LZ-End Parsing in Linear Time

Dominik Kempa[1] **and Dmitry Kosolobov**[2]

1 **University of Helsinki, Helsinki, Finland**
`dominik.kempa@cs.helsinki.fi`
2 **University of Helsinki, Helsinki, Finland**
`dkosolobov@mail.ru`

Abstract

We present a deterministic algorithm that constructs in linear time and space the LZ-End parsing (a variation of LZ77) of a given string over an integer polynomially bounded alphabet.

1998 ACM Subject Classification E.4 Data compaction and compression

Keywords and phrases LZ-End, LZ77, construction algorithm, linear time

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.53

1 Introduction

Lempel–Ziv (LZ77) parsing [34] has been a cornerstone of data compression for the last 40 years. It lies at the heart of many common compressors such as `gzip`, `7-zip`, `rar`, and `lz4`. More recently LZ77 has crossed-over into the field of *compressed indexing of highly repetitive data* that aims to store repetitive databases (such as repositories of version control systems, Wikipedia databases [31], collections of genomes, logs, Web crawls [8], etc.) in small space while supporting fast substring retrieval and pattern matching queries [4, 7, 13, 14, 15, 23, 27]. For this kind of data, in practice, LZ77-based techniques are more efficient in terms of compression than the techniques used in the standard compressed indexes such as FM-index and compressed suffix array (see [24, 25]); moreover, often the space overhead of these standard indexes hidden in the $o(n)$ term, where n is the length of the uncompressed text, turns out to be too large for highly repetitive data [4].

One of the first and most successful indexes for highly repetitive data was proposed by Kreft and Navarro [24]. In its simplest form LZ77 greedily splits the input text into substrings (called *phrases*) such that each phrase is a first occurrence of a single letter or the longest substring that has an earlier occurrence. The index in [24] is built upon a small modification of LZ77 parsing called *LZ-End* (introduced in [22]) which assumes that the end of an earlier occurrence of each phrase aligns with the end of some previous phrase. This enables much faster retrieval of substrings of the compressed text without decompression.

While basic LZ77 parsing is solved optimally in many models [2, 9, 17, 18, 21, 26, 30], the construction of LZ-End remains a problem. Kreft and Navarro [24] presented an algorithm that constructs the LZ-End parsing of a string of length n in $O(n\ell(\log\sigma + \log\log n))$ time and $O(n)$ space, where ℓ is the length of the longest phrase in the parsing and σ is the alphabet size. They also presented a more space efficient version that works in $O(n\ell\log^{1+\epsilon} n)$ time and uses $O(n\log\sigma)$ *bits* of space, where ϵ is an arbitrary positive constant. This construction algorithm provides unsatisfactory time guarantees: it is quadratic in the worst case.

In [19] we described an algorithm that builds the LZ-End parsing of a read-only string of length n in $O(n\log\ell)$ *expected* time and $O(z+\ell)$ space, where z is the number of phrases and ℓ is the length of the longest phrase. In this paper we present an optimal-time deterministic algorithm constructing the LZ-End parsing. We assume that the input string (of length n) is drawn from a polynomially bounded integer alphabet $\{0, 1, \dots, n^{O(1)}\}$ and the computational model is the standard word RAM with $\Theta(\log n)$-bit machine words.

▶ **Theorem 1.** *The LZ-End parsing of a string of length n over the alphabet $\{0, 1, \dots, n^{O(1)}\}$ can be computed in $O(n)$ time and space.*

The paper is organized as follows. In Section 2 we describe an algorithm that constructs the LZ-End parsing in $O(n \log\log n)$ time and linear space; we believe that this intermediate result is especially interesting for practice. In Section 3, we obtain an $O(n \log^2 n)$-time algorithm based on a completely different approach. Finally, we combine the two developed techniques in Section 4 and thus obtain a linear algorithm.

Preliminaries. Let s be a string of length $|s| = n$. We write $s[i]$ for the ith letter of s and $s[i..j]$ for $s[i]s[i{+}1]\cdots s[j]$. The *reversal of* s is the string $\overleftarrow{s} = s[n]\cdots s[2]s[1]$. A string u is a *substring* of s if $u = s[i..j]$ for some i and j; the pair (i, j) is not necessarily unique and we say that i specifies an *occurrence* of u in s. A substring $s[1..j]$ (resp., $s[i..n]$) is a *prefix* (resp. *suffix*) of s. For any i, j, the set $\{k \in \mathbb{Z} \colon i \le k \le j\}$ (possibly empty) is denoted by $[i..j]$. Our notation for arrays is similar: e.g., $a[i..j]$ denotes an array indexed by the numbers $[i..j]$.

Hereafter, s denotes the input string of length n over the integer alphabet $\{0, 1, \dots, n^{O(1)}\}$. We extensively use a number of classical arrays built on the reversal $\overleftarrow{s}$ (the definitions slightly differ from the standard ones to avoid excessive mappings between the positions of s and $\overleftarrow{s}$): the *suffix array* $\mathsf{SA}[1..n]$ such that $\overleftarrow{s[1..\mathsf{SA}[1]]} < \overleftarrow{s[1..\mathsf{SA}[2]]} < \cdots < \overleftarrow{s[1..\mathsf{SA}[n]]}$ (lexicographically), the *inverse suffix array* $\mathsf{ISA}[1..n]$ such that $\mathsf{SA}[\mathsf{ISA}[i]] = i$ for $i \in [1..n]$, and the *longest common prefix (LCP) array* $\mathsf{LCP}[1..n{-}1]$ such that, for $i \in [1..n{-}1]$, $\mathsf{LCP}[i]$ is equal to the length of the longest common prefix of $\overleftarrow{s[1..\mathsf{SA}[i]]}$ and $\overleftarrow{s[1..\mathsf{SA}[i{+}1]]}$. We equip the array LCP with the *range minimum query (RMQ)* data structure [10] that, for any $i, j \in [1..n]$ such that $\mathsf{ISA}[i] < \mathsf{ISA}[j]$, allows us to compute in $O(1)$ time the value $\min\{\mathsf{LCP}[k] \colon \mathsf{ISA}[i] \le k < \mathsf{ISA}[j]\}$, which is equal to the length of the longest common suffix of $s[1..i]$ and $s[1..j]$. For brevity, this combination of LCP and RMQ is called the *LCP structure*. It is well known that all these structures can be built in $O(n)$ time (e.g., see [6]).

The *LZ-End parsing* [22, 23, 24] of a string s is a decomposition $s = f_1 f_2 \cdots f_z$ constructed by the following greedy process: if we have already processed a prefix $s[1..k] = f_1 f_2 \cdots f_{i-1}$, then $f_i[1..|f_i|{-}1]$ is the longest prefix of $s[k{+}1..|s|{-}1]$ that is a suffix of a string $f_1 f_2 \cdots f_j$ for some $j < i$; the substrings f_i are called *phrases*. For instance, the string *ababaaaaaac* has the LZ-End parsing *a.b.aba.aa.aaac*.

2 First Suboptimal Algorithm

Our first approach is based on two combinatorial properties of the LZ-End parsing that were observed in [19]. First, the definition of the LZ-End parsing easily implies the following lemma suggesting a way how to perform the construction of the LZ-End parsing incrementally.

▶ **Lemma 2.** *Let $f_1 f_2 \cdots f_z$ be the LZ-End parsing of a string s. If i is the maximal integer such that the string $f_{z-i} f_{z-i+1} \cdots f_z$ is a suffix of a string $f_1 f_2 \cdots f_j$ for $j < z - i$, then, for any letter a, the LZ-End parsing of the string sa is $f'_1 f'_2 \cdots f'_{z'}$, where $z' = z - i$, $f'_1 = f_1, f'_2 = f_2, \dots, f'_{z'-1} = f_{z'-1}$, and $f'_{z'} = f_{z-i} f_{z-i+1} \cdots f_z a$.*

Secondly, it turns out that the number of phrases that might "unite" into a new phrase when a letter has been appended (as in Lemma 2) is severely restricted.

▶ **Lemma 3** (see [19]). *If $f_1f_2\cdots f_z$ is the LZ-End parsing of a string s, then, for any letter a, the last phrase in the LZ-End parsing of the string sa is 1) $f_{z-1}f_za$ or 2) f_za or 3) a.*

The algorithm presented in this section builds the LZ-End parsing incrementally. When a prefix $s[1..k]$ is processed, we have the LZ-End parsing $f_1f_2\cdots f_z$ of $s[1..k]$ and we are to construct the parsing for the string $s[1..k{+}1]$. By Lemma 3, if $f_{z-1}f_z$ (resp., f_z) is a suffix of $f_1f_2\cdots f_j$ for some $j < z-1$ (resp., $j < z$), then the last phrase in the parsing of $s[1..k{+}1]$ is $f_{z-1}f_zs[k{+}1]$ (resp., $f_zs[k{+}1]$); otherwise, the last phrase is $s[k{+}1]$.

To process the cases of Lemma 3 efficiently, we maintain a bit array $M[1..n]$ that marks those prefixes in the lexicographically sorted set of all reversed prefixes of s that end at phrase boundaries: for $i \in [1..n]$, $M[i] = 1$ iff $s[1..\mathsf{SA}[i]] = f_1f_2\cdots f_j$ for some $j \in [1..z]$ in the LZ-End parsing $f_1f_2\cdots f_z$ of the current prefix $s[1..k]$. We equip M with the *van Emde Boas data structure* [33] that allows us to compute, for any given i, the maximal $j < i$ (resp., the minimal $j' > i$) (if any) such that $M[j] = 1$ (resp., $M[j'] = 1$); we use a dynamic version of this data structure that occupies $O(n)$ space and supports queries on M and modifications of the form $M[i] \leftarrow 1$ or $M[i] \leftarrow 0$ in $O(\log\log n)$ deterministic time (e.g., see [5]).

Let us describe how to check whether f_z has an *earlier* occurrence in $s[1..k] = f_1f_2\cdots f_z$ that ends at a phrase boundary. We first find in $O(\log\log n)$ time the maximal $j < \mathsf{ISA}[k]$ and the minimal $j' > \mathsf{ISA}[k]$ such that $M[j] = 1$ and $M[j'] = 1$. Suppose such j and j' exist (the case when either $M[1..\mathsf{ISA}[k]{-}1]$ or $M[\mathsf{ISA}[k]{+}1..n]$ consists of all zero is similar but simpler). Using the LCP structure, we compute in $O(1)$ time the length t (resp., t') of the longest common suffix of $s[1..k]$ and $s[1..\mathsf{SA}[j]]$ (resp., $s[1..k]$ and $s[1..\mathsf{SA}[j']]$). It is straightforward that f_z has an earlier occurrence in $s[1..k]$ ending at a phrase boundary iff $\max\{t, t'\} \geq |f_z|$.

Analogously, to check whether $f_{z-1}f_z$ has an earlier occurrence in $s[1..k]$ ending at a phrase boundary different from the boundary $|f_1f_2\cdots f_{z-1}|$, we temporarily unmark the bit $M[\mathsf{ISA}[k - |f_z|]]$ modifying the van Emde Boas data structure accordingly, then obtain the numbers t and t' in the same way as above, and restore $M[\mathsf{ISA}[k - |f_z|]]$ with the van Emde Boas data structure, all in $O(\log\log n)$ time; $f_{z-1}f_z$ has the required earlier occurrence iff $\max\{t, t'\} \geq |f_{z-1}f_z|$. Finally, by Lemmas 2 and 3, we obtain the LZ-End parsing of the prefix $s[1..k{+}1]$ by removing, based on the above computations, zero, one, or two last phrases from the list of all phrases and adding a new last phrase. The array M and the van Emde Boas data structure are modified accordingly. Thus, we have proved the following lemma.

▶ **Lemma 4.** *The LZ-End parsing of a string of length n over the alphabet $\{0, 1, \ldots, n^{O(1)}\}$ can be computed in $O(n \log\log n)$ time and $O(n)$ space.*

This algorithm provides good time guarantees (unlike the algorithms of Kreft and Navarro [24]) and seems to be of practical interest.

It is easy to see that the van Emde Boas data structure that is required to search predecessors and successors in the dynamic bit array M is the bottleneck of the described algorithm. It is known that for the insertion-only bit arrays there is an analogous data structure (the split-find data structure discussed below) that works in $O(n)$ overall time. Then, it is natural to ask whether the array M really requires a lot of deletions in the worst case or, based on the LZ77 intuition[1], only a few phrases in the LZ-End parsing of the current prefix might be removed in the future. The following example shows that a significant

[1] A similar incremental construction procedure for LZ77 would, at each step, append a letter to the end of the current string and then modify only the last phrase of the currently built parsing.

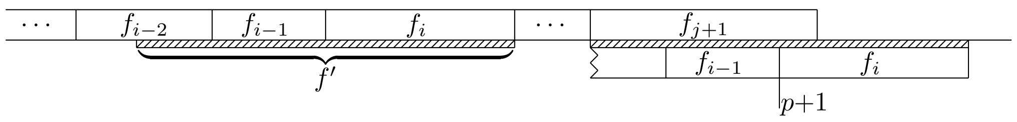

Figure 1 Case (2) in Lemma 6; f_i occurs at position $p{+}1$ located inside the phrase f_{j+1}.

amount of phrases from the LZ-End parsing of the current prefix can be removed in the final parsing and, therefore, the described approach strongly relies on the dynamic predecessor structure, which is known to require $\omega(1)$ query time [1, 28].

▶ **Example 5.** Choose an integer $k > 0$. Define $s_k = a_k$ and $s_i = a_i b_{i+1} s_{i+1}$ for $i = k{-}1, \ldots, 2, 1$, where a_i and b_i are distinct letters. Define $t_i = c_k c_{k-1} \cdots c_i$, where c_i are letters different from a_i and b_i. Our example is the string $s = s_k t_k s_{k-1} t_{k-1} \cdots s_2 t_2 s_1 t_1 b_2 s_2$ (notice the ending $b_2 s_2$). The string s is depicted below with separators "|", which are not real letters, at the end of each phrase of the LZ-End parsing of s (for readability, s is split into lines corresponding to the substrings $s_i t_i$ and the lines are aligned):

$$
\begin{array}{r}
a_k|c_k| \\
a_{k-1}|b_k|a_k c_k c_{k-1}| \\
\cdots\cdots\cdots\cdots \\
a_2|b_3|a_3 b_4 a_4 \cdots b_{k-2} a_{k-2} b_{k-1} a_{k-1} b_k a_k c_k c_{k-1} c_{k-2} \cdots c_2| \\
a_1|b_2|a_2 b_3 a_3 b_4 a_4 \cdots b_{k-2} a_{k-2} b_{k-1} a_{k-1} b_k a_k c_k c_{k-1} c_{k-2} \cdots c_2 c_1| \\
b_2 a_2|b_3 a_3|b_4 a_4| \cdots b_{k-2} a_{k-2}|b_{k-1} a_{k-1}|b_k a_k|.
\end{array}
$$

The parsing of any substring $s_i t_i$, for $i \in [2..k]$, consists of three phrases: two phrases corresponding to the letters a_i and b_{i+1} that did not occur before and the phrase $s_{i+1} t_{i+1} c_i$, where $s_{i+1} t_{i+1}$ is the previous line and c_i is a letter that did not occur before. Now consider the parsing of the last line $b_2 s_2$. For $i < k$, there is only one occurrence of a_i before the last line that is succeeded by a separator "|" and this occurrence is preceded by c_{i+1}. Analogously, for $i \le k$, the only earlier occurrence of b_i succeeded by "|" is preceded by $c_i a_{i-1}$. This observation easily implies that the parsing of the last line consists of $k{-}1$ phrases $b_i a_i$.

Now consider the string $st_0 = sc_k c_{k-1} \cdots c_0$. The last phrase of the LZ-End parsing of st_0 is $b_2 s_2 t_0$ because $b_2 s_2 t_1$ is a suffix of the substring $s_1 t_1$ (kth line). Thus, this last phrase "absorbs" $k - 1$ last phrases of the parsing of s. It remains to notice that the length of s is $\Theta(k^2)$ and the number of phrases in the parsing of s is $\Theta(k)$.

3 Second Suboptimal Algorithm

Our second algorithm follows the definition of the LZ-End parsing constructing phrases greedily one by one from left to right. The algorithm itself is inefficient but we will show in Section 4 that its techniques can be combined with the incremental solution of Section 2 in order to obtain a linear algorithm.

Suppose that $f_1, f_2, \ldots, f_j$ are the first j phrases of the LZ-End parsing of s and f is a candidate for a new phrase, i.e., $f_1 f_2 \cdots f_j f$ is a prefix of s and $f[1..|f|{-}1]$ is a suffix of $f_1 f_2 \cdots f_k$ for $k \in [1..j]$. Our method "grows" f relying on the following lemma (see Fig. 1).

▶ **Lemma 6.** *Suppose that $f_1 f_2 \cdots f_z$ is the LZ-End parsing of a prefix of s. If, for $j \in [1..z{-}1]$, the phrase f_{j+1} is not present in the LZ-End parsing of the whole string s, then there exists $i \in [1..j]$ such that either (1) f_{j+1} is a suffix of the phrase f_i or (2) $f_1 f_2 \cdots f_i$ has a suffix f' such that $f_1 f_2 \cdots f_j f'$ is a prefix of s, f_{j+1} is a prefix of f', and $0 < |f'| - |f_i| < |f_{j+1}|$.*

Proof. Since f_{j+1} is not a phrase of the LZ-End parsing of s, it follows from Lemma 2 that there exists $j' \in [1..j]$ such that the first $j'+1$ phrases of the LZ-End parsing of s are $f_1, f_2, \ldots, f_{j'}, f$, where $f[1..|f|{-}1]$ contains f_{j+1} as a substring. By definition, there exists $i' \in [1..j']$ such that $f[1..|f|{-}1]$ is a suffix of $f_1 f_2 \cdots f_{i'}$. Suppose that the corresponding copy of f_{j+1} in $f_1 f_2 \cdots f_{i'}$ occurs at position q. Then, by construction, the suffix of $f_1 f_2 \cdots f_{i'}$ starting at position q occurs at position $|f_1 f_2 \cdots f_j|{+}1$. Hence, if $s[q..q{+}|f_{j+1}|{-}1] = f_{j+1}$ is a suffix of a phrase or is "intersected" by a phrase boundary, we easily obtain, respectively, (1) or (2). Otherwise, there is a phrase $\hat{f}$ such that $s[q..q{+}|f_{j+1}|{-}1] = f_{j+1}$ is a substring of $\hat{f}[1..|\hat{f}|{-}1]$ and the suffix of $\hat{f}$ starting at position q occurs at position $|f_1 f_2 \cdots f_j|{+}1$. In other words, this situation is analogous to the situation with f and we can analogously consider the "source" of $\hat{f}[1..|\hat{f}|{-}1]$ and the corresponding copy of f_{j+1} in this source. Then we repeat the analysis. Since this recursive procedure moves us to the left every time, it cannot continue forever and we will eventually find that either (1) or (2) holds. ◀

To extend f according to the case (1) of Lemma 6, we store all constructed phrases $f_1, f_2, \ldots, f_j$ in the lexicographically sorted order of their reversals, i.e., we store a permutation $i_1, i_2, \ldots, i_j$ of $[1..j]$ such that $\overleftarrow{f}_{i_1} \le \overleftarrow{f}_{i_2} \le \cdots \le \overleftarrow{f}_{i_j}$. By the binary search in this sorted set, we find in $O(\log n)$ time, using the LCP structure, whether f is a suffix of f_i for some $i \in [1..j]$. This part is similar to the incremental approach of Section 2 (but less efficient).

To extend f according to the case (2) of Lemma 6, we process every position p of the considered occurrence of f in s and try to find a phrase f_i such that, as it is depicted in Figure 1 (assuming $f_{j+1} = f$), $i \in [1..j]$, f_i occurs at position $p{+}1$ embracing the last position of f (i.e., $p + |f_i| \ge |f_1 f_2 \cdots f_j f|$), and the prefix of f ending at position p is a suffix of $f_1 f_2 \cdots f_{i-1}$ (the details follow). Let us describe the data structures required to find such f_i.

Auxiliary data structures. First, we build the *suffix tree* of the string s (see the definition in, e.g., [6]); note that, unlike the suffix array SA that was built for the reversal $\overleftarrow{s}$, the suffix tree is built for the string s itself and, thus, contains the suffixes $s[1..n], s[2..n], \ldots, s[n..n]$. For simplicity, assume that $s[n]$ is a special letter that does not occur in $s[1..n{-}1]$ and, hence, the suffixes of s are in the one-to-one correspondence with the leaves of the tree. We build an array of pointers mapping suffixes of s to the corresponding leaves. It is well known (e.g., see [6]) that the suffix tree of s with this array can be constructed in $O(n)$ time.

Recall that the suffix tree has *explicit* and *implicit* vertices. The *string depth* of an (explicit or implicit) vertex is the length of the string written on the path connecting the root and the vertex. We augment the suffix tree with the following dynamic data structure.

▶ **Lemma 7.** *In $O(n)$ time one can build on the suffix tree of s a data structure supporting the following operations:*

1. *for a given number $w \in [1..n]$ and (explicit or implicit) vertex v, mark v and assign the weight w to v, all in $O(\log n)$ time;*
2. *for given numbers i, j, d and a leaf, find a marked vertex v that is an ancestor of this leaf, has weight $w \in [i..j]$, and has string depth at least d, all in $O(\log^2 n)$ time.*

Proof. The *heavy path decomposition* [32] is a decomposition of all vertices of the suffix tree into disjoint paths (called *heavy paths*), each of which descends from a vertex to a leaf, so that all ancestors of any given leaf belong to at most $\log n$ distinct heavy paths. It is shown in [32] that the heavy path decomposition can be constructed in $O(n)$ time.

We equip each heavy path with an (initially empty) dynamic 2-dimensional orthogonal range reporting data structure of [3]. To mark a given vertex v and assign a weight $w \in [1..n]$ to it, we simply insert in $O(\log n)$ time [3] the pair (d, w), where d is the string depth of v, into the range reporting data structure corresponding to the heavy path containing v. In order to answer a query for given numbers i, j, d and a leaf, we consecutively process each of $O(\log n)$ ancestral heavy paths of this leaf starting from the deepest one: for each path, we perform in $O(\log n)$ time [3] the range reporting query $[d..d'] \times [i..j]$, where d' is equal to the string depth of the vertex having the "head" of the previously processed heavy path as a child and $d' = n$ for the path containing the leaf. It is easy to see that one of these range reporting queries must find (if any) the required marked ancestor whose weight is in the given range $[i..j]$ and whose string depth is at least d. Auxiliary structures organizing all fast navigation on the heavy paths can be easily constructed in $O(n)$ time. ◀

Also we utilize the following data structure, which can be viewed as a simplified version of the weighted ancestor data structures from [16, 20].

▶ **Lemma 8** (see [16, 20]). *In $O(n)$ time one can build on the suffix tree of s a data structure that, for a given leaf and a number d, allows us to find in $O(\log n)$ time the ancestor (explicit or implicit) of the leaf with the string depth d.*

To find an (explicit or implicit) vertex corresponding to a substring $s[i..j]$, one can perform the query of Lemma 8 on $d = j - i + 1$ and on the leaf corresponding to $s[i..n]$.

Algorithm. At the beginning, the algorithm builds in $O(n)$ time the LCP structure and the suffix tree of s equipped with the data structures of Lemmas 7 and 8. We maintain the following invariant: if the first j phrases $f_1, \ldots, f_j$ of the LZ-End parsing of s are already constructed, then, for each $i \in [1..j]$, the vertex (implicit or explicit) of the suffix tree of s corresponding to the string f_i is marked and has weight $\mathsf{ISA}[|f_1 \cdots f_{i-1}|]$; we also store in a dynamic balanced tree a permutation $i_1, \ldots, i_j$ of the set $[1..j]$ such that $\overleftarrow{f_{i_1}} \le \cdots \le \overleftarrow{f_{i_j}}$.

Suppose that we have already constructed j phrases $f_1, f_2, \ldots, f_j$ and f is a candidate for the new phrase f_{j+1}, i.e., $f[1..|f|-1]$ is a suffix of $f_1 f_2 \cdots f_k$ for some $k \in [1..j]$. First, using the LCP structure and the balanced tree containing $i_1, \ldots, i_j$, we perform in $O(\log n)$ time the binary search in the sorted set $\overleftarrow{f_{i_1}}, \ldots, \overleftarrow{f_{i_j}}$ and find whether f is a suffix of f_i for some $i \in [1..j]$. If such f_i exists, then we "grow" f by one letter according to the case (1) of Lemma 6 and the string fa, where $a = s[|f_1 \cdots f_j f|+1]$, becomes a new candidate for f_{j+1}. Otherwise, we consecutively process from left to right each position p in the considered occurrence of f (i.e., $|f_1 f_2 \cdots f_j| < p < |f_1 f_2 \cdots f_j f|$) and check whether there is a phrase f_i, for $i \in [1..j]$, such that the prefix u of f ending at p (i.e., $u = s[|f_1 f_2 \cdots f_j|+1..p]$) is a suffix of $f_1 f_2 \cdots f_{i-1}$ and f_i occurs at position $p{+}1$ embracing the position $|f_1 f_2 \cdots f_j f|$ (i.e., $p + |f_i| \ge |f_1 f_2 \cdots f_j f|$); the procedure finding such f_i for a given p works in $O(\log^2 n)$ time and is described below in Lemma 9. Once such f_i is found for a position p, we "grow" f according to the case (2) of Lemma 6 so that the string $s[|f_1 f_2 \cdots f_j|+1..p+|f_i|+1]$, which contains f as a proper prefix, becomes a new candidate for f_{j+1}. Obviously, if we processed a position p in this way and could not extend f, then there is no reason to consider p in the future. Hence, the whole left to right processing of the positions of f can start not from the first position $|f_1 f_2 \cdots f_j|+1$ of f but from the last processed position of f (if any).

It follows from Lemma 6 that if we could not grow f neither by the processing of all positions p inside f nor by the processing of the case (1) of Lemma 6 described above, then f is the new phrase f_{j+1}. In this case, to maintain the invariant, we find the (explicit or implicit) vertex corresponding to the string $f = f_{j+1}$, mark this vertex, and assign the weight $\mathsf{ISA}[|f_1 f_2 \cdots f_j|]$ to it; all this is done in $O(\log n)$ time using the data structures from Lemmas 7 and 8. Further, using the binary search and the LCP structure, we insert the string $\overleftarrow{f}_{j+1}$ in an appropriate place of the sorted set $\overleftarrow{f_{i_1}}, \ldots, \overleftarrow{f_{i_j}}$ and modify the balanced tree storing $i_1, \ldots, i_j$ accordingly, all in $O(\log n)$ time. Finally, the string $s[|f_1 \cdots f_{j+1}|{+}1]$ becomes a candidate for the next phrase f_{j+2} and we continue the construction.

Once we have performed in $O(\log^2 n)$ time the procedure finding an "extending" phrase f_i for a given position p inside f (see Lemma 9 below), we either grow the current candidate f by at least one letter or we do not grow f and this was the last processing of this position. By this observation, the overall running time of the algorithm is $O(n \log^2 n)$. The procedure itself is described in the following lemma, assuming that $h-1 = j$, $f_h = f$, and $S = \{f_1, f_2, \ldots, f_j\}$; the lemma is formulated in a more general form that will be useful below in Section 4.

▶ **Lemma 9.** *Let $f_1 f_2 \cdots f_h$ be the LZ-End parsing of a prefix of s. Suppose that, for each f_i from a subset S of phrases, the vertex of the suffix tree of s corresponding to f_i is marked and has weight $\mathsf{ISA}[|f_1 f_2 \cdots f_{i-1}|]$. Then, for any position p such that $|f_1 f_2 \cdots f_{h-1}| < p < |f_1 f_2 \cdots f_h|$, one can find in $O(\log^2 n)$ time $f_i \in S$ (if any) such that $p + |f_i| \geq |f_1 f_2 \cdots f_h|$, f_i occurs at position $p{+}1$, and $s[|f_1 f_2 \cdots f_{h-1}|{+}1..p]$ is a suffix of $f_1 f_2 \cdots f_{i-1}$.*

Proof. Suppose that the required phrase $f_i \in S$ indeed exists. By assumption, the vertex v of the suffix tree that corresponds to the string f_i is marked and has weight $\mathsf{ISA}[|f_1 f_2 \cdots f_{i-1}|]$. The vertex v is an ancestor of the leaf corresponding to $s[p{+}1..n]$. Denote $u = s[|f_1 f_2 \cdots f_{h-1}|{+}1..p]$. Let $[\ell_u..r_u]$ be the maximal subrange of the range $[1..n]$ such that, for each $d \in [\ell_u..r_u]$, the string $s[1..\mathsf{SA}[d]]$ has a suffix u; the range $[\ell_u..r_u]$ can be calculated by the binary search in $O(\log n)$ time using the LCP structure. Since u is a suffix of $f_1 f_2 \cdots f_{i-1}$, the weight $\mathsf{ISA}[|f_1 f_2 \cdots f_{i-1}|]$ of the vertex v lies in the range $[\ell_u..r_u]$. Using the data structure of Lemma 7, we try to find in $O(\log^2 n)$ time a marked ancestor v of the leaf corresponding to $s[p{+}1..n]$ such that the weight of v is in the range $[\ell_u..r_u]$ and the string depth of v is at least $|f_1 f_2 \cdots f_h| - p$ (so that the string f_i corresponding to v occurs at position $p{+}1$ and embraces the last position of f_h; see Figure 1 assuming $f_{j+1} = f_h$). If such ancestor exists, we have found f_i. Otherwise, we decide that such $f_i \in S$ does not exist. It is straightforward that in this way we will necessarily find such $f_i \in S$ if it really exists. ◀

4 Linear Algorithm

Now we combine the two approaches described in Sections 2 and 3. On a high level, it is convenient to think that our algorithm is incremental as in Section 2 but it is guaranteed that only at most $\log^3 n$ last phrases from the LZ-End parsing of the currently processed prefix can be removed in the future. (In fact, any polylogarithmic threshold from $\omega(\log^2 n)$ will suffice.) After the processing of a prefix $s[1..k]$, we have the LZ-End parsing $s[1..k] = f_1 f_2 \cdots f_z$ and the phrases of this parsing are split into two groups: a set of first phrases $f_1, f_2, \ldots, f_j$ that cannot be removed from the parsing in the future (this is similar to the approach of Section 3) and at most $\log^3 n$ last phrases $f_{j+1}, f_{j+2}, \ldots, f_z$ that might be removed in the future. The phrases from the former group are called *static*. When the number of non-static phrases exceeds the threshold $\log^3 n$, the algorithm rebuilds the set of non-static phrases and, during this process, possibly marks some of them as static (see the detailed discussion below).

The algorithm maintains a bit array $M[1..n]$ defined as in Section 2 but only for the static phrases: $M[i] = 1$ iff $s[1..\mathsf{SA}[i]] = f_1 f_2 \cdots f_h$ for a static phrase f_h of the current parsing $f_1 f_2 \cdots f_z$. It follows from the above high level description that one can modify M only changing bits to ones. Therefore, the van Emde Boas data structure that answered predecessor/successor queries on M can be replaced with the following *split-find data structure* [12] (the settings of bits to ones can be viewed as splittings of continuous ranges of zeroes.)

▶ **Lemma 10** (see [12]). *There is a (split-find) data structure that, for any $i \in [1..n]$, can find (if any) the maximal $j \le i$ (resp., minimal $j \ge i$) such that $M[j] = 1$ in $O(1)$ time and can perform (at most) n assignments $M[i] \leftarrow 1$ in overall $O(n)$ time.*

▶ **Lemma 11.** *For any $p \in [1..n]$, one can find in $O(1)$ time a static phrase f_i for which the length of the longest common suffix of $s[1..p]$ and $f_1 f_2 \cdots f_i$ is maximal (among all static f_i).*

Proof. The procedure is the same as in Section 2 but now we use the structure of Lemma 10 for predecessor/successor queries. We omit the details as they are straightforward. ◀

An analogous data structure for predecessor/successor queries on non-static phrases is organized using the so-called *fusion tree* [11].

▶ **Lemma 12** (see [11, 29]). *The fusion tree can maintain a set of at most $\log^3 n$ integers under the following operations, each of which takes $O(1)$ time:*
1. *insert an integer x with user-defined satellite information into the set;*
2. *remove an integer x from the set;*
3. *for an integer x, find (if any) in the set the maximal $y \le x$ (resp., minimal $y \ge x$) with the corresponding satellite information.*

▶ **Lemma 13.** *Suppose that, for each phrase f_i from a set S of phrases, the fusion tree stores the number $\mathsf{ISA}[|f_1 f_2 \cdots f_i|]$ with the satellite information containing a pointer to f_i. Then, for any $p \in [1..n]$, one can find in $O(1)$ time a phrase $f_i \in S$ for which the length of the longest common suffix of $s[1..p]$ and $f_1 f_2 \cdots f_i$ is maximal (among all $f_i \in S$).*

Proof. The proof is analogous to the proof of Lemma 11. We omit the obvious details. ◀

During the incremental construction, the fusion tree stores the set of all non-static phrases as described in Lemma 13. Suppose that we have processed a prefix $s[1..k]$ and $f_1 f_2 \cdots f_z$ is the LZ-End parsing of this prefix. To check whether f_z has an earlier occurrence ending at a phrase boundary, we temporarily remove f_z from the fusion tree, apply Lemmas 11 and 13 thus obtaining, respectively, static and non-static phrases f_i and $f_{i'}$ described in these lemma, and use the LCP structure to calculate the lengths of the longest common suffixes of f_z and $f_1 f_2 \cdots f_i$, and of f_z and $f_1 f_2 \cdots f_{i'}$; then, f_z has the required occurrence iff one of these two computed lengths is greater than or equal to $|f_z|$. To check whether $f_{z-1} f_z$ has an earlier occurrence ending at a phrase boundary, we do the same but also temporarily remove f_{z-1} from the fusion tree. After this, the temporarily removed phrases f_{z-1} and f_z are restored. According to the results of the checking, we remove zero, or one (f_z), or two (f_{z-1}, f_z) phrases from the fusion tree and insert a new phrase, resp., $s[k{+}1]$, or $f_z s[k{+}1]$, or $f_{z-1} f_z s[k{+}1]$, thus constructing the parsing of $s[1..k{+}1]$. The whole procedure takes $O(1)$ time. Clearly, such incremental algorithm works in $O(n)$ overall time but sometimes we have a problem: the new non-static phrase inserted in the fusion tree can exceed the limit of $\log^3 n$ elements. Such overflows of the fusion tree are fixed in two ways described below.

Overflows of the fusion tree 1. Let the fusion tree contain the phrases $f_{j+1}, f_{j+2}, \ldots, f_z$ of the LZ-End parsing $f_1 f_2 \cdots f_z$ of $s[1..k]$. Suppose that the fusion tree overflows when the letter $s[k{+}1]$ is appended; obviously, this can happen only if $z - j = \lfloor \log^3 n \rfloor$ and the last phrase of the parsing of $s[1..k{+}1]$ is $s[k{+}1]$. We are to rebuild the current set of non-static phrases $f_{j+1}, f_{j+2}, \ldots, f_z$ (we assume that $s[k{+}1]$ is not inserted in the fusion tree yet) in order to fix the coming overflow. The algorithm maintains a variable t that contains the sum of the lengths of all non-static phrases, i.e., $t = |f_{j+1} f_{j+2} \cdots f_z|$ at the given moment.

We try to unload the fusion tree performing the following procedure consecutively for each of the t positions $k+1, k+2, \ldots, k+t$ from left to right (for simplicity, assume that $k+t < n$; the case $k+t \geq n$ is analogous): for a position p, we apply Lemmas 11, 13 and use the LCP structure in the same way as above in order to check in $O(1)$ time whether the string $s[|f_1 f_2 \cdots f_{z-1}|{+}1..p]$ is a suffix of a string $f_1 f_2 \cdots f_i$ for some $i \in [1..z{-}1]$. Suppose that this checking has succeeded and $q \in [k{+}1..k{+}t]$ is the leftmost position for which $s[|f_1 f_2 \cdots f_{z-1}|{+}1..q]$ is a suffix of $f_1 f_2 \cdots f_i$ for some $i \in [1..z{-}1]$. Then, it follows from Lemma 3 that the parsing of the string $s[1..q{+}1]$ is $f_1 f_2 \cdots f_{z-1} f$, where $f = s[|f_1 f_2 \cdots f_{z-1}|{+}1..q{+}1]$. Hence, once such position q is found, we stop the processing of the positions and modify the fusion tree in $O(1)$ time removing the phrase f_z and putting the new phrase f inside. Since the modified fusion tree contains only the phrases $f_{j+1}, f_{j+2}, \ldots, f_{z-1}, f$ (i.e., the same number $z-j$), it is not overflowed and, therefore, our incremental algorithm can continue the execution from the prefix $s[1..q{+}1]$.

Since each position is analyzed in $O(1)$ time, the processing takes $O(q-k)$ time if such position q was found (and $O(t)$ time otherwise). Therefore, if every overflow of the fusion tree during the work of the algorithm is successfully fixed by the described method, then the construction of the LZ-End parsing of the whole string s takes $O(n)$ time. It remains to consider the case when this method could not find the required position q.

Overflows of the fusion tree 2. As in Section 3, at the beginning, our algorithm builds the suffix tree of s equipped with the data structures of Lemmas 7 and 8. We maintain the following invariant: for each static phrase f_i such that $|f_i| \geq \log^3 n$, the (explicit or implicit) vertex of the suffix tree corresponding to f_i is marked and has weight $\mathsf{ISA}[|f_1 f_2 \cdots f_{i-1}|]$, i.e., the invariant is like in Section 3 but only for static and sufficiently long phrases.

Suppose that, after the fusion tree overflow occurred on the prefix $s[1..k{+}1]$ of s, we processed all $t = |f_{j+1} f_{j+2} \cdots f_z|$ positions $k+1, k+2, \ldots, k+t$ as above but could not "grow" the last phrase f_z of the parsing $f_1 f_2 \cdots f_z$ of $s[1..k]$. We say that a phrase f_h of the parsing is *extendable* if there is $q \geq |f_1 f_2 \cdots f_h|$ such that $s[|f_1 f_2 \cdots f_{h-1}|{+}1..q]$ is a suffix of $f_1 f_2 \cdots f_i$ for some $i \in [1..h{-}1]$. (Note that q cannot be equal to n since we assumed that $s[n]$ does not occur in $s[1..n{-}1]$.) Since the positions $k+1, k+2, \ldots, k+t$ all were unsuccessfully processed by the above procedure trying to "extend" f_z, q must be greater than $k+t$ and, by Lemma 6, the phrase f_i "extending" f_h can be chosen so that f_i starts inside f_h (as in Fig. 1) and has length at least $t+1 \geq \log^3 n$. For simplicity of exposition, we summarize this in the following lemma, which is an easy corollary of Lemma 6.

▶ **Lemma 14.** *Let t be a positive integer. Denote by $f_1 f_2 \cdots f_z$ the LZ-End parsing of a prefix $s[1..k]$ of s. Suppose that, for each $q \in [k..k{+}t]$, there is no $i \in [1..z{-}1]$ such that $s[|f_1 f_2 \cdots f_{z-1}|{+}1..q]$ is a suffix of $f_1 f_2 \cdots f_i$. Then, for any extendable phrase f_h with $h \in [1..z]$, there exist $i \in [1..h{-}1]$ and a position p such that $|f_1 f_2 \cdots f_{h-1}| < p < |f_1 f_2 \cdots f_h|$, $p{+}|f_i| > k{+}t$, f_i occurs at position $p{+}1$, and $s[|f_1 f_2 \cdots f_{h-1}|{+}1..p]$ is a suffix of $f_1 f_2 \cdots f_{i-1}$.*

Since $t = |f_{j+1} f_{j+2} \cdots f_z|$, at most $\log^2 n$ non-static phrases have length $\geq t/\log^2 n$ and most non-static phrases ($\geq z - j - \log^2 n = \lfloor \log^3 n \rfloor - \log^2 n$) have length $< t/\log^2 n$. (The choice of the threshold $t/\log^2 n$ is clarified below.) By a simple traversal of non-static phrases, we find in $O(z-j) \subset O(t)$ time the rightmost non-static phrase f_h such that $|f_h| < t/\log^2 n$. By Lemma 14, if f_h is extendable, then it can be "extended" by a phrase f_i of length $>t$ such that f_i occurs at a position inside f_h. Since $|f_i| > t$ and t is the sum of the lengths of all non-static phrases, f_i must be static. Therefore, by the invariant, the vertex of the suffix tree corresponding to f_i is marked and has weight $\mathsf{ISA}[|f_1f_2\cdots f_{i-1}|]$. Based on this observation, the algorithm decides whether f_h is extendable processing each position of f_h in $O(\log^2 n)$ time by the procedure of Lemma 9 (assuming that the set S from Lemma 9 corresponds to the invariant) in the same way as in Section 3. The overall time of this processing is $O(|f_h|\log^2 n) = O(\frac{t}{\log^2 n}\log^2 n) = O(t)$. (That is why the threshold is $t/\log^2 n$.) The further overflow fixing procedure depends on whether the phrase f_h is extendable.

Suppose that f_h is not extendable (it is a simpler case). Then, the algorithm marks the non-static phrases $f_{j+1}, f_{j+2}, \ldots, f_h$ as static, sets $M[\mathsf{ISA}[|f_1f_2\cdots f_{h'}|]] \leftarrow 1$, for $h' \in [j{+}1..h]$, modifying the data structure of Lemma 10 accordingly, and removes these phrases from the fusion tree. This is correct due to the following straightforward lemma.

▶ **Lemma 15.** *Suppose that $f_1f_2\cdots f_z$ is the LZ-End parsing of a prefix $s[1..k]$ of s. For $h \in [1..z]$, if the phrase f_h is non-extendable, then so are all the phrases $f_1, f_2, \ldots, f_{h-1}$.*

To maintain the invariant, for each new static phrase $f_{h'}$ such that $|f_{h'}| \geq \log^3 n$, the algorithm finds in $O(\log n)$ time using the data structure of Lemma 8 the vertex of the suffix tree corresponding to the string $f_{h'}$ and marks this vertex assigning the weight $\mathsf{ISA}[|f_1f_2\cdots f_{h'-1}|]$ to it in $O(\log n)$ time using the data structure of Lemma 7. After this, only phrases of length $\geq t/\log^2 n$ can remain in the fusion tree. Since there are at most $\log^2 n$ such phrases, the fusion tree is not overflowed and we can continue our incremental algorithm from the prefix $s[1..k{+}1]$ whose parsing is $f_1f_2\cdots f_z s[k{+}1]$. (This case of non-extendable f_h makes the overall time estimation of the algorithm non-trivial; see the discussion below.)

Suppose that f_h is extendable and we found $q > k + t$ and a static phrase f_i such that $s[|f_1f_2\cdots f_{h-1}|{+}1..q]$ is a suffix of $f_1f_2\cdots f_i$ (it is important that f_i is static). We are to compute the LZ-End parsing of the string $s[1..q{+}1]$ based on the following lemma.

▶ **Lemma 16.** *Let $f_1f_2\cdots f_z$ be the LZ-End parsing of a prefix $s[1..k]$ of s. Suppose that, for $h \in [1..z]$ and $q > k$, $s[|f_1f_2\cdots f_{h-1}|{+}1..q]$ is a suffix of $f_1f_2\cdots f_i$ for a static phrase f_i. Then, the LZ-End parsing of $s[1..q{+}1]$ has one of the following forms: (1) $f_1f_2\cdots f_m f$, for $m < h$, or (2) $f_1f_2\cdots f_m f' f$, for $m < h - 1$, such that $f_1f_2\cdots f_z$ is a prefix of $f_1f_2\cdots f_m f'$.*

Proof. Suppose that the LZ-End parsing of $s[1..q]$ coincides with the parsing of $s[1..k]$ on the first d phrases, i.e., $s[1..q] = f_1f_2\cdots f_d f'_1 f'_2\cdots f'_c$ for some $c \geq 1$. It follows from Lemma 2 that $f_1f_2\cdots f_z$ is a prefix of $f_1f_2\cdots f_d f'_1$. Since f_i is static, we have $i \leq d$, i.e., the phrases $f_1, f_2, \ldots, f_i$ are presented in the parsing of $s[1..q]$. Therefore, by Lemma 2, the LZ-End parsing of $s[1..q{+}1]$ is either $f_1f_2\cdots f_d f'_1 f$ (here, the new phrase f "absorbs" the phrases $f'_2, f'_3, \ldots, f'_c$; we put $m := d$ and $f' := f'_1$) or $f_1f_2\cdots f_m f$ for some $m \leq d$ (the new phrase f "absorbs" f'_1 and, probably, some of the phrases $f_d, f_{d-1}, \ldots$). Since f necessarily "absorbs" the phrases $f_hf_{h+1}\cdots f_z$, we have $d < h - 1$ in the former case and, hence, $m < h - 1$. ◀

The main problem is to find f and f' from Lemma 16 (and to determine whether f' really exists). For this, we perform a version of the incremental algorithm for the positions $k+t+1, k+t+2, \ldots, q$ from left to right; the difference is that, during this, we do not store any auxiliary phrases that appear as substrings of $s[k{+}1..q]$ because anyway, by Lemma 16, they are not present in the final parsing of $s[1..q{+}1]$. Let us discuss this in more details.

Let $Q = \{q_1, q_2, \ldots, q_c\}$ be the increasing sequence of all positions from $[k{+}t{+}1..q]$ such that the LZ-End parsing of $s[1..q_d{+}1]$, for any $d \in [1..c]$, has the form $f_1f_2\cdots f_{m_d} f'_d$ for some $m_d \leq z$ and some phrase f'_d. It is convenient to imagine an incremental algorithm (as in Section 2) that builds the parsing of the string $s[1..q{+}1]$ incrementally starting from the parsing of $s[1..k{+}t]$; our goal is to determine the moments when this algorithm passes

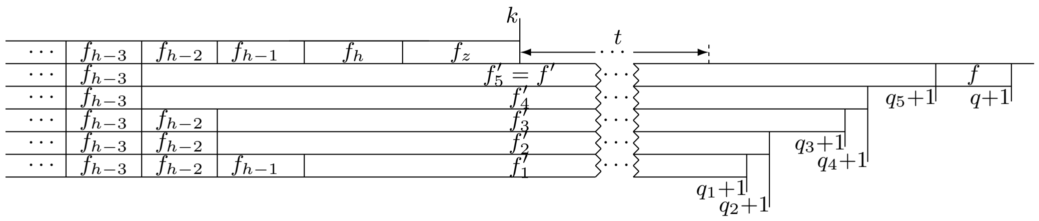

Figure 2 Construction of the parsing from Lemma 16; here $h = z - 1$, $c = 5$, $m_1 = h - 1, m_2 = m_3 = h{-}2, m_4 = m_5 = m = h{-}3$. Each line depicts the parsing of $s[1..q'{+}1]$ for $q' \in \{q_1, q_2, q_3, q_4, q\}$.

the positions from Q (also see Fig. 2 for clarifications). By the choice of q, the string $s[|f_1 f_2 \cdots f_{h-1}|{+}1..q]$ is a suffix of $f_1 f_2 \cdots f_i$, for static phrase f_i, and, therefore, if none of the positions $[k{+}t{+}1..q{-}1]$ belongs to Q, then q must belong to it. By definition of Q, the case (1) of Lemma 16 is realized iff $q_c = q$. Further, the phrases f', f and the number m from Lemma 16 can be determined as follows: $f = f'_c$ and $m = m_c$ if $q_c = q$ (case (1)), and $f' = f'_c$, $f = s[q_c{+}2..q{+}1]$, $m = m_c$ otherwise (case (2)). Note that the "source" of the phrase f found during the calculation of q_c and m_c below might differ from the "source" $f_1 f_2 \cdots f_i$ and might have a longer common suffix with $s[1..q]$. Thus, it remains to compute q_c and m_c through the imitation of the work of our imaginary incremental algorithm.

During the processing of the positions $[k{+}t{+}1..q]$, our real algorithm maintains a variable m that is equal to m_d for the last processed position $q_d \in Q$ (initially, $m = z$) and the fusion tree stores only the phrases $f_{j+1}, f_{j+2}, \ldots, f_m$ (so that, initially, it stores all non-static phrases). For each $p = k + t + 1, k + t + 2, \ldots, q$ from left to right, we apply Lemmas 11, 13 and use the LCP structure (in the same way as in the beginning of this section) to find in $O(1)$ time a phrase $f_{i'}$ such that $f_{i'}$ either is static or is currently in the fusion tree and the length ℓ of the longest common suffix of $s[1..p]$ and $f_1 f_2 \cdots f_{i'}$ is maximal (among all such phrases $f_{i'}$). Then, p belongs to Q iff $\ell \geq p - |f_1 f_2 \cdots f_m|$. Further, if $\ell \geq p - |f_1 f_2 \cdots f_{m-1}|$ and $i' \neq m$, we remove the phrase f_m from the fusion tree and decrease m by one; in the special case when $\ell \geq p - |f_1 f_2 \cdots f_{m-1}|$ and $i' = m$, we remove f_m from the fusion tree, repeat the processing of p, and, if m did not change in this second attempt, restore f_m. The variable m is decreased only by one since, as it follows from Lemmas 2 and 3, for any $d \in [1..c]$, we have either $m_d = m_{d-1}$ or $m_d = m_{d-1} - 1$, assuming $m_0 = z$. The described algorithm computes the numbers q_c and m_c (and, thus, f, f', and m) in $O(q - k)$ time.

We remove the phrases $f_{m+1}, \ldots, f_z$ from the fusion tree and put f and f' (if f' does exist) in it. So, by Lemma 16, the set of non-static phrases of $s[1..q{+}1]$ consists of either $f_{j+1}, f_{j+2}, \ldots, f_m, f$, for $m < h$, or $f_{j+1}, f_{j+2}, \ldots, f_m, f', f$, for $m < h{-}1$. Since $h{-}j \leq z{-}j$, there are at most $z - j = \lfloor \log^3 n \rfloor$ phrases in this set. Therefore, the fusion tree is not overflowed anymore and the algorithm can continue the execution from the prefix $s[1..q{+}1]$.

The correctness of the whole algorithm of this section should be clear at this point.

Time estimation. The algorithm processes each position of s in $O(1)$ time from left to right until it reaches a position $k{+}1$ where the fusion tree overflows when the letter $s[k{+}1]$ is appended. The overflow is fixed in two ways. First, the algorithm processes each of the positions $k + 1, k + 2, \ldots, k + t$ in $O(1)$ time from left to right, for an appropriate value of t, until it finds a position q such that our usual algorithm can continue the execution from the prefix $s[1..q{+}1]$ with the fixed non-overflowed fusion tree. It is obvious that all fixing procedures of this kind take $O(n)$ overall time. Thus, it remains to consider the time required to fix the overflows in which the processing of the corresponding positions $k + 1, k + 2, \ldots, k + t$ could not help; we refer to the overflows of this kind as *hard overflows*.

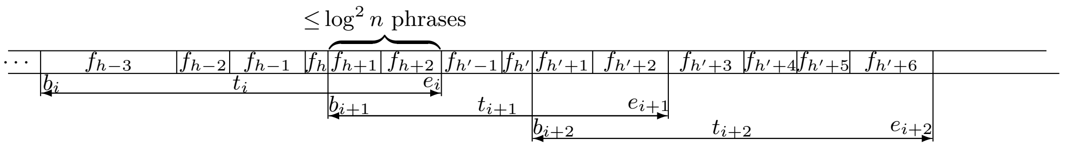

Figure 3 The case in the proof of Lemma 17 when f_h and $f_{h'}$ both are not extendable. The depicted phrases are from the LZ-End parsing of the prefix $s[1..e_{i+2}]$.

To maintain the invariant, the algorithm marks in the suffix tree the vertices corresponding to the static phrases of length at least $\log^3 n$. As there are at most $O(n/\log^3 n)$ such phrases and each marking takes $O(\log n)$ time, the overall time required for the maintenance of the invariant is $o(n)$ and, hence, we can exclude the time spent on these markings from the consideration. Denote by t_i the value of the variable t at the moment when the ith hard overflow occurs. Suppose that the ith hard overflow occurs when the algorithm reaches a prefix $s[1..k_i]$, for some k_i, and tries to process $s[1..k_i{+}1]$. The processing of this hard overflow takes $O(t_i + q_i - k_i)$ time, where $s[1..q_i{+}1]$ is a prefix from which the algorithm continues its execution after the fixing of the overflow. It is easy to see that $\sum_i (t_i + q_i - k_i) = O(n) + \sum_i t_i$ and, hence, it suffices to prove that, for any input string $s[1..n]$, we have $\sum t_i = O(n)$.

Consider the ith hard overflow. Suppose that it occurs on a prefix $s[1..k]$ with the LZ-End parsing $f_1 f_2 \cdots f_z$ and the fusion tree contains the phrases $f_{j+1}, f_{j+2}, \ldots, f_z$ at this moment. Denote $b_i = |f_1 f_2 \cdots f_j| + 1$ and $e_i = |f_1 f_2 \cdots f_z|$ ("b" and "e" are shortenings for "begin" and "end"). Since $t_i = |f_{j+1} f_{j+2} \cdots f_z|$, we have $e_i = b_i + t_i - 1$.

▶ **Lemma 17.** *Suppose that d is the number of hard overflows occurred during the processing of a string s. Then, for any $i \in [1..d{-}2]$, we have $b_{i+2} + e_{i+2} \geq b_i + e_i + t_i$.*

Proof. Note that the sequences $\{b_i\}$ and $\{e_i\}$ are non-decreasing and $b_i < e_i$ for any $i \in [1..d]$. Recall that the ith hard overflow occurs after the processing of the prefix $s[1..e_i]$ and the procedure fixing the overflow tries to "extend" a non-static phrase f_h of the LZ-End parsing of $s[1..e_i]$. Due to Lemma 14, if f_h is extendable, then the algorithm continues its execution from a prefix $s[1..q{+}1]$ for some $q > e_i + t_i$. Therefore, we obtain $e_{i+1} \geq q > e_i + t_i$ and, hence, $b_i + e_i + t_i < b_i + e_{i+1} \leq b_{i+2} + e_{i+2}$.

Suppose that f_h is not extendable. Then, the algorithm marks f_h and all phrases to the left of f_h as static and only at most $\log^2 n$ phrases (of length $\geq t_i/\log^2 n$) remain in the fusion tree (see Fig. 3 and 4). Now consider the $(i{+}1)$st hard overflow. It follows from the above discussion that only at most $\log^2 n$ first phrases of the fusion tree can contain phrases of the parsing of $s[1..e_i]$ at this moment. The procedure fixing the $(i{+}1)$st hard overflow analogously tries to "extend" a non-static phrase $f_{h'}$ of the LZ-End parsing of $s[1..e_{i+1}]$. This phrase $f_{h'}$ is the rightmost phrase of length $< t_{i+1}/\log^2 n$. Since there are at least $\lfloor \log^3 n \rfloor - \log^2 n$ phrases of length $< t_{i+1}/\log^2 n$, the phrase $f_{h'}$ cannot coincide with any of the phrases from the parsing of $s[1..e_i]$ and, therefore, it must occur at a position to the right of the position e_i (see Fig. 3 and 4 for a clarification).

Suppose that $f_{h'}$ is not extendable (see Fig. 3). Then, the algorithm marks $f_{h'}$ and all phrases to the left of $f_{h'}$ as static. Hence, during the $(i{+}2)$nd hard overflow, b_{i+2} must be greater than the rightmost position of $f_{h'}$ and, thus, $b_{i+2} > e_i$ (see Fig. 3). Since $e_i = b_i + t_i - 1$, the later implies $b_i + t_i \leq b_{i+2}$ and, hence, $b_i + e_i + t_i \leq b_{i+2} + e_{i+2}$.

Suppose that $f_{h'}$ is extendable. Since $t_i \leq (b_{i+1} - b_i) + t_{i+1}$ (see Fig. 4), we derive $b_i + e_i + t_i \leq b_i + e_i + (b_{i+1} - b_i) + t_{i+1} = b_{i+1} + e_i + t_{i+1} \leq b_{i+1} + e_{i+1} + t_{i+1}$. By Lemma 14, since $f_{h'}$ is found to be extendable, after the $(i{+}1)$st hard overflow the algorithm continues

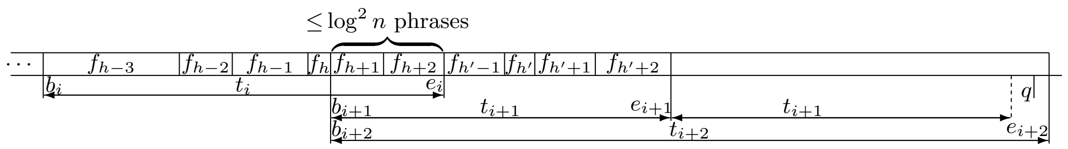

Figure 4 The case in the proof of Lemma 17 when f_h is not extendable and $f_{h'}$ is extendable. The depicted phrases are from the LZ-End parsing of the prefix $s[1..e_{i+1}]$.

its execution from a prefix $s[1..q{+}1]$ for some $q > e_{i+1} + t_{i+1}$. Therefore, we obtain $e_{i+2} \geq q > e_{i+1} + t_{i+1}$ (see Fig. 4), which implies $b_i + e_i + t_i \leq b_{i+1} + e_{i+1} + t_{i+1} < b_{i+2} + e_{i+2}$. ◀

It follows from Lemma 17 that $\sum_{i=1}^{d-2} t_i \leq \sum_{i=1}^{d-2}(b_{i+2} + e_{i+2} - b_i - e_i) = b_d + e_d + b_{d-1} + e_{d-1} - b_1 - e_1 - b_2 - e_2$, which is obviously $O(n)$. Therefore, since $t_d + t_{d-1} = O(n)$, we obtain $\sum_{i=1}^{d} t_i = O(n)$. This finally proves Theorem 1.

References

1 P. Beame and F. E. Fich. Optimal bounds for the predecessor problem. In *STOC 1999*, pages 295–304. ACM, 1999. `doi:10.1006/jcss.2002.1822`.

2 D. Belazzougui and S. J. Puglisi. Range predecessor and Lempel–Ziv parsing. In *SODA 2016*, pages 2053–2071. SIAM, 2016. `doi:10.1137/1.9781611974331.ch143`.

3 G. E. Blelloch. Space-efficient dynamic orthogonal point location, segment intersection, and range reporting. In *SODA 2008*, pages 894–903. SIAM, 2008.

4 F. Claude, A. Fariña, M. A. Martínez-Prieto, and G. Navarro. Universal indexes for highly repetitive document collections. *Information Systems*, 61:1–23, 2016. `doi:10.1016/j.is.2016.04.002`.

5 T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms, third edition*. MIT press, 2009.

6 M. Crochemore and W. Rytter. *Jewels of stringology*. World Scientific Publishing Co. Pte. Ltd., 2002.

7 H. Ferrada, T. Gagie, T. Hirvola, and S. J. Puglisi. Hybrid indexes for repetitive datasets. *Phil. Trans. R. Soc. A*, 372, 2014. `doi:10.1098/rsta.2013.0137`.

8 P. Ferragina and G. Manzini. On compressing the textual web. In *WSDM 2010*, pages 391–400. ACM, 2010. `doi:10.1145/1718487.1718536`.

9 J. Fischer, T. Gagie, P. Gawrychowski, and T. Kociumaka. Approximating LZ77 via small-space multiple-pattern matching. In *ESA 2015*, volume 9294 of *LNCS*, pages 533–544. Springer, 2015. `doi:10.1007/978-3-662-48350-3_45`.

10 J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *CPM 2006*, volume 4009 of *LNCS*, pages 36–48. Springer, 2006. `doi:10.1007/11780441_5`.

11 M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993. `doi:10.1016/0022-0000(93)90040-4`.

12 H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. In *STOC 1983*, pages 246–251. ACM, 1983. `doi:10.1145/800061.808753`.

13 T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. A faster grammar-based self-index. In *LATA 2012*, volume 7183 of *LNCS*, pages 240–251. Springer, 2012. `doi:10.1007/978-3-642-28332-1_21`.

14 T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. LZ77-based self-indexing with faster pattern matching. In *LATIN 2014*, volume 8392 of *LNCS*, pages 731–742. Springer, 2014. `doi:10.1007/978-3-642-54423-1_63`.

15 T. Gagie, P Gawrychowski, and S. J. Puglisi. Faster approximate pattern matching in compressed repetitive texts. In *ISAAC 2011*, volume 7074 of *LNCS*, pages 653–662. Springer, 2011. doi:10.1007/978-3-642-25591-5_67.

16 P. Gawrychowski, M. Lewenstein, and P. K. Nicholson. Weighted ancestors in suffix trees. In *ESA 2014*, volume 8737 of *LNCS*, pages 455–466. Springer, 2014. doi:10.1007/978-3-662-44777-2_38.

17 J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Linear time Lempel–Ziv factorization: Simple, fast, small. In *CPM 2013*, volume 7922 of *LNCS*, pages 189–200. Springer, 2013. doi:10.1007/978-3-642-38905-4_19.

18 J. Kärkkäinen, D. Kempa, and S. J Puglisi. Lempel-Ziv parsing in external memory. In *DCC 2014*, pages 153–162. IEEE, 2014. doi:10.1109/DCC.2014.78.

19 D. Kempa and D. Kosolobov. LZ-End parsing in compressed space. In *DCC 2017*, pages 350–359. IEEE, 2017. doi:10.1109/DCC.2017.73.

20 T. Kopelowitz and M. Lewenstein. Dynamic weighted ancestors. In *SODA 2007*, pages 565–574. SIAM, 2007.

21 D. Kosolobov. Faster lightweight Lempel–Ziv parsing. In *MFCS 2015*, volume 9235 of *LNCS*, pages 432–444, 2015. doi:10.1007/978-3-662-48054-0_36.

22 S. Kreft and G. Navarro. LZ77-like compression with fast random access. In *DCC 2010*, pages 239–248. IEEE, 2010. doi:10.1109/DCC.2010.29.

23 S. Kreft and G. Navarro. Self-indexing based on LZ77. In *CPM 2011*, volume 6661 of *LNCS*, pages 41–54. Springer, 2011. doi:10.1007/978-3-642-21458-5_6.

24 S. Kreft and G. Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013. doi:10.1016/j.tcs.2012.02.006.

25 V. Mäkinen and G. Navarro. Compressed full-text indexes. *ACM Computing Surveys (CSUR)*, 39(1):2, 2007. doi:10.1145/1216370.1216372.

26 J. I. Munro, G. Navarro, and Y. Nekrich. Space-efficient construction of compressed indexes in deterministic linear time. In *SODA 2017*, pages 408–424. SIAM, 2017. doi:10.1137/1.9781611974782.26.

27 G. Navarro. Indexing text using the Ziv-Lempel trie. *J. Discrete Algorithms*, 2(1):87–114, 2004. doi:10.1016/S1570-8667(03)00066-2.

28 M. Pătraşcu and M. Thorup. Time-space trade-offs for predecessor search. In *STOC 2006*, pages 232–240. ACM, 2006. doi:10.1145/1132516.1132551.

29 M. Pătraşcu and M. Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. In *FOCS 2014*, pages 166–175. IEEE, 2014. doi:10.1109/FOCS.2014.26.

30 A. Policriti and N. Prezza. Computing LZ77 in run-compressed space. In *DCC 2016*, pages 23–32. IEEE, 2016. doi:10.1109/DCC.2016.30.

31 J. Sirén, N. Välimäki, V. Mäkinen, and G. Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In *SPIRE 2008*, volume 5280 of *LNCS*, pages 164–175. Springer, 2008. doi:10.1007/978-3-540-89097-3_17.

32 D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983. doi:10.1016/0022-0000(83)90006-5.

33 P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical systems theory*, 10(1):99–127, 1976. doi:10.1007/BF01683268.

34 J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Information Theory*, 23(3):337–343, 1977. doi:10.1109/TIT.1977.1055714.