texttt{left}(P)$) – a pointer to the node $\mathrm{left}(v)$ ($\mathrm{right}(v)$, resp.).
- $\texttt{rskip}(P, k)$ ($\texttt{lskip}(P, k)$) – let $v = v_1, v_2, \ldots$ be the nodes to the right (left) of v in the $\texttt{level}(P)$-th level in $\overline{\mathcal{T}}(S)$, in the left to right (right to left) order and let K be the largest integer such that all nodes $v_1, \ldots, v_K$ correspond to the same signature as v. If $k \leq K$ and v_{k+1} exists, return a pointer to v_{k+1}, otherwise return `nil`.

Note that the nodes $\mathrm{right}(v)$ and $\mathrm{left}(v)$ might not have a common parent with the node v.

Additionally, we can retrieve information about the pointed node v using several functions:

- $\texttt{sig}(P)$ – the signature corresponding to v, i.e., $\mathrm{sig}(v)$.
- $\texttt{degree}(P)$ – the number of children of v.
- $\texttt{index}(P)$ – if $v \neq s$, an integer k such that $P = \texttt{child}(\texttt{parent}(P, k))$, otherwise 0.
- $\texttt{level}(P)$ – the level of v, i.e., the number of edges on the path from v to the leaf of $\overline{\mathcal{T}}(S)$ (note that this may not be equal to $level(\texttt{sig}(P))$, for example if v has a single child that represents the same signature as v).
- $\texttt{repr}(P)$ – a pair of indices (i, j) such that v represents $w[i..j]$.
- $\texttt{rext}(P)$ ($\texttt{lext}(P)$) – let $v = v_1, v_2, \ldots$ be the nodes to the right (left) of v in the $\texttt{level}(P)$-th level in $\overline{\mathcal{T}}(S)$, in the left to right (right to left) order. Return the maximum K such that all nodes $v_1, \ldots, v_K$ correspond to the same signature as v.

Moreover, the interface provides a way to check if two pointers P, Q to the nodes of the same tree $\overline{\mathcal{T}}(S)$ point to the same node: $P = Q$ is clearly equivalent to $\texttt{repr}(P) = \texttt{repr}(Q)$.

The interface of pointers for trees $\mathcal{T}(S)$ is more limited. The functions `root`, `begin`, `end`, `parent`, `child`, `sig`, `degree`, `index` and `repr` are defined analogously as for uncompressed tree pointers. The definitions of `right` and `left` are slightly different. In order to define these functions, we introduce the notion of the *level-l layer* of $\mathcal{T}(S)$. The level-l layer $\Lambda_s(l)$ (also denoted by $\Lambda_s(l)$) is a list of nodes u of $\mathcal{T}(S)$ such that $level(\mathrm{sig}(u)) \leq l$ and either $\mathrm{sig}(u) = s$ or $level(\mathrm{sig}(\mathrm{par}_{\mathcal{T}(s)}(u))) > l$, ordered from the leftmost to the rightmost nodes. In other words, we consider a set of nodes of level l in $\overline{\mathcal{T}}(S)$ and then replace each node with its first descendant (including itself) that belongs to $\mathcal{T}(S)$. This allows us to define `left` and `right`:

- $\texttt{right}(P, l)$ ($\texttt{left}(P, l)$) – assume $v \in \Lambda_s(l)$. Return a pointer to the node to the right (left) of v on $\Lambda_s(l)$, if such node exists.

For example, assume that P points to the leftmost leaf in $\mathcal{T}(w)$ in Fig. 1. Then $\texttt{right}(P, 0)$ is the pointer to the leaf representing the first a, but $\texttt{right}(P, 2)$ is the pointer to the node representing the string *anan*.

Remark 4.1. *The functions* `right` *and* `left` *operating on $\mathcal{T}(S)$ trees (defined above) are only used for the purpose of implementing the pointers to uncompressed parse trees.*

4.2 Pointers to Compressed Parse Trees As we later show, the pointers to $\overline{\mathcal{T}}(S)$ trees can be implemented using pointers to $\mathcal{T}(S)$ in a black-box fashion. Thus, we first describe the pointers to trees $\mathcal{T}(S)$.

Lemma 4.2. *Assume that the grammar $\mathcal{G}$ is growing over time. We can implement the entire interface of pointers to trees $\mathcal{T}(S)$ where $s \in \bar{\Sigma}(\mathcal{G})$ in such a way that all operations take $O(1)$ worst-case time. The additional time needed to update the shared pointer state is also $O(1)$ per creation of a new signature.*

Before we embark on proving this lemma, we describe auxiliary data structures constituting the shared infrastructure for pointer operations. Let $s \in \bar{\Sigma}(\mathcal{G})$ and let $\mathcal{T}_{\geq l}(s)$ be the tree $\mathcal{T}(S)$ with the nodes of levels less than l removed. Now, define $\texttt{First}(s, l)$ and $\texttt{Last}(s, l)$ to be the signatures corresponding to the leftmost and rightmost leaves of $\mathcal{T}_{\geq l}(s)$, respectively. We aim at being able to compute functions `First` and `Last` in constant time, subject to insertions of new signatures into $\bar{\Sigma}(\mathcal{G})$.

Lemma 4.3. *We can maintain a set of signatures $\bar{\Sigma}(\mathcal{G})$ subject to signature insertions and queries $\texttt{First}(s, l)$ and $\texttt{Last}(s, l)$, so that both insertions and queries take constant time.*

Proof. For brevity, we only discuss the implementation of `First`, as `Last` can be implemented analogously. We form a forest F of rooted trees on the set $\bar{\Sigma}(\mathcal{G})$, such that each $x \in \Sigma$ is a root of some tree in F. Let $s \in \bar{\Sigma}(\mathcal{G}) \setminus \Sigma$. If $s \to s_l s_r$, then s_l is a parent of s in F and if $s \to s_p^k$, where $k \in \mathbb{N}$, then s_p is a parent of s. For each $s \in \bar{\Sigma}(\mathcal{G})$ we also store a $2B$-bit mask $\texttt{b}(s)$ with the i-th bit set if some (not necessarily proper) ancestor of s in F has level equal to i. The mask requires a constant number of machine words. When a new signature s' is introduced, a new leaf has to be added to F. Denote by s_p the parent of s' in F. The mask $\texttt{b}(s')$ can be computed in $O(1)$ time: it is equal $\texttt{b}(s_p) + 2^{level(s')}$. We also store a dictionary `bcnt` that maps each introduced value $\texttt{b}(s)$ to the number of bits set in $\texttt{b}(s)$. Note that since $level(s') > level(s_p)$, $\texttt{bcnt}(\texttt{b}(s')) = \texttt{bcnt}(\texttt{b}(s_p)) + 1$.

Now, it is easy to see that $\texttt{First}(s, l)$ is the highest (closest to the root) ancestor a of s in F, such that $level(a) \geq l$. In order to find a, we employ the *dynamic level ancestor* data structure of Alstrup et al. [2]. The data structure allows us to maintain a dynamic forest subject to leaf insertions and queries for the k-th nearest ancestor of any node. Both the updates and the queries are supported in worst-case $O(1)$ time. Thus, we only need to find such number k that a is the k-th nearest ancestor of s in F. Let $\texttt{b}'(s)$ be the mask $\texttt{b}(s)$ with the bits numbered from l to $2B - 1$ cleared. The mask $\texttt{b}'(s)$ can be computed in constant time with standard bitwise operations. Note that $\texttt{b}'(s) = \texttt{b}(a')$ for some ancestor a' of s and as a result `bcnt` contains the key $\texttt{b}'(s)$. Hence, $\texttt{bcnt}(\texttt{b}'(s))$ is the number of bits set in $\texttt{b}'(s)$. Also, $\texttt{b}'(s)$ denotes the number of ancestors of s at levels less than l. Thus $k = \texttt{bcnt}(\texttt{b}(s)) - \texttt{bcnt}(\texttt{b}'(s))$ is the number of ancestors of s at levels not less than l. Consequently, the $(k-1)$-th nearest ancestor of s is the needed node a. □

Equipped with the functions `First` an `Last`, we now move to the proof of Lemma 4.2.

Proof of Lemma 4.2. Let w be such that $\mathcal{T}(w) = \mathcal{T}(S)$. Recall that we are interested in pointers to $\mathcal{T}(S)$ for some fixed $s \in \bar{\Sigma}(\mathcal{G})$. A functional stack will prove to be useful in the following description. For our needs, let the functional stack be a data structure storing values of type $\mathcal{A}$. Functional stack has the following interface:

- $\texttt{empty}()$ – return an empty stack
- $\texttt{push}(S, val)$ – return a stack S' equal to S with a value $val \in \mathcal{A}$ on top.
- $\texttt{pop}(S)$ – return a stack S' equal to S with the top value removed.
- $\texttt{top}(S)$ – return the top element of the stack S.

The arguments of the above functions are kept intact. Such stack can be easily implemented as a functional list [33].

Denote by r the root of $\mathcal{T}(S)$. The pointer P to v is implemented as a functional stack containing some of the ancestors of v in $\mathcal{T}(S)$, including v and r, accompanied with some additional information (a `nil` pointer is represented by an empty stack). The stack is ordered by the levels of the ancestors, with v lying always at the top. Roughly speaking, we omit the ancestors being the internal nodes on long subpaths of $r \to v$ descending to the leftmost (rightmost) child each time. Gąsieniec et al. [15] applied a similar idea to implement forward iterators on grammar-compressed strings (traversing only the leaves of the parse tree). Independently with our work, Lohrey et al. [28] introduced bidirectional iterators (still traversing the leaves only), using some techniques similar to ours. In fact, the construction in [28] is designed to traverse trees derived by a grammar producing trees rather than strings.

More formally, let A_v be the set of ancestors of v in $\mathcal{T}(S)$, including v. The stack contains an element per each $a \in A_v \setminus (X_l \cup X_r)$, where X_l (X_r respectively) is a set of such ancestors $b \in A_v \setminus \{r\}$ that b simultaneously:

(1) contains v in the subtree rooted at its leftmost (rightmost resp.) child,

(2) is the leftmost (rightmost) child of $\text{par}_{\mathcal{T}(s)}(b)$.

The stack elements come from the set $\bar{\Sigma}(\mathcal{G}) \times (\{\texttt{L}, \texttt{R}, \bot\} \cup \mathbb{N}_+) \times \mathbb{N}$. Let for any node u of $\mathcal{T}(S)$, $\delta(u)$ denote an integer such that u represents the substring of w

starting at $\delta(u)+1$. If $a = r$, then the stack contains the entry $(s, \perp, 0)$. If a is the leftmost child of its parent in $\mathcal{T}(S)$, then the stack contains the entry $(\mathrm{sig}(a), \mathtt{L}, \delta(a))$. Similarly, for a which is the rightmost child of $\mathrm{par}_{\mathcal{T}(s)}(a)$, the stack contains $(\mathrm{sig}(a), \mathtt{R}, \delta(a))$. Otherwise, a is the i-th child of some k-th power signature, where $i \in (1, k)$, and the stack contains $(\mathrm{sig}(a), i, \delta(a))$ in this case. Note that the top element of the stack is always of the form $(\mathrm{sig}(v), *, *)$.

Having fixed the internal representation of a pointer P, we are now ready to give the implementation of the $\mathcal{T}(S)$-pointer interface. In order to obtain $\mathtt{sig}(P)$, we take the first coordinate of the top element of P. The $\mathtt{degree}(P)$ can be easily read from $\mathcal{G}$ and $\mathtt{sig}(P)$.

Let us now describe how $\mathtt{index}(P)$ works. If P points to the root, then the return value is clearly 0. Otherwise, let the second coordinate of the top element be y. We have that $y \in \{\mathtt{L}, \mathtt{R}, 2, 3, 4, \ldots\}$. If $y \notin \{\mathtt{L}, \mathtt{R}\}$, we may simply return y. Otherwise, if $y = \mathtt{L}$, $\mathtt{index}(P)$ returns 1. When $y = \mathtt{R}$, we return $\mathtt{degree}(\mathtt{parent}(P))$.

The third "offset" coordinate of the stack element $(\mathrm{sig}(v), y, \delta(v))$ is only maintained for the purpose of computing the value $\mathtt{repr}(P)$, which is equal to $(\delta(v) + 1, \delta(v) + \mathit{length}(\mathrm{sig}(v)))$. We show that the offset $\delta(v)$ of any entry depends only on v, y and the previous preceding stack element $(\mathrm{sig}(v'), *, \delta(v'))$ (the case when v is the root of $\mathcal{T}(S)$ is trivial). Recall that v' is an ancestor of v in $\mathcal{T}(S)$.

- If $y = \mathtt{L}$, then on the path $v' \to v$ in $\mathcal{T}(S)$ we only descend to the leftmost children, so $\delta(v) = \delta(v')$.
- If $y = j$, then $\mathrm{sig}(v') \to s_1^k$, $j \in (1, k)$ and v' is the parent of v in $\mathcal{T}(S)$. Hence, $\delta(v) = \delta(v') + (j-1) \cdot \mathit{length}(s_1)$ in this case.
- If $y = \mathtt{R}$, then on the path $v' \to v$ in $\mathcal{T}(S)$ we only descend to the rightmost children, so $\delta(v) = \delta(v') + \mathit{length}(\mathrm{sig}(v')) - \mathit{length}(\mathrm{sig}(v))$.

For the purpose of clarity, we hide the third coordinate of the stack element in the below discussion, as it can be computed according to the above rules each time `push` is called. Thus, in the following we assume that the stack contains elements from the set $\bar{\Sigma}(\mathcal{G}) \times (\{\mathtt{L}, \mathtt{R}, \perp\} \cup \mathbb{N}_+)$.

The implementation of the primitives `root`, `begin` and `end` is fairly easy: $\mathtt{root}(s)$ returns $\mathtt{push}(\mathtt{empty}(), (s, \perp))$, $\mathtt{begin}(s)$ returns the pointer $\mathtt{push}(\mathtt{root}(s), (\mathtt{First}(s, 0), \mathtt{L}))$, whereas $\mathtt{end}(s)$ returns $\mathtt{push}(\mathtt{root}(s), (\mathtt{Last}(s, 0), \mathtt{R}))$.

It is beneficial to have an auxiliary function `collapse`. If the stack P contains (s_1, y) as the top element and (s_2, y) as the next element, where $y \in \{\mathtt{L}, \mathtt{R}\}$, $\mathtt{collapse}(P)$ returns a stack P' with (s_2, y) removed, i.e. $\mathtt{push}(\mathtt{pop}(\mathtt{pop}(P)), \mathtt{top}(P))$. Now, to execute $\mathtt{child}(P, k)$, we compute the k-th child w of v from $\mathcal{G}$ and return $\mathtt{collapse}(\mathtt{push}(P, (\mathrm{sig}(w), z))$, where $z = \mathtt{L}$ if $k = 1$, $z = \mathtt{R}$ if v has exactly k children and $z = k$ otherwise.

The $\mathtt{parent}(P)$ is equal to $\mathtt{pop}(P)$ if $\mathtt{top}(P) = (s_v, k)$, for an integer k and $s_v = \mathrm{sig}(v)$. If, however, $\mathtt{top}(P) = (s_v, \mathtt{L})$, then the actual parent of v might not lie on the stack. Luckily, we can use the `First` function: if $(s_u, *)$ is the second-to top stack entry, then the needed parent is $\mathtt{First}(s_u, \mathit{level}(s_v) - 1)$. Thus, we have $\mathtt{parent}(P) = \mathtt{collapse}(\mathtt{push}(\mathtt{pop}(P), (\mathtt{First}(s_u, \mathit{level}(s_v) - 1), \mathtt{L})))$ in this case. The case when $\mathtt{top}(P) = (s_v, \mathtt{R})$ is analogous – we use `Last` instead of `First`.

To conclude the proof, we show how to implement $\mathtt{right}(P, l)$ (the implementation of $\mathtt{left}(P, l)$ is symmetric). We first find Q – a pointer to the nearest ancestor a of v such that v is not in the subtree rooted at a's rightmost-child. To do that, we first skip the "rightmost path" of nearest ancestors of v by computing a pointer P' equal to P if $\mathtt{top}(P)$ is not of the form $(*, \mathtt{R})$ and $\mathtt{pop}(P)$ otherwise. The pointer Q can be now computed by calling $\mathtt{parent}(P')$. Now we can compute the index j of the child b of a such that b contains the desired node: if $\mathtt{top}(P') = (*, \mathtt{L})$, then $j = 2$, and if $\mathtt{top}(P') = (*, k)$, then $j = k + 1$. The pointer R to b can be obtained by calling $\mathtt{collapse}(\mathtt{push}(Q, \mathtt{sig}(\mathtt{child}(Q, j)), y))$, where y is equal to $\mathtt{R}$ if b is the rightmost child of a and j otherwise. The exact value of the pointer $\mathtt{right}(P, l)$ depends on whether b has level no more than l. If so, then $\mathtt{right}(P, l) = R$. Otherwise, $\mathtt{right}(P, l) = \mathtt{push}(R, (\mathtt{First}(\mathtt{sig}(R), l), \mathtt{L}))$.

To sum up, each operation on pointers take worst-case $O(1)$ time, as they all consist of executing a constant number of functional stack operations. □

4.3 Pointers to Compressed Parse Trees

We now show the implementation of pointers to uncompressed parse trees.

Lemma 4.4. *Assume that the grammar $\mathcal{G}$ is growing over time. We can implement the entire interface of pointers to trees $\overline{\mathcal{T}}(S)$ where $s \in \bar{\Sigma}(\mathcal{G})$ in such a way that all operations take $O(1)$ worst-case time. The additional worst-case time needed to update the shared pointer state is also $O(1)$ per creation of new signature.*

Proof. To implement the pointers to the nodes of a tree $\overline{\mathcal{T}}(S)$ we use pointers to the nodes of $\mathcal{T}(S)$ (see Lemma 4.2). Namely, the pointer $\overline{P}$ to a tree $\overline{\mathcal{T}}(S)$ is a pair (P, l), where P is a pointer to $\mathcal{T}(S)$ and l is a level.

Recall that $\overline{\mathcal{T}}(S)$ can be obtained from $\mathcal{T}(S)$ by dissolving nodes with one child. Thus, a pointer $\overline{P}$ to a node u of $\overline{\mathcal{T}}(S)$ is represented by a pair (P, l), where:

- If $u \in \mathcal{T}(S)$, P points to u. Otherwise, it points to the first descendant of u in $\overline{\mathcal{T}}(S)$ that is also a node of $\mathcal{T}(S)$,
- l is the level of u in $\overline{\mathcal{T}}(S)$.

Thus, $\texttt{level}(\overline{P})$ simply returns l. The operations $\texttt{root}(s)$ and $\texttt{begin}(s)$ can be implemented directly, by calling the corresponding operations on $\mathcal{T}(S)$. Observe that the desired nodes are also nodes of $\mathcal{T}(S)$. Moreover, thanks to our representation, the return value of $\texttt{sig}(\overline{P})$ is $\texttt{sig}(P)$.

To run $\texttt{parent}(\overline{P})$ we consider two cases. If the parent of $\overline{P}$ is a node of $\mathcal{T}(S)$, then $\texttt{parent}(\overline{P})$ returns $(\texttt{parent}(P), l+1)$. Otherwise, it simply returns $(P, l+1)$. Note that we may detect which of the two cases holds by inspecting the level of $\texttt{parent}(P)$.

The implementation of $\texttt{child}(\overline{P}, k)$ is similar. If the node pointed by $\overline{P}$ is also a node of $\mathcal{T}(S)$, we return $(\texttt{child}(P, k), l-1)$. Otherwise, k has to be equal to 1 and we return $(P, l-1)$.

If $\texttt{parent}(\overline{P})$ does exist and is a node of $\mathcal{T}(S)$, the return value of $\texttt{index}(\overline{P})$ is the same as the return value of $\texttt{index}(P)$. Otherwise, the parent of the node pointed by $\overline{P}$ either has a single child or does not exist, so $\texttt{index}(\overline{P})$ returns 1.

In the function `lext` we have two cases. If l is odd, then $\texttt{lext}(\overline{P}) = 0$, as the odd levels of $\overline{\mathcal{T}}(S)$ do not contain adjacent equal signatures. The same applies to the case when P points to the root of $\mathcal{T}(S)$. Otherwise, the siblings of v constitute a block of equal signatures on the level l. Thus, we return $\texttt{index}(\overline{P}) - 1$. The implementation of `rext` is analogous.

$\texttt{right}(\overline{P})$ returns $(\texttt{right}(P, l), l)$ and $\texttt{left}(\overline{P})$ returns $(\texttt{left}(P, l), l)$.

Finally, $\texttt{right}(\overline{P})$ can be used to implement $\texttt{rskip}(\overline{P}, k)$ (`lskip` is symmetric). If $k = \texttt{rext}(\overline{P}) + 1$, then $\texttt{rskip}(\overline{P}, k) = \texttt{right}(\texttt{rskip}(\overline{P}, k-1))$. Also, $\texttt{rskip}(\overline{P}, 0) = \overline{P}$ holds in a trivial way. In the remaining case we have $k \in [1, \texttt{rext}(\overline{P})]$ and it follows that $\texttt{sig}(\texttt{parent}(\overline{P})) \to \texttt{sig}(\overline{P})^q$ for $q = \texttt{index}(\overline{P}) + \texttt{rext}(\overline{P})$. Thus, we return $\texttt{child}(\texttt{parent}(\overline{P}), \texttt{index}(\overline{P}) + k))$. □

Remark 4.5. *In the following we sometimes use the pointer notation to describe a particular node of $\overline{\mathcal{T}}(S)$. For example, when we say "node P", we actually mean the node $v \in \overline{\mathcal{T}}(S)$ such that P points to v.*

5 Context-Insensitive Nodes

In this section we provide a combinatorial characterization of context-insensitive nodes, introduced in Section 3.5. Before we proceed, let us state an important property of our way of parsing, which is also required to implement longest common prefix queries.

Lemma 5.1. *Let $i \in \mathbb{Z}_{>0}$, $w, w' \in \bar{\Sigma}^*$ and let p be the longest common prefix of w and w'. There exists a decomposition $p = p_\ell p_r$ such that $|\textsc{Rle}(p_r)| \leq 1$ and $\textsc{Shrink}_i(p_\ell)$ is the longest common prefix of $\textsc{Shrink}_i(w)$ and $\textsc{Shrink}_i(w')$.*

Proof. Observe that the longest common prefix of $\textsc{Shrink}_i(w)$ and $\textsc{Shrink}_i(w')$ expands to a common prefix of w and w'. We set p_ℓ to be this prefix and p_r to be the corresponding suffix of p. If $p_r = \varepsilon$ we have nothing more to prove and thus suppose $p_r \neq \varepsilon$.

Note that p_r starts at the last position of p before which both $\textsc{Shrink}_i(w)$ and $\textsc{Shrink}_i(w')$ place block boundaries. Recall that $\textsc{Shrink}_i$ places a block boundary between two positions only based on the characters on these positions. Thus, since the blocks starting with p_r are (by definition of p_ℓ) different in w and w', in one of these strings, without loss of generality in w, this block extends beyond p. Consequently, p_r is a proper prefix of a block. Observe, however, that a block may contain more than one distinct character only if its length is exactly two. Thus, $|\textsc{Rle}(p_r)| = 1$ as claimed. □

Next, let us recall the notions introduced in Section 3.5. Here, we provide slightly more formal definitions. For arbitrary strings w, x, y we say that $x{\cdot}w{\cdot}y$ (which is formally a triple (x, w, y)) is *extension* of w. We often identify the extension with the underlying concatenated string xwy, remembering, however, that a particular occurrence of w is distinguished. If $x = \varepsilon$ or $y = \varepsilon$, we say that $x{\cdot}w{\cdot}y$ is a *right extension* or *left extension*, respectively.

Let us fix an extension $x{\cdot}w{\cdot}y$ of w. Consider a node v at level l of $\overline{\mathcal{T}}(w)$. This node represents a particular fragment $w[i..j]$ which has a naturally corresponding fragment of the extension. We say that a node v' at level l of $\overline{\mathcal{T}}(xwy)$ is a *counterpart* of v with respect to $x{\cdot}w{\cdot}y$ (denoted $v \approx v'$) if $\mathrm{sig}(v) = \mathrm{sig}(v')$ and v' represents the fragment of xwy which naturally corresponds to $w[i..j]$.

Lemma 5.2. *If a level-l node $v \in \overline{\mathcal{T}}(w)$ has a counterpart v' with respect to a right extension $\varepsilon{\cdot}w{\cdot}y$, then* $\mathrm{child}(v, k) \approx \mathrm{child}(v', k)$ *for any integer k,* $\mathrm{left}(v) \approx \mathrm{left}(v')$, *and* $\mathrm{left}(\mathrm{par}(v)) \approx \mathrm{left}(\mathrm{par}(v'))$. *Moreover, if* $\mathrm{sig}(v) \neq \mathrm{sig}(\mathrm{left}(v))$, *then* $\mathrm{par}(\mathrm{left}(v)) \approx \mathrm{par}(\mathrm{left}(v'))$. *In particular, the nodes do not exist for v if and only if the respective nodes do not exist for v'.*

Analogously, if v has a counterpart v' with respect to a left extension $x{\cdot}w{\cdot}\varepsilon$, then $\mathrm{child}(v, k) \approx \mathrm{child}(v', k)$ *for any integer k,* $\mathrm{right}(v) \approx \mathrm{right}(v')$, *and* $\mathrm{right}(\mathrm{par}(v)) \approx \mathrm{right}(\mathrm{par}(v'))$. *Moreover, if* $\mathrm{sig}(v) \neq \mathrm{sig}(\mathrm{right}(v))$, *then* $\mathrm{par}(\mathrm{right}(v)) \approx \mathrm{par}(\mathrm{right}(v'))$.

Proof. Due to symmetry of our notions, we only prove statements concerning the right extension. First, observe that $\text{child}(v,k) \approx \text{child}(v',k)$ simply follows from the construction of a parse tree and the definition of a counterpart.

Next, let us prove that $\text{left}(v) \approx \text{left}(v')$. If $l = 0$, this is clear since all leaves of $\overline{\mathcal{T}}(w)$ have natural counterparts in $\overline{\mathcal{T}}(wy)$. Otherwise, we observe that $\text{left}(u) = \text{left}(\text{par}(\text{child}(u,1)))$ holds for every u. This lets us apply the inductive assumption for the nodes $\text{child}(v,1) \approx \text{child}(v',1)$ at level $l-1$ to conclude that $\text{left}(v) \approx \text{left}(v')$.

This relation can be iteratively deduced for all nodes to the left of v and v', and thus v and v' represent symbols of $\overline{\textsc{Shrink}_l}(w)$ and $\overline{\textsc{Shrink}_l}(wy)$ within the longest common prefix p of these strings. Lemma 5.1 yields a decomposition $p = p_\ell p_r$ where $p' := \textsc{Shrink}_{l+1}(p_\ell)$ is the longest common prefix of $\overline{\textsc{Shrink}_{l+1}}(w)$ and $\overline{\textsc{Shrink}_{l+1}}(wy)$ while $|\textsc{Rle}(p_r)| \le 1$. Clearly, nodes at level $l+1$ representing symbols within this longest common prefix are counterparts. Thus, whenever u represents a symbol within p_ℓ and u' is its counterpart with respect to $\varepsilon{\cdot}w{\cdot}y$, we have $\text{par}(u) \approx \text{par}(u')$.

Observe that $\text{sig}(v) \neq \text{sig}(\text{left}(v))$ means that $\text{left}(v)$ represents a symbol within p_ℓ, and this immediately yields $\text{par}(\text{left}(v)) \approx \text{par}(\text{left}(v'))$. Also, both in $\overline{\mathcal{T}}(w)$ and in $\overline{\mathcal{T}}(wy)$ nodes representing symbols in p_r have a common parent, so $\text{left}(\text{par}(v)) \approx \text{left}(\text{par}(v'))$ holds irrespective of v and v' representing symbols within p_ℓ or within p_r. □

Lemma 5.3. *Let $v \in \overline{\mathcal{T}}(w)$. If v has counterparts with respect to both $x{\cdot}w{\cdot}\varepsilon$ and $\varepsilon{\cdot}w{\cdot}y$, then v has a counterpart with respect to $x{\cdot}w{\cdot}y$.*

Proof. We proceed by induction on the level l of v. All nodes of level 0 have natural counterparts with respect to any extension, so the statement is trivial for $l = 0$, which lets us assume $l > 0$. Let v^L and v^R be the counterparts of v with respect to $x{\cdot}w{\cdot}\varepsilon$ and $\varepsilon{\cdot}w{\cdot}y$, respectively. Also, let $v_1, \ldots, v_k$ be the children of v. Note that these nodes have counterparts $v_1^L, \ldots, v_k^L$ with respect to $x{\cdot}w{\cdot}\varepsilon$ and $v_1^R, \ldots, v_k^R$ with respect to $\varepsilon{\cdot}w{\cdot}y$. Consequently, by the inductive hypothesis they have counterparts $v'_1, \ldots, v'_k$ with respect to $x{\cdot}w{\cdot}y$.

Observe that $v_i^L \approx v'_i$ when xwy is viewed as an extension $\varepsilon{\cdot}xw{\cdot}y$ of xw. Now, Lemma 5.2 gives $\text{right}(v^L) = \text{right}(\text{par}(v_i^L)) \approx \text{right}(\text{par}(v'_i))$ and thus nodes $v'_1, \ldots, v'_k \in \overline{\mathcal{T}}(xwy)$ have a common parent v'. We shall prove that v' is the counterpart of v with respect to $x{\cdot}w{\cdot}y$. For this it suffices to prove that it does not have any children to the left of v'_1 or to the right of v'_k. However, since $\text{right}(v^L) \approx \text{right}(v')$ and v_k^L is the rightmost child of v^L, v'_k must be the rightmost child of v'. Next, observe that $v_i^R \approx v'_i$ when xwy is viewed as an extension $x{\cdot}wy{\cdot}\varepsilon$ of wy. Hence, Lemma 5.2 implies that $\text{left}(v^R) = \text{left}(\text{par}(v_i^R)) \approx \text{left}(\text{par}(v'_i))$, i.e., $\text{left}(v^R) \approx \text{left}(v')$. As v_1^R is the leftmost child of v^R, v'_1 must be the leftmost child of v'. □

A node $v \in \overline{\mathcal{T}}(w)$ is called *context-insensitive* if it is preserved in any extension of v, and *left* (*right*) *context-insensitive*, if it is preserved in any left (resp. right) extension. The following corollary translates the results of Lemmata 5.2 and 5.3 in the language of context-insensitivity. Its weaker version is stated in Section 3.5 as Lemma 3.9.

Corollary 5.4. *Let v be a node of $\overline{\mathcal{T}}(w)$.*

(a) If v is right context-insensitive, so are nodes $\text{left}(v)$, $\text{left}(\text{par}(v))$, *and all children of v. Moreover, if* $\text{sig}(v) \neq \text{sig}(\text{left}(v))$, *then* $\text{par}(\text{left}(v))$ *is also right context-insensitive.*

(b) If v is left context-insensitive, so are nodes $\text{right}(v)$, $\text{right}(\text{par}(v))$, *and all children of v. Moreover, if* $\text{sig}(v) \neq \text{sig}(\text{right}(v))$, *then* $\text{par}(\text{right}(v))$ *is also left context-insensitive.*

(c) If v is both left context-insensitive and right context-insensitive, it is context-insensitive.

We say that a collection L of nodes in $\overline{\mathcal{T}}(w)$ or in $\mathcal{T}(w)$ forms a *layer* if every leaf has exactly one ancestor in L (possibly the leaf itself). Equivalently, a layer is a maximal antichain with respect to the ancestor-descendant relation. Note that the left-to-right order of $\overline{\mathcal{T}}(w)$ gives a natural linear order on L, i.e., L can be treated as a sequence of nodes. The sequence of their signatures is called the *decomposition* corresponding to the layer. Observe a single decomposition may correspond to several layers in $\overline{\mathcal{T}}(w)$, but only one of them does not contain nodes with exactly one child. In other words, there is a natural bijection between decompositions and layers in $\mathcal{T}(w)$.

We call a layer (left/right) context-insensitive if all its elements are (left/right) context-insensitive.

We also extend the notion of context insensitivity to the underlying decomposition. The decompositions can be seen as sequences of signatures. For two decompositions D, D' we define their concatenation $D \cdot D'$ to be the concatenation of the underlying lists. The following fact relates context-insensitivity with concatenation of words and their decompositions.

Fact 5.5. *Let D be a right context-insensitive decomposition of w and let D' be a left-context insensitive decomposition of w'. The concatenation $D \cdot D'$ is a decomposition*

of ww'. Moreover, if D and D' are context-insensitive, then $D \cdot D'$ is also context-insensitive.

Proof. Let L and L' be layers in $\overline{\mathcal{T}}(w)$ and $\overline{\mathcal{T}}(w')$ corresponding to D and D', respectively. Note that ww' can be seen as a right extension $\varepsilon{\cdot}w{\cdot}w'$ of w and as a left extension $w{\cdot}w'{\cdot}\varepsilon$ of w'. Thus, all nodes in $L \cup L'$ have counterparts in $\overline{\mathcal{T}}(ww')$ and these counterparts clearly form a layer. Consequently, $D \cdot D'$ is a decomposition of ww'. To see that $D \cdot D'$ is context-insensitive if D and D' are, it suffices to note that any extension $x{\cdot}ww'{\cdot}y$ of ww' can be associated with extensions $x{\cdot}w{\cdot}w'y$ of w and $xw{\cdot}w'{\cdot}y$ of w'. □

Fact 5.6. *Let $v, v' \in \mathcal{T}(w)$ be adjacent nodes on a layer L. If v and v' correspond to the same signature s, they are children of the same parent.*

Proof. Let $l = \mathit{level}(s)$ and note that both v and v' both belong to $\Lambda_w(l)$, i.e., they represent adjacent symbols of $\overline{\textsc{Shrink}_l}(w)$. However, $\textsc{Shrink}_{l+1}$ never places a block boundary between two equal symbols. Thus, v and v' have a common parent at level $l+1$. □

6 Adding New Strings to the Collection

6.1 Constructing Context-Insensitive Decompositions Recall that the run-length encoding partitions a string into maximal blocks (called *runs*) of equal characters and represents a block of k copies of symbol S by a pair (S, k), often denoted as S^k. For $k = 1$, we simply use S instead of $(S, 1)$ or S^1. In Section 3.1, we defined the Rle function operating on symbols as a component of our parse scheme. Below, we use it just as a way of compressing a sequence of signatures. Formally, the output of the Rle function on a sequence of items is another sequence of items, where each maximal block of k consecutive items x is replaced with a single item x^k.

We store run-length encoded sequences as linked lists. This way we can create a new sequence consisting of k copies of a given signature s (denoted by s^k) and concatenate two sequences A, B (we use the notation $A \cdot B$), both in constant time. Note that concatenation of strings does not directly correspond to concatenation of the underlying lists: if the last symbol of the first string is equal to the first symbol of the second string, two blocks need to be merged.

Decompositions of strings are sequences of symbols, so we may use Rle to store them space-efficiently. As mentioned in Section 3.5, this is crucial for context-insensitive decompositions. Namely, string w turns out to have a context-insensitive decomposition D such that $|\textsc{Rle}(D)| = O(\textsc{Depth}(w))$. Below we give a constructive proof of this result.

Algorithm 1 Compute a context-insensitive decomposition of a string w given by a signature s.

```
1:  function layer(s)
2:      P := begin(s)
3:      Q := end(s)
4:      S = T = ε
5:      while P ≠ Q and parent(P) ≠ parent(Q) do
6:          if sig(P) ≠ sig(right(P)) and parent(P) = parent(right(P)) then
7:              P' := P  ▷ Pointers P' and Q' are set for the purpose of analysis only.
8:              P := parent(P)
9:          else
10:             P' := rskip(P, rext(P))
11:             S := S · sig(P)^rext(P)
12:             P := right(parent(P))
13:         if sig(Q) ≠ sig(left(Q)) and parent(Q) = parent(left(Q)) then
14:             Q' := Q
15:             Q := parent(Q)
16:         else
17:             Q' := lskip(Q, lext(Q))
18:             T := sig(Q)^lext(Q) · T
19:             Q := left(parent(Q))
20:     if sig(P) = sig(Q) then
21:         return S · sig(P)^(index(Q)−index(P)+1) · T
22:     else
23:         return S · sig(P) · sig(Q) · T
```

Lemma 6.1. *Given $s \in \bar{\Sigma}(\mathcal{G})$, one can compute the run-length encoding of a context-insensitive decomposition of $w = \mathit{str}(s)$ in $O(\textsc{Depth}(w))$ time.*

Proof. The decomposition is constructed using procedure `layer` whose implementation is given as Algorithm 1. Let us define $\texttt{right}^k$ as the k-th iterate of `right` (where $\texttt{right}(\texttt{nil}) = \texttt{nil}$) and $\texttt{left}^k$ as the k-th iterate of `left`. Correctness of Algorithm 1 follows from the following claim.

Claim 6.2. *Before the $(l+1)$-th iteration of the **while** loop, P points to a left context-insensitive node at level l and Q points to a right context-insensitive node at level l. Moreover $Q = \texttt{right}^m(P)$ for some $m \geq 0$ and $S \cdot (\texttt{sig}(\texttt{right}^0(P)), \ldots, \texttt{sig}(\texttt{right}^m(P))) \cdot T$ is a context-insensitive decomposition of w.*

Proof. Initially, the invariant is clearly satisfied with $m = |w| - 1 \geq 0$ because all leaves are context-insensitive. Thus, let us argue that a single iteration of the **while** loop preserves the invariant. Note that the iteration is performed only when $m > 0$.

Let P_1 and P_2 be the values of P and Q before the $(l+1)$-th iteration of the **while** loop. Now assume that the loop has been executed for the $(l+1)$-th time. Observe that $P' = \texttt{right}^{m_p}(P_1)$ and $Q' = \texttt{left}^{m_q}(Q_1)$ for some $m_p, m_q \geq 0$ and from $\texttt{parent}(P_1) \neq \texttt{parent}(Q_1)$ it follows that $m_p + m_q \leq m$. Also, S and T are extended so that $S \cdot (\texttt{sig}(\texttt{right}^0(P')), \ldots, \texttt{sig}(\texttt{right}^{m-m_p-m_q}(P')) \cdot T$ is equal to the decomposition we got from the invariant. Moreover, observe that P' points to the leftmost child of the new value of P, and Q' points to the rightmost child of the new Q. Consequently, after these values are set, we have $Q = \texttt{right}^{m'}(P)$ for some $m' \geq 0$ and $S \cdot (\texttt{sig}(\texttt{right}^0(P)), \ldots, \texttt{sig}(\texttt{right}^{m'}(P))) \cdot T$ is a decomposition of w.

Let us continue by showing that the new values of P and Q satisfy the first part of the invariant. Note that $m' \geq 0$ implies P and Q are not set to `nil`. In the **if** block at line 6, Corollary 5.4(b) implies that $\texttt{parent}(P) = \texttt{parent}(\texttt{right}(P))$ indeed points to a left context-insensitive at level $l+1$. The same holds for $\texttt{right}(\texttt{parent}(P))$ in the **else** block. A symmetric argument shows that Q is set to point to a right context-insensitive node at level $i+1$.

Along with Corollary 5.4(c) these conditions imply that all nodes $\texttt{right}^i(P)$ for $0 \leq i \leq m''$ are context-insensitive and thus the new representation is context-insensitive. □

To conclude the proof of correctness of Algorithm 1, we observe that after leaving the main loop the algorithm simply constructs the context-insensitive decomposition mentioned in the claim. Either P points at the root of $\overline{\mathcal{T}}(S)$ or $\texttt{parent}(P) = \texttt{parent}(Q)$. If P points at the root or $\texttt{sig}(P) = \texttt{sig}(Q)$, we only need to add a single run-length-encoded entry between S and T. Otherwise, $\texttt{parent}(P)$ has exactly two children P and Q with different signatures, so we need to add two entries $\texttt{sig}(P)$ and $\texttt{sig}(Q)$ between S and T. □

We conclude with a slightly more general version of Lemma 6.1.

Lemma 6.3. *Given $s \in \bar{\Sigma}(\mathcal{G})$ and lengths $|x|, |y|$ such that $str(s) = xwy$, one can compute in time $O(\textsc{Depth}(xwy))$ a run-length-encoded context-insensitive decomposition of w.*

Proof. The algorithm is the same as the one used for Lemma 6.1. The only technical modification is that the initial values of P and Q need to be set to leaves representing the first and the last position of w in xwy. In general, we can obtain in $O(\textsc{Depth}(xwy))$ time a pointer to the k-th leftmost leaf of $\overline{\mathcal{T}}(xwy)$. This is because each signature can be queried for its length. Note that for nodes with more than two children we cannot scan them in a linear fashion, but instead we need to exploit the fact that these children correspond to fragments of equal length and use simple arithmetic to directly proceed to the right child.

Finally, we observe that by Lemma 5.2, the subsequent values of P and Q in the implementation on $\overline{\mathcal{T}}(xwy)$ are counterparts of the values of P and Q in the original implementation on $\overline{\mathcal{T}}(w)$. □

Remark 6.4. *In order to obtain a left context-insensitive decomposition we do not need to maintain the list T in Algorithm 1 and it suffices to set $Q = \texttt{parent}(Q)$ at each iteration. Of course, Lemma 6.3 needs to be restricted to left extensions $x \cdot w \cdot \varepsilon$. A symmetric argument applies to right context-insensitive decompositions.*

6.2 Supporting Operations We now use the results of previous sections to describe the process of updating the collection when `make_string`, `split`, and `concat` operations are executed. Recall that we only need to make sure that the grammar $\mathcal{G}$ represents the strings produced by these updates. The core of this process is the following algorithm.

Algorithm 2 Iterate through all nodes above a layer L

```
1: function layer(L)
2:     while |L| > 1 do
3:         v := a lowest-level node of T(w) among all
           proper ancestors of L
4:         L := L \ {children of v} ∪ {v}
```

Lemma 6.5. *Algorithm 2 maintains a layer L of $\mathcal{T}(w)$. It terminates and upon termination, the only element of L is the root of $\mathcal{T}(w)$. Moreover, line 3 is run exactly once per every proper ancestor of the initial L in $\mathcal{T}(w)$.*

Proof. Consider line 3 of Algorithm 2. Clearly, $v \notin L$ and every child of v belongs to L. Recall that every node in $\mathcal{T}(w)$, in particular v, has at least two children. Thus, by replacing the fragment of L consisting of all children of v with v we obtain a new layer L', such that $|L'| < |L|$. Hence, we eventually obtain a layer consisting of a single node. From the definition of a layer we have that in every parse tree there is only one single-node layer and its only element is the root of $\mathcal{T}(w)$. The lemma follows. □

We would like to implement an algorithm that is analogous to Algorithm 2, but operates on decompositions, rather than on layers. In such an algorithm in every step we replace some fragment d of a decomposition D with a signature s that generates d. We say that we *collapse* a production rule $s \to d$.

In order to describe the new algorithm, for a decomposition D we define the set of *candidate production rules*

as follows. Let $s_1^{\alpha_1}, \ldots, s_k^{\alpha_k}$ be the run-length encoding of a decomposition D. We define two types of candidate production rules. First, for every $i \in \{1, \ldots, k\}$ we have rules $s \to s_i^{\alpha_i}$. Moreover, for $i \in \{1, \ldots, k-1\}$ we have $s \to s_i s_{i+1}$. The *level* of a candidate production rule $s \to s'_1 \ldots s'_m$ is the level of the signature s.

The following lemma states that collapsing a candidate production rule of minimal level corresponds to executing one iteration of the loop in Algorithm 2.

Lemma 6.6. *Let L be a layer of $\mathcal{T}(w)$ and let D be the corresponding decomposition. Consider a minimum-level candidate production rule $s \to s'_1 \ldots s'_m$ for D. Let $v'_1, \ldots, v'_m$ be the nodes of $\mathcal{T}(w)$ corresponding to $s'_1, \ldots, s'_m$. These nodes are the only children of a node v which satisfies $\mathrm{sig}(v) = s$.*

Proof. Let $l = \mathit{level}(s)$ and let v be a minimum-level node above L. Observe that all children of v belong to L and the sequence of their signatures forms a substring of D. By Fact 5.6 this substring is considered while computing candidate productions, and the level of the production is clearly l. Consequently, all nodes above L have level at least l. In other words, every node of $\Lambda_w(l-1)$ has an ancestor in L. Since $\mathit{level}(s'_i) < l$, we have in particular $v'_i \in \Lambda_w(l-1)$, i.e., the corresponding signatures $s'_1, \ldots, s'_m$ form a substring of $\overline{\textsc{Shrink}_{l-1}}(w)$. We shall prove that they are a single block formed by $\textsc{Shrink}_l$. It is easy to see that no block boundary is placed between these symbols. If l is even, we must have $m = 2$ and $\textsc{Shrink}_l$ simply cannot form larger blocks. Thus, it remains to consider odd l when $s'_i = s'$ for a single signature s'. We shall prove that the block of symbols s' in $\overline{\textsc{Shrink}_{l-1}}(w)$ is not any longer. By Fact 5.6 such a block would be formed by siblings of v'_i. However, for each of them an ancestor must belong to L. Since their proper ancestors coincide with proper ancestors of v'_i, these must be the sibling themselves. This means, however, that a block of symbols s' in D corresponding to $s'_1, \ldots, s'_m$ was not maximal, a contradiction. □

Lemma 6.7. *Let L be a layer in $\mathcal{T}(w)$ and D be its corresponding decomposition. Given a run-length encoding of D of length d, we may implement an algorithm that updates D analogously to Algorithm 2. It runs in $O(d + \textsc{Depth}(w))$ time. The algorithm fails if $\textsc{Depth}(w) > 2 \cdot B$.*

Proof. By Lemma 6.6, it suffices to repeatedly collapse the candidate production rule of the smallest level. We maintain the run-length encoding of the decomposition D as a doubly linked list Y, whose every element corresponds to a signature and its multiplicity. Moreover, for $i = 1, \ldots, 2B$ we maintain a list P_i of candidate production rules of level i. The candidate production rules of higher levels are ignored. With each rule stored in some P_i we associate pointers to elements of Y, that are removed when the rule is collapsed. Observe that collapsing a production rule only affects at most two adjacent elements of Y, since we maintain a run-length encoding of the decomposition.

Initially, we compute all candidate production rules for the initial decomposition D. Observe that this can be done easily in time proportional to d. Then, we iterate through the levels in increasing order. When we find a level i, such that the list P_i is nonempty, we collapse the production rules from the list P_i. Next, we check if there are new candidate production rules and add them to the lists P_i if necessary. Whenever we find a candidate production rule $s \to y$ to collapse, we check if $\mathcal{G}$ already contains a signature s_2 associated with an equivalent production rule $s_2 \to y$ for some s_2. If this is the case, the production rule from $\mathcal{G}$ is collapsed. Otherwise, we collapse the candidate production rule, and add signature s to $\mathcal{G}$. Note that this preserves Invariant 3.7. Also, this is the only place where we modify $\mathcal{G}$.

Note that the candidate production rules are never removed from the lists P_i. Thus, collapsing one candidate production rule may make some production rules in lists P_j $(j > i)$ obsolete (i.e., they can no longer be collapsed). Hence, before we collapse a production rule, we check in constant time whether it can still be applied, i.e., the affected signatures still exist in the decomposition.

As we iterate through levels, we collapse minimum-level candidate production rules. This process continues until the currently maintained decomposition has length 1. If this does not happen before reaching level $2B$, the algorithm fails. Otherwise, we obtain a single signature that corresponds to a single-node layer in $\mathcal{T}(w)$. The only string in such a layer is the root of $\mathcal{T}(w)$, so the decomposition of length 1 contains solely the signature of the entire string w. As a result, when the algorithm terminates successfully, the signature of w belongs to $\bar{\Sigma}(\mathcal{G})$. Hence, the algorithm indeed adds w to $\mathcal{G}$.

Regarding the running time, whenever the list Y is modified by collapsing a production rule, at most a constant number of new candidate production rules may appear and they can all be computed in constant time, directly from the definition. Note that to compute the candidate production rule of the second type, for two distinct adjacent signatures s_1 and s_2 in D we compute the smallest index $i > \max(\mathit{level}(s_1), \mathit{level}(s_2))$, such that $h_{i/2}(s_1) = 0$ and $h_{i/2}(s_2) = 1$. This can be done in constant time using bit operations, because the random bits $h_i(\cdot)$ are stored in machine words. Since we are only interested in candidate production rules of level at most $2B$, we have enough random bits to compute them.

Every time we collapse a production rule, the length of the list Y decreases. Thus, this can be done at most d times. Moreover, we need $O(d)$ time to compute the candidate production rules for the initial decomposition. We also iterate through $O(\textsc{Depth}(w))$ lists P_i. In total, the algorithm requires $O(d + \textsc{Depth}(w))$ time. □

Corollary 6.8. *Let D be a decomposition of a string w. Assume that for every signature s in D we have $s \in \bar{\Sigma}(\mathcal{G})$. Then, given a run-length encoding of D of length d, we may add w to $\mathcal{G}$ in $O(d + \textsc{Depth}(w))$ time.*

The algorithm preserves Invariant 3.7 and fails if $\textsc{Depth}(w) > 2B$.

Using Corollary 6.8 we may easily implement `make_string`, `split` and `concat` operations.

Lemma 6.9. *Let w be a string of length n. We can execute* `make_string`(w) *in $O(n+\textsc{Depth}(w))$ time. This operation fails if $\textsc{Depth}(w) > 2B$.*

Proof. Observe that w (treated as a sequence) is a decomposition of w. Thus, we may compute its run-length encoding in $O(n)$ time and then apply Corollary 6.8. □

Lemma 6.10. *Let w be a string represented by $\mathcal{G}$ and $1 \le k \le |w|$. Then, we can execute a* `split`(w, k) *operation in $O(\textsc{Depth}(w) + \textsc{Depth}(w[1..k]) + \textsc{Depth}(w[k+1..]))$ time. This operation fails if $\max(\textsc{Depth}(w[1..k]), \textsc{Depth}(w[k+1..]) > 2B$.*

Proof. Using Lemma 6.3 we compute the run-length-encoded decompositions of $w[1..k]$ and $w[k+1..]$ in $O(\textsc{Depth}(w))$ time. Then, we apply Corollary 6.8. □

Lemma 6.11. *Let w_1 and w_2 be two strings represented by $\mathcal{G}$. Then, we can execute a* `concat`(w_1, w_2) *operation in $O(\textsc{Depth}(w_1) + \textsc{Depth}(w_2) + \textsc{Depth}(w_1 \cdot w_2))$ time. The operation fails if $\textsc{Depth}(w_1 \cdot w_2) > 2B$.*

Proof. We use Lemma 6.1 to compute the run-length-encoded context-insensitive decompositions of w_1 and w_2 in $O(\textsc{Depth}(w_1) + \textsc{Depth}(w_2))$ time. By Fact 3.10 the concatenation of these decompositions is a decomposition of $w_1 \cdot w_2$. Note that given run-length encodings of two strings, we may easily obtain the run-length encoding of the concatenation of the strings. It suffices to concatenate the encodings and possibly merge the last symbol of the first string with the first symbol of the second one. Once we obtain the run-length encoding of a decomposition of $w_1 \cdot w_2$, we may apply Corollary 6.8 to complete the proof.

Note that for the proof, we only need a right-context insensitive decomposition of w_1 and a left-context insensitive decomposition of w_2. Thus, we apply the optimization of Remark 6.4. However, it does not affect the asymptotic running time. □

6.3 Conclusions The results of this section are summarized in the following theorem.

Theorem 6.12. *There exists a data structure which maintains a family of strings $\mathcal{W}$ and supports* `concat` *and* `split` *in $O(\log n + \log t)$ time,* `make_string` *in $O(n + \log t)$ time, where n is the total length of strings involved in the operation and t is the total input size of the current and prior updates (linear for each* `make_string`*, constant for each* `concat` *and* `split`*). An update may fail with probability $O(t^{-c})$ where c can be set as an arbitrarily large constant. The data structure assumes that total length of strings in $\mathcal{W}$ takes at most B bits in the binary representation.*

Proof. Consider an update which creates a string w of length n. By Lemma 3.2, we have that

$$\mathbb{P}(\textsc{Depth}(w) \le 8(c \ln t + \ln n)) \ge 1 - e^{c \ln t} = 1 - 1/t^c.$$

Hence, an update algorithm may fail as soon as it notices that $\textsc{Depth}(w) > 8(c \ln t + \ln n)$. The total length of strings in $\mathcal{W}$ is at least $\max(n, t)$, and thus $8(c \ln t + \ln n) \le 8(c+1)B \ln 2$. We extend the machine word to $B' = 4(c+1)B \ln 2 = O(B)$ bits, which results in a constant factor overhead in the running time.

This lets us assume that updates implemented according to Lemmata 6.9 to 6.11 do not fail. By Lemma 6.9, `make_string` runs in $O(n + \textsc{Depth}(w))$ time, and, by Lemmata 6.10 and 6.11, both `concat` and `split` run in $O(\log n + \textsc{Depth}(w))$ time. Since we may terminate the algorithms as soon as $\textsc{Depth}(w)$ turns out to be larger than $8(c \ln t + \ln n)$, we can assume that `make_string`(w) runs in $O(n + \log t)$ time, while `concat` and `split` run in $O(\log n + \log t)$ time. □

Note that the failure probability in Theorem 6.12 is constant for the first few updates. Thus, when the data structure is directly used as a part of an algorithm, the algorithm could fail with constant probability. However, as discussed in the full version of the paper [13], restarting the data structure upon each failure is a sufficient mean to eliminate failures while keeping the original asymptotic running time with high probability with respect to the total size of updates. Below, we illustrate this phenomenon in a typical setting of strings with polynomial lengths.

Corollary 6.13. *Suppose that we use the data structure to build a dynamic collection of n strings, each of length at most $\mathrm{poly}(n)$. This can be achieved by a Las Vegas algorithm whose running time with high probability is $O(N + M \log n)$ where N is the total length of strings given to* `make_string` *operations and M is the total number of updates (*`make_string`*,* `concat`*, and* `split`*).*

Proof. We use Theorem 6.12 and restart the data structure in case of any failure. Since the value of n fits in a machine word (as this is the assumption in the word RAM model), by extending the machine word to $O(B)$ bits, we may ensure that the total length of strings in $\mathcal{W}$ (which is poly(n)) fits in a machine word. Moreover, $t = \text{poly}(n)$, so $\log t = O(\log n)$, i.e., the running time of `make_string`$(|w|)$ is $O(|w| + \log n)$ and of `split` and `concat` is $O(\log n)$. We set c in Theorem 6.12 as a sufficiently large constant so that restarting the data structure upon each failure ensures that the overall time bound holds with the desired probability. □

7 Dynamic String Collections: Lower Bounds

In this section, we prove a lower bound for any data structure maintaining a dynamic collection of non-persistent strings under operations `make_string`(w), `concat`(w_1, w_2) and `split`(w, k) and answering equality queries `equal`(w_1, w_2). In such a setting, if we are maintaining poly(n) strings of length at most poly(n), the amortized complexity of at least one operation among `concat`, `split`, `equal` is $\Omega(\log n)$. The lower bound applies to any Monte Carlo structure returning correct answers w.h.p. and remains valid for binary alphabet if `equal` supports comparing strings of length $\Theta(\log n)$ only (otherwise, comparing strings of length one is enough).

The lower bound is based on the following result of Pătraşcu and Demaine [34]. We want to maintain $\sqrt{n}$ permutations $\pi_1, \pi_2, \ldots, \pi_{\sqrt{n}} \in S_{\sqrt{n}}$ on $\sqrt{n}$ elements. An update `update`(i, π) sets $\pi_i := \pi$ and a query `verify`(i, π) checks whether $\pi = p_i$, where $p_i := \pi_i \circ \pi_{i-1} \circ \ldots \circ \pi_1$. We consider sequences of $\sqrt{n}$ pairs of operations, each pair being `update`(i, π) with $i \in [1, \sqrt{n}]$ and $\pi \in S_{\sqrt{n}}$ chosen uniformly at random and `verify`(i, π) with $i \in [1, \sqrt{n}]$ chosen uniformly at random and $\pi = \pi_i \circ \pi_{i-1} \circ \ldots \circ \pi_1$. That is, `verify` is asked to prove a tautology, and it must probe enough cells to certify it. Then, if the word size is $\Theta(\log n)$, it can be proved that the expected total number of cell probes (and hence also the total time complexity) must be $\Omega(n \log n)$ through an entropy-based argument (even for Monte Carlo algorithms). The essence of this argument is that if we consider ℓ indices $q_1 < q_2 < \ldots q_\ell$ and execute at least one `update`(i, π) with $i \in [q_{j-1} + 1, q_j]$, for every $j = 2, 3, \ldots, \ell$, then all partial sums p_{q_j} are independent random variables uniformly distributed in $S_{\sqrt{n}}$.

Pătraşcu and Demaine [34] use the above result to show a lower bound for dynamic connectivity by considering an intermediate problem. For every $i = 0, 1, \ldots, \sqrt{n}$ we create a layer of $\sqrt{n}$ nodes $v_{i,1}, v_{i,2}, \ldots, v_{i,\sqrt{n}}$. Then, for every $i = 1, 2, \ldots, \sqrt{n}$ and $j = 1, 2, \ldots, \sqrt{n}$ we connect $v_{i-1,j}$ with $v_{i,\pi_i(j)}$ to obtain a collection of disjoint paths. Now, `verify`(i, π) can be implemented by asking if $v_{0,j}$ is connected to $v_{i,\pi(j)}$ for every $j = 1, 2, \ldots, \sqrt{n}$. The implementation of `update`(i, π) first removes all edges between nodes from the $(i-1)$-th and i-th layer and then connects them according to π. To show a lower bound for maintaining a dynamic collection of strings, we represent every path with a string. The j-th string s_j describes the path connecting $v_{0,j}$ and $v_{\sqrt{n},p_{\sqrt{n}}(j)}$: $s_j = p_0(j)p_1(j)\ldots p_{\sqrt{n}}(j)$, where every $p_i(j)$ is a separate character. Additionally, for every j we prepare a one-character string c_j such that $c_j \neq c_k$ for $j \neq k$.

To check if $v_{0,j}$ and $v_{i,\pi(j)}$ are connected, we split s_j into three parts $s'_j s''_j s'''_j$ such that $|s'_j| = i$, $|s''_j| = 1$ and $|s'''_j| = \sqrt{n} - i$, compare s''_j with the prepared string $c_{\pi(j)}$, and concatenate s'_j, s''_j and s'''_j to recover s_j.

The difficulty lies in implementing `update`(i, π), i.e., replacing all edges between the $(i-1)$-th and i-th layer. We split every s_j into two parts $s'_j s''_j$ such that $|s'_j| = i$ and $|s''_j| = \sqrt{n} - i + 1$. Then we would like to create the new s_j by concatenating s'_j and $s''_{j'}$ such that $s'_j[i] = k$ and $s''_{j'}[i+1] = \pi(k)$, but we are not able to efficiently find the corresponding j' for every j by only comparing strings. We change the meaning of `update`(i, π): instead of $\pi_i := \pi$ the algorithm is required to set $\pi_i := p_i \circ \pi \circ p_{i-1}^{-1}$. This can be implemented by simply concatenating s'_j and $s''_{\pi(j)}$ to form the new s_j, for every $j = 1, 2, \ldots, \sqrt{n}$. The expected total number of cell probes must still be $\Omega(n \log n)$ by the same argument: if we consider ℓ indices $q_1 < q_2 < \ldots q_\ell$ and call `update`(i, π) with $i \in [q_{j-1} + 1, q_j]$ for every $j = 2, 3, \ldots, \ell$, then all p_{q_j} are independent random variables uniformly distributed in $S_{\sqrt{n}}$. Thus, we obtain the following theorem.

Theorem 7.1. *For any data structure maintaining a dynamic collection of* poly(n) *binary non-persistent strings of length* poly(n) *subject to operations* `make_string`*,* `concat`*,* `split`*, and* `equal`*, correct w.h.p., the amortized complexity of* `concat`*,* `split`*, or* `equal` *is* $\Omega(\log n)$*.*

References

[1] S. Alstrup, G. S. Brodal, and T. Rauhe. Pattern matching in dynamic texts. In *SODA 2000*, pages 819–828. ACM/SIAM.

[2] S. Alstrup and J. Holm. Improved algorithms for finding level ancestors in dynamic trees. In *ICALP 2000*, pages 73–84.

[3] A. Amir, G. Franceschini, R. Grossi, T. Kopelowitz, M. Lewenstein, and N. Lewenstein. Managing unbounded-length keys in comparison-driven data structures with applications to online indexing. *SIAM J. Comput.*, 43(4):1396–1416, 2014.

[4] D. Breslauer and G. F. Italiano. Near real-time suffix tree construction via the fringe marked ancestor problem. *J. Discrete Algorithms*, 18:32–48, 2013.

[5] R. Cole and R. Hariharan. Dynamic LCA queries on trees. *SIAM Journal on Computing*, 34(4):894–923, Apr. 2005.

[6] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, July 1986.

[7] V. Diekert, A. Jeż, and M. Kufleitner. Solutions of word equations over partially commutative structures. In *ICALP 2015*, volume 55 of *LIPIcs*, pages 127:1–127:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

[8] V. Diekert, A. Jeż, and W. Plandowski. Finding all solutions of equations in free groups and monoids with involution. *Inf. Comput.*, 251:263–286, 2016.

[9] P. Ferragina. Dynamic text indexing under string updates. *Journal of Algorithms*, 22(2):296–328, Feb. 1997.

[10] P. Ferragina and R. Grossi. Optimal on-line search and sublinear time update in string matching. *SIAM Journal on Computing*, 27(3):713–736, 1998.

[11] J. Fischer and P. Gawrychowski. Alphabet-dependent string searching with wexponential search trees. In *CPM 2015*, volume 9133 of *LNCS*, pages 160–171. Springer, 2015.

[12] P. Gawrychowski and A. Jeż. LZ77 factorisation of trees. In *FSTTCS 2016*, volume 65 of *LIPIcs*, pages 35:1–35:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

[13] P. Gawrychowski, A. Karczmarz, T. Kociumaka, J. Lacki, and P. Sankowski. Optimal dynamic strings. *CoRR*, abs/1511.02612, 2015.

[14] M. Gu, M. Farach, and R. Beigel. An efficient algorithm for dynamic text indexing. In *SODA 1994*, pages 697–704, Philadelphia, PA, USA. ACM/SIAM.

[15] L. Gąsieniec, R. M. Kolpakov, I. Potapov, and P. Sant. Real-time traversal in grammar-based compressed files. In *DCC 2005*, page 458. IEEE Computer Society.

[16] A. Jeż. Context unification is in PSPACE. In *ICALP 2014*, volume 8573 of *LNCS*, pages 244–255. Springer.

[17] A. Jeż. The complexity of compressed membership problems for finite automata. *Theory Comput. Syst.*, 55(4):685–718, 2014.

[18] A. Jeż. Approximation of grammar-based compression via recompression. *Theor. Comput. Sci.*, 592:115–134, 2015.

[19] A. Jeż. Faster fully compressed pattern matching by recompression. *ACM Transactions on Algorithms*, 11(3):20:1–20:43, 2015.

[20] A. Jeż. One-variable word equations in linear time. *Algorithmica*, 74(1):1–48, 2016.

[21] A. Jeż. A really simple approximation of smallest grammar. *Theor. Comput. Sci.*, 616:141–150, 2016.

[22] A. Jeż. Recompression: A simple and powerful technique for word equations. *J. ACM*, 63(1):4:1–4:51, 2016.

[23] A. Jeż and M. Lohrey. Approximation of smallest linear tree grammar. *Inf. Comput.*, 251:215–251, 2016.

[24] T. Kopelowitz. On-line indexing for general alphabets via predecessor queries on subsets of an ordered list. In *FOCS 2012*, pages 283–292. IEEE Computer Society.

[25] G. Kucherov and Y. Nekrich. Full-fledged real-time indexing for constant size alphabets. *Algorithmica*, Aug 2016.

[26] M. Lohrey. Equality testing of compressed strings. In *WORDS 2015*, volume 9304 of *LNCS*, pages 14–26. Springer.

[27] M. Lohrey. Algorithmics on SLP-compressed strings: A survey. *Groups Complexity Cryptology*, 4(2):241–299, 2012.

[28] M. Lohrey, S. Maneth, and C. P. Reh. Traversing grammar-compressed trees with constant delay. In *DCC*, pages 546–555. IEEE, 2016.

[29] K. Mehlhorn, R. Sundar, and C. Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997.

[30] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *FOCS 1985*, pages 478–489. IEEE Computer Society.

[31] T. Nishimoto, T. I, S. Inenaga, H. Bannai, and M. Takeda. Dynamic index and LZ factorization in compressed space. In *Prague Stringology Conference 2016*, pages 158–170. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague.

[32] T. Nishimoto, T. I, S. Inenaga, H. Bannai, and M. Takeda. Fully dynamic data structure for LCE queries in compressed space. In *MFCS 2016*, volume 58 of *LIPIcs*, pages 72:1–72:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

[33] C. Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[34] M. Patrascu and E. D. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM Journal on Computing*, 35(4):932–963, 2006.

[35] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *POPL 1989*, pages 315–328. ACM.

[36] S. C. Sahinalp and U. Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm. In *FOCS 1996*, pages 320–328. IEEE Computer Society.

[37] S. C. Sahinalp and U. Vishkin. Data compression using locally consistent parsing. Technical report, University of Maryland, Department of Computer Science, 1995.

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

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

[40] U. Vishkin. Randomized speed-ups in parallel computation. In *STOC 1984*, pages 230–239, New York, NY, USA. ACM.