Space-Efficient Construction of Compressed Indexes in Deterministic Linear Time

J. Ian Munro∗  Gonzalo Navarro†  Yakov Nekrich‡

Abstract

We show that the compressed suffix array and the compressed suffix tree of a string \(T\) can be built in \(O(n)\) deterministic time using \(O(n \log \sigma)\) bits of space, where \(n\) is the string length and \(\sigma\) is the alphabet size. Previously described deterministic algorithms either run in time that depends on the alphabet size or need \(\omega(n \log \sigma)\) bits of working space. Our result has immediate applications to other problems, such as yielding the first deterministic linear-time LZ77 and LZ78 parsing algorithms that use \(O(n \log \sigma)\) bits.

1 Introduction

In the string indexing problem we pre-process a string \(T\), so that for any query string \(P\) all occurrences of \(P\) in \(T\) can be found efficiently. Suffix trees and suffix arrays are two most popular solutions of this fundamental problem. A suffix tree is a compressed trie on suffixes of \(T\); it enables us to find all occurrences of a string \(P\) in \(T\) in time \(O(|P| + \text{occ})\) where occ is the number of times \(P\) occurs in \(T\) and \(|P|\) denotes the length of \(P\). In addition to indexing, suffix trees also support a number of other, more sophisticated, queries. The suffix array of a string \(T\) is the lexicographically sorted array of its suffixes. Although suffix arrays do not support all queries that can be answered by the suffix tree, they use less space and are more popular in practical implementations. While the suffix tree occupies \(O(n \log n)\) bits of space, the suffix array can be stored in \(n \log n\) bits.

During the last twenty years there has been a significant increase in interest in compressed indexes, i.e., data structures that keep \(T\) in compressed form and support string matching queries. The compressed suffix array (CSA) [19, 13, 38] and the compressed suffix tree (CST) [39] are compressed counterparts of the suffix array and the suffix tree respectively. A significant part of compressed indexes relies on these two data structures or their variants. Both CSA and CST can be stored in \(O(n \log \sigma)\) bits or less; we refer to e.g. [6] or [32] for an overview of compressed indexes.

It is well known that both the suffix array and the suffix tree can be constructed in \(O(n)\) time [28, 42, 43, 23]. The first algorithm that constructs the suffix tree in linear time independently of the alphabet size was presented by Farach [12]. There are also algorithms that directly construct the suffix array of \(T\) in \(O(n)\) time [22, 24]. If the (uncompressed) suffix tree is available, we can obtain CST and CSA in \(O(n)\) time. However this approach requires \(O(n \log n)\) bits of working space. The situation is different if we want to construct compressed variants of these data structures using only \(O(n \log \sigma)\) bits of space. Within this space the algorithm of Hon et al. [21] constructs the CST in \(O(n \log^\epsilon n)\) time for an arbitrarily small constant \(\epsilon > 0\). In the same paper the authors also showed that CSA can be constructed in \(O(n \log \log n)\) time. The algorithm of Okanohara and Sadakane constructs the CSA in linear time, but needs \(O(n \log \sigma \log \log n)\) bits of space [36]. Belazzougui [1] described randomized algorithms that build both CSA and CST in \(O(n)\) time and \(O(n \log \sigma)\) bits of space. His approach also provides deterministic algorithms with runtime \(O(n \log \log \sigma)\) [2]. In this paper we show that randomization is not necessary in order to construct CSA and CST in linear time. Our algorithms run in \(O(n)\) deterministic time and require \(O(n \log \sigma)\) bits of space.

Suffix trees, in addition to being an important part of many compressed indexes, also play an important role in many string algorithms. One prominent example is Lempel-Ziv parsing of a string using \(O(n \log \sigma)\) bits. The best previous solutions for this problem either take \(O(n \log \log \sigma)\) deterministic time or \(O(n)\) randomized time [25, 9]. For instance Köppl and Sadakane [25] showed how we can obtain LZ77- and LZ78-parsing for a string \(T\) in \(O(n)\) deterministic time and \(O(n \log \sigma)\) bits, provided that the CST of \(T\) is constructed. Thus our algorithm, combined with their results, leads to the first linear-time deterministic LZ-parsing algorithm that
needs $O(n \log \sigma)$ bits of space.

**Overview.** The main idea of our approach is the use of batch processing. Certain operations, such as rank and select queries on sequences, are a bottleneck of previous deterministic solutions. Our algorithms are divided into a large number of small tasks that can be executed independently. Hence, we can collect large batches of queries and answer all queries in a batch. This approach speeds up the computation because, as will be shown later, answering all queries in a batch takes less time than answering the same set of queries one-by-one. For example, our algorithm for generating the Burrows-Wheeler Transform of a text $T$ works as follows. We cut the original text into slices of $\Delta = \log_\sigma n$ symbols. The BWT sequence is constructed by scanning all slices in the right-to-left order. All slices are processed at the same time. That is, the algorithm works in $\Delta$ steps and during the $j$-th step, for $0 \leq j \leq \Delta - 1$, we process all suffixes that start at position $i\Delta - j - 1$ for all $1 \leq i \leq n/\Delta$. Our algorithm maintains the sorted list of suffixes and keeps information about those suffixes in a symbol sequence $B$. For every suffix $S_i = T[i\Delta - j - 1..]$ processed during the step $j$, we must find its position in the sorted list of suffixes. Then the symbol $T[i\Delta - j - 2]$ is inserted at the position that corresponds to $S_i$ in $B$. Essentially we can find the position of every new suffix $S_i$ by answering a rank query on the sequence $B$. Details are given in Section 2. Next we must update the sequence by inserting the new symbols into $B$. Unfortunately we need $\Omega(\log n/\log \log n)$ time in general to answer rank queries on a dynamic sequence [15]. Even if we do not have to update the sequence, we need $\Omega(\log \log \sigma)$ time to answer a rank query [7]. In our case, however, the scenario is different: There is no need to answer queries one-by-one. We must provide answers to a large batch of $n/\Delta$ rank queries with one procedure. In this paper we show that the lower bounds for rank queries can be circumvented in the batched scenario: we can answer the batch of queries in $O(n/\Delta)$ time, i.e., in constant time per query. We also demonstrate that a batch of $n/\Delta$ insertions can be processed in $O(n/\Delta)$ time. This result is of independent interest.

Data structures that answer batches of rank queries and support batched updates are described in Sections 3, A.2, and A.3. This is the most technically involved aspect of our result. In Section 3 we show how answers to a large batch of queries can be provided. In Section A.2 we describe a special labeling scheme that assigns monotonously increasing labels to elements of a list. We conclude this portion in Section A.3 where we show how the static data structure can be dynamized. Next we turn to the problem of constructing the compressed suffix tree. First we describe a data structure that answers partial rank queries in constant time and uses $O(n \log \log \sigma)$ additional bits in Section A.4: unlike previous solutions, our data structure can be constructed in $O(n)$ deterministic time. This result is plugged into the algorithm of Belazzougui [1] to obtain the suffix tree topology in $O(n)$ deterministic time. Finally we show how the permuted LCP array (PLCP) can be constructed in $O(n)$ time, provided we already built the suffix array and the suffix tree topology; the algorithm is described in Section 5. Our algorithm for constructing PLCP is also based on batch processing of rank queries. To make this paper self-contained we provide some background on compressed data structures and indexes in Section A.1.

We denote by $T[i..]$ the substring of $T$ that begins with $T[i]$ and ends with $T[j]$, $T[i..] = T[i]T[i+1]...T[j]$ and $T[i..] = T[i]T[i+1]...T[j-1]T[j]$. We assume that the text $T$ ends with a special symbol $\$ and $\$ lexicographically precedes all other symbols in $T$. The alphabet size is $\sigma$ and symbols are integers in $[0..\sigma - 1]$ (so $\$ corresponds to 0). In this paper, as in the previous papers on this topic, we use the word RAM model of computation. A machine word consists of $\log n$ bits and we can execute standard bit operations, addition and subtraction in constant time. We will assume for simplicity that the alphabet size $\sigma \leq n^{1/4}$. This assumption is not restrictive because for $\sigma > n^{1/4}$ linear-time algorithms that use $O(n \log \sigma) = O(n \log n)$ bits are already known.

2 Linear Time Construction of the Burrows-Wheeler Transform

In this section we show how the Burrows-Wheeler transform (BWT) of a text $T$ can be constructed in $O(n)$ time using $O(n \log \sigma)$ bits of space. Let $\Delta = \log_\sigma n$. We can assume w.l.o.g. that the text length is divisible by $\Delta$ (if this is not the case we can pad the text $T$ with $\lfloor n/\Delta \rfloor \Delta - n$ $\$-symbols). The BWT of $T$ is a sequence $B$ defined as follows: if $T[k..]$ is the $(i + 1)$-th lexicographically smallest suffix, then $B[i] = T[k - 1]$.\footnote{So $B[0]$ has the lexicographically smallest suffix $(i + 1 = 1)$ and so on. The exact formula is $B[i] = T[(k - 1) \mod n]$. We will write $B[i] = T[k - 1]$ to avoid tedious details.} Thus the symbols of $B$ are the symbols that precede the suffixes of $T$, sorted in lexicographic order. We will say that $T[k - 1]$ represents the suffix $T[k..]$ in $B$. Our algorithm divides the suffixes of $T$ into $\Delta$ classes and constructs $B$ in $\Delta$ steps. We say that a suffix $S$ is a $j$-suffix for $0 \leq j < \Delta$ if $S = T[i\Delta - j - 1..]$ for some $i$, and denote by $S_j$ the set of all $j$-suffixes,
$S_j = \{ T[i\Delta - j - 1..] | 1 \leq i \leq n/\Delta \}$. During the $j$-th step we process all $j$-suffixes and insert symbols representing $j$-suffixes at appropriate positions of the sequence $B$.

Steps 0–1. We sort suffixes in $S_0$ and $S_1$ by constructing a new text and representing it as a sequence of $n/\Delta$ meta-symbols. Let $T_1 = T[n-1]T[0]T[1] \ldots T[n-2]$ be the text $T$ rotated by one symbol to the right and let $T_2 = T[n-2]T[n-1]T[0] \ldots T[n-3]$ be the text obtained by rotating $T_1$ one symbol to the right. We represent $T_1$ and $T_2$ as sequences of length $n/\Delta$ over meta-alphabet $\sigma^\Delta$ (each meta-symbol corresponds to a string of length $\Delta$). Thus we view $T_1$ and $T_2$ as sequences of meta-symbols; see Fig. 1.

Let $T_3 = T_1 \circ T_2$ denote the concatenation of $T_1$ and $T_2$. To sort the suffixes of $T_3$, we sort the meta-symbols of $T_3$ and rename them with their ranks. Since meta-symbols correspond to $(\log n)$-bit integers, we can sort them in time $O(n)$ using radix sort. Then we apply a linear-time and linear-space suffix array construction algorithm [22] to $T_3$. We thus obtain a sorted list of suffixes $L$ for the meta-symbol sequence $T_3$. Suffixes of $T_3$ correspond to the suffixes from $S_0 \cup S_1$ in the original text $T$: the suffix $T[i\Delta - 1..]$ corresponds to the suffix of $S_0$ starting with meta-symbol $[T[i\Delta - 1..]T[i\Delta]..]$ in $T_3$ and the suffix $T[i\Delta - 2..]$ corresponds to the suffix of $S_1$ starting with $[T[i\Delta - 2..]T[i\Delta]..]$. Since we assume that the special symbol $\$ is smaller than all other symbols, this correspondence is order-preserving. Hence by sorting the suffixes of $T_3$ we obtain the sorted list $L'$ of suffixes in $S_0 \cup S_1$. Now we are ready to insert symbols representing $j$-suffixes into $B$: Initially $B$ is empty. Then the list $L'$ is traversed and for every suffix $T[k..]$ that appears in $L'$ we add the symbol $T[k-1]$ at the end of $B$.

When suffixes in $S_0$ and $S_1$ are processed, we need to record some information for the next step of our algorithm. For every suffix $S \in S_1$ we keep its position in the sorted list of suffixes. The position of suffix $T[i\Delta - 2..]$ is stored in the entry $W[i]$ of an auxiliary array $W$, which at the end of the $j$-th step will contain the positions of the suffixes $T[i\Delta - j - 1..]$. We also keep an auxiliary array $Acc$ of size $\sigma$: $Acc[a]$ is equal to the number of occurrences of symbols $i \leq a - 1$ in the current sequence $B$.

Step j for $j \geq 2$. Suppose that suffixes from $S_0$, $\ldots$, $S_{j-1}$ are already processed. The symbols that precede suffixes from these sets are stored in the sequence $B$; the $k$-th symbol $B[k]$ in $B$ is the symbol that precedes the $k$-th lexicographically smallest suffix from $\cup_{i=0}^{j-1} S_i$. For every suffix $T[i\Delta - j - 1..]$, we know its position $W[i]$ in $B$. Every suffix $S_i = T[i\Delta - j - 1..] \in S_j$ can be represented as $S_i = aS_i'$ for some symbol $a$ and the suffix $S_i' = T[i\Delta - j..] \in S_{j-1}$. We look up the position $t_i = W[i]$ of $S_i'$ and answer rank query $r_i = \text{rank}_a(t_i, B)$. We need $\Omega(\log \log \log n)$ time to answer a single rank query on a static sequence [7]. If updates are to be supported, then we need $\Omega(\log n/\log \log n)$ time to answer such a query [15]. However in our case the scenario is different: we perform a batch of $n/\Delta$ queries to sequence $B$, i.e., we have to find $r_i$ for all $t_i$.

During Step 2 the number of queries is equal to $|B|/2$ where $|B|$ denotes the number of symbols in $B$. During step $j$ the number of queries is $|B|/j \geq |B|/\Delta$. We will show in Section 3 that such a large batch of rank queries can be answered in $O(1)$ time per query. Now we can find the rank $p_i$ of $S_i$ among $\cup_{i=1}^{j-1} S_i$: there are exactly $p_i$ suffixes $S_i \in \cup_{i=1}^{j-1} S_i$ that are smaller than $S_i$, where $p_i = Acc[a] + r_i$. Correctness of this computation can be proved as follows.

**Proposition 2.1.** Let $S_i = aS_i'$ be an arbitrary suffix from the set $S_j$. For every occurrence of a symbol $a' < a$ in the sequence $B$, there is exactly one suffix $S_p < S_i$ in $\cup_{i=1}^{j-1} S_i$, such that $S_p$ starts with $a'$. Further, there are exactly $r_i$ suffixes $S_v$ in $\cup_{i=1}^{j-1} S_i$ such that $S_v \leq S_i$ and $S_v$ starts with $a$.

**Proof.** Suppose that a suffix $S_p$ from $S_i$, such that $j \geq t \geq 2$, starts with $a' < a$. Then $S_p = a' S_p'$ for some $S_p' \in S_{t-1}$. By definition of the sequence $B$, there is exactly one occurrence of $a'$ in $B$ for every such $S_p'$. Now suppose that a suffix $S_i \in S_i$, such that $j \geq t \geq 2$, starts with $a$ and $S_v \leq S_i$. Then $S_v = a S_v'$ for some $S_v' \in S_{t-1}$ and $S_v' \leq S_i$. For every such $S_v'$ there is exactly one occurrence of the symbol $a$ in $B[1..t_i]$, where $t_i$ is the position of $S_i'$ in $B$.

The above calculation did not take into account the suffixes from $S_0$. We compute the number of suffixes $S_k \in S_0$ such that $S_k < S_i$ using the approach of Step 0–1. Let $T_1$ be the text obtained by rotating $T$ one symbol to the right. Let $T'$ be the text obtained by rotating $T$ $j + 1$ symbols to the right. We can sort suffixes of $S_0$ and $S_j$ by concatenating $T_1$ and $T'$, viewing the resulting text $T''$ as a sequence of $2n/\Delta$ meta-symbols and constructing the suffix array for $T''$. When suffixes in $S_0 \cup S_j$ are sorted, we traverse the sorted list of suffixes; for every suffix $S_i \in S_j$ we know the number $q_i$ of lexicographically smaller suffixes from $S_0$.

We then modify the sequence $B$: We sort new suffixes $S_i$ by $o_i = p_i + q_i$. Next we insert the symbol $T[i\Delta - j - 1]$ at position $o_i + 1$ in $B$ (assuming the first index of $B$ is $B[0]$): insertions are performed in increasing order of $o_i$. We will show that this procedure
also takes $O(1)$ time per update for a large batch of insertions. Finally we record the position of every new suffix from $S_j$ in the sequence $B$. Since the positions of suffixes from $S_{j-1}$ are not needed any more, we use the entry $W[i]$ of $W$ to store the position of $T[i \Delta - j - 1]$. The array $Acc$ is also updated.

When Step $\Delta - 1$ is completed, the sequence $B$ contains $n$ symbols and $B[i]$ is the symbol that precedes the $(i + 1)$-th smallest suffix of $T$. Thus we obtained the BWT of $T$. Step 0 of our algorithm uses $O((n/\Delta) \log n) = O(n \log \sigma)$ bits. For all the following steps we need to maintain the sequence $B$ and the array $W$. $B$ uses $O(\log \sigma)$ bits per symbol and $W$ needs $O((n/\Delta) \log n) = O(n \log \sigma)$ bits. Hence our algorithm uses $O(n \log \sigma)$ bits of workspace. Procedures for querying and updating $B$ are described in the following section. Our result can be summed up as follows.

**Theorem 2.1.** Given a string $T[0..n-1]$ over an alphabet of size $\sigma$, we can construct the BWT of $T$ in $O(n)$ deterministic time using $O(n \log \sigma)$ bits.

### 3 Batched Rank Queries on a Sequence

In this section we show how a batch of $m$ rank queries for $\frac{n}{\log n} \leq m \leq n$ can be answered in $O(m)$ time on a sequence $B$ of length $n$. We start by describing a static data structure. A data structure that supports batches of queries and batches of insertions will be described later. We will assume $\sigma \geq \log^3 n$; if this is not the case, the data structure from [14] can be used to answer rank queries in time $O(1)$.

Following previous work [17], we divide $B$ into chunks of size $\sigma$ (except for the last chunk that contains at most $\sigma$ symbols). For every symbol $a$ we keep a binary sequence $M_a = 1^{d_1}01^{d_2}0 \ldots 1^{d_f}$ where $f$ is the total number of chunks and $d_i$ is the number of occurrences of $a$ in the chunk. We keep the following information for every chunk $C$. Symbols in a chunk $C$ are represented as pairs $(a, i)$: we store a pair $(a, i)$ if and only if $C[i] = a$. These pairs are sorted by symbols and pairs representing the same symbol $a$ are sorted by their positions in $C$; all sorted pairs from a chunk are kept in a sequence $R$. The array $F$ consists of $\sigma$ entries; $F[a]$ contains a pointer to the first occurrence of a symbol $a$ in $R$ (or null if $a$ does not occur in $C$). Let $R_a$ denote the subsequence of $R$ that contains all pairs $(a, \cdot)$ for some symbol $a$. If $R_a$ contains at least $\log^2 n$ pairs, we split $R_a$ into groups $H_{a,r}$ of size $\Theta(\log^2 n)$. For every group, we keep its first pair in the sequence $R'$. Thus $R'$ is also a subsequence of $R$. For each pair $(a', i')$ in $R'$ we also store the partial rank of $C[i']$ in $C$, rank$_{C[i']}((i', C)$.

All pairs in $H_{a,r}$ are kept in a data structure $D_{a,r}$ that contains the second components of pairs $(a, i) \in H_{a,r}$. Thus $D_{a,r}$ contains positions of $\Theta(\log^2 n)$ consecutive symbols $a$. If $R_a$ contains less than $\log^2 n$ pairs, we keep all pairs starting with symbol $a$ in one group $H_{a0}$. Every $D_{a,r}$ contains $O(\log^2 n)$ elements. Hence we can implement $D_{a,r}$ so that predecessor queries are answered in constant time: for any integer $q$, we can find the largest $x \in H_{a,r}$ satisfying $x \leq q$ in $O(1)$ time [16]. We can also find the number of elements $x \in H_{a,r}$ satisfying $x \leq q$ in $O(1)$ time. This operation on $H_{a,r}$ can be implemented using bit techniques similar to those suggested in [34]; details are to be given in the full version of this paper.

**Queries on a Chunk.** Now we are ready to answer a batch of queries in $O(1)$ time per query. First we describe how queries on a chunk can be answered. Answering a query rank$_a(i, C)$ on a chunk $C$ is equivalent to counting the number of pairs $(a, j)$ in $R$ such that $j \leq i$. Our method works in three steps. We start by sorting the sequence of all queries on $C$. Then we “merge” the sorted query sequence with $R'$. That is, we find for every rank$_a(i, C)$ the rightmost pair $(a, j')$ in $R'$, such that $j' \leq i$. Pair $(a, j')$ gives us with an approximate answer to rank$_a(i, C)$ (up to an additive $O(\log^2 n)$ term). Then we obtain the exact answer to each query by searching in some data structure $D_{a,j}$. Since $D_{a,j}$ contains only $O(\log^2 n)$ elements, the search can be completed in $O(1)$ time. A more detailed description follows.

Suppose that we must answer $v$ queries rank$_a(i_1, C)$, rank$_a(i_2, C)$, ..., rank$_a(i_v, C)$ on a chunk $C$. We sort the sequence of queries by pairs $(a_j, i_j)$ in increasing order. This sorting step takes $O(\sigma/\log^2 n + v)$ time, where $v$ is the number of queries: if $v < \sigma/\log^3 n$, we sort in $O(v \log n) = O(\sigma/\log^2 n)$ time; if $v \geq \sigma/\log^3 n$, we sort in $O(v)$ time using radix
sort (e.g., with radix $\sqrt{\sigma}$). Then we simultaneously traverse the sorted sequence of queries and $R'$; for each query pair $(a_j, i_j)$ we identify the pair $(a_t, t_i)$ in $R'$ such that either (i) $p_t \leq i_j \leq p_{t+1}$ and $a_j = a_t = a_{t+1}$ or (ii) $p_t \leq i_j$, $a_j = a_t$, and $a_t \neq a_{t+1}$. That is, we find the largest $p_t \leq i_j$ such that $(a_t, p_t) \in R'$ for every query pair $(a_j, i_j)$. If $(a_t, p_t)$ is found, we search in the group $H_{a_t, p_t}$ that starts with the pair $(a_t, p_t)$. If the symbol $a_j$ does not occur in $R'$, then we search in the leftmost group $H_{a_t, 0}$. Using $D_{a_t, p_t}$ (resp. $D_{a_t, 0}$), we find the largest position $x_i \in H_{a_t, p_t}$ such that $x_i \leq i_j$. Thus $x_t$ is the largest position in $C$ satisfying $x_t \leq i_j$ and $C[x_t] = a_j$. We can then compute $\text{rank}_{a_t}(x_t, C)$ as follows: Let $n_1$ be the partial rank of $C[p_t]$, $n_1 = \text{rank}_{C[p_t]}(p_t, C)$. Recall that we explicitly store this information for every position in $R'$. Let $n_2$ be the number of positions $i \in H_{a_t, p_t}$ satisfying $i \leq x_t$. We can compute $n_2$ in $O(1)$ time using $D_{a_t, p_t}$. Then $\text{rank}_{a_t}(x_t, C) = n_1 + n_2$. Since $C[x_t]$ is the rightmost occurrence of $a_j$ up to $C[i_j]$, $\text{rank}_{a_t}(i_j, C) = \text{rank}_{a_t}(x_t, C)$. The time needed to traverse the sequence $R'$ is $O(\sigma / \log^2 n)$ for all the queries. Other computations take $O(1)$ time per query. Hence the sequence of $\sigma$ queries on a chunk is answered in $O(\sigma + \sigma / \log^2 n)$ time.

**Global Sequence.** Now we consider the global sequence of queries $\text{rank}_{a_1}(i_1, B)$, $\ldots$, $\text{rank}_{a_m}(i_m, B)$. First we assign queries to chunks (e.g., by sorting all queries by $\lfloor i/\sigma \rfloor + 1$ using radix sort). We answer the batch of queries on the $j$-th chunk in $O(m_j + \sigma / \log^2 n)$ time where $m_j$ is the number of queries on the $j$-th chunk. Since $\sum m_j = m$, all $m$ queries are answered in $O(m + n / \log^2 n) = O(n)$ time. Now we know the rank $n_{j,2} = \text{rank}_{a_j}(i'_j, C)$, where $i'_j = i_j - \lfloor i/\sigma \rfloor \sigma$ is the relative position of $B[i_j]$ in its chunk $C$.

The binary sequences $M_a$ allows us reduce rank queries on $B$ to rank queries on a chunk $C$. All sequences $M_a$ contain $n + \lfloor n/\sigma \rfloor$ bits; hence they use $O(n)$ bits of space. We can compute the number of occurrences of $a$ in the first $j$ chunks in $O(1)$ time by answering one select query. Consider a query rank $\text{rank}_{a_j}(i_j, B)$ and suppose that $n_{j,2}$ is already known. We compute $n_{j,1}$, where $n_{j,1} = \text{select}_0([i_j/\sigma], M_a) - \lfloor i_j/\sigma \rfloor$ is the number of times $a_j$ occurs in the first $\lfloor i_j/\sigma \rfloor$ chunks. Then we compute $\text{rank}_{a_j}(i_j, B) = n_{j,1} + n_{j,2}$.

**Theorem 3.1.** We can keep a sequence $B[0..n-1]$ over an alphabet of size $\sigma$ in $O(n \log \sigma)$ bits of space so that a batch of $m$ rank queries can be answered in $O(m)$ time, where $\frac{n}{\log n} \leq m \leq n$.

The static data structure of Theorem 3.1 can be dynamized so that batched queries and batched insertions are supported. Our dynamic data structures supports a batch of $m$ queries in time $O(m)$ and a batch of $m$ insertions in amortized time $O(m)$ for any $m$ that satisfies $\frac{n}{\log \sigma} \leq m \leq n$. We describe the dynamic data structure in Sections A.2 and A.3.

**4 Building the Suffix Tree**

Belazzougui proved the following result [1]: if we are given the BWT $B$ of a text $T$ and if we can report all the distinct symbols in a range of $B$ in optimal time, then in $O(n)$ time we can: (i) enumerate all the suffix array intervals corresponding to internal nodes of the suffix tree and (ii) for every internal node list the labels of its children and their intervals. Further he showed that, if we can enumerate all the suffix tree intervals in $O(n)$ time, then we can build the suffix tree topology [39] in $O(n)$ time. The algorithms need only $O(n)$ additional bits of space. We refer to Lemmas 4 and 1 and their proofs in [1] for details.

In Section A.4 we show that a partial rank data structure can be built in $O(n)$ deterministic time. This can be used to build the desired structure that reports the distinct symbols in a range, in $O(n)$ time and using $O(n \log \log \sigma)$ bits. The details are given in Section A.5. Therefore, we obtain the following result.

**Lemma 4.1.** If we already constructed the BWT of a text $T$, then we can build the suffix tree topology in $O(n)$ time using $O(n \log \log \sigma)$ additional bits.

In Section 5 we show that the permuted LCP array of $T$ can be constructed in $O(n)$ time using $O(n \log \sigma)$ bits of space. Thus we obtain our main result on building compressed suffix trees.

**Theorem 4.1.** Given a string $T[0..n-1]$ over an alphabet of size $\sigma$, we can construct the compressed suffix tree of $T$ in $O(n)$ deterministic time using $O(n \log \sigma)$ additional bits.

**5 Constructing the Permuted LCP Array**

The permuted LCP array is defined as $\text{PLCP}[i] = j$ if and only if $SA[r^i] = i$ and the longest common prefix of $T[\text{SA}[r^i]]$ and $T[\text{SA}[r^i-1]]$ is of length $j$. In other words $\text{PLCP}[i]$ is the length of the longest common prefix of $T[i..]$ and the suffix that precedes it in the lexicographic ordering. In this section we show how the permuted LCP array $\text{PLCP}[0..n-1]$ can be built in linear time.

**Preliminaries.** For $i = 0, 1, \ldots, n$ let $\ell_i = \text{PLCP}[i]$. It is easy to observe that $\ell_i \leq \ell_{i+1} + 1$: if the longest common prefix of $T[i..]$ and $T[j..]$ is $q$, then the longest common prefix of $T[i+1..]$ and $T[j+1..]$ is at least $q - 1$. Let $\Delta = \Delta \log \log \sigma$ for $\Delta = \log \sigma n$. 

By the same argument $\ell_i \leq \ell_{i+\Delta'} + \Delta'$. To simplify the description we will further assume that $\ell_1 = 0$. It can also be shown that $\sum_{i=0}^{n-1} (\ell_i - \ell_{i-1}) = O(n)$.

We will denote by $B$ the BWT sequence of $T$; $\overline{B}$ denotes the BWT of the reversed text $\overline{T} = T[n-1]T[n-2] \ldots T[1]T[0]$. Let $p$ be a factor (substring) of $T$ and let $c$ be a character. The operation $\text{extendright}(p, c)$ computes the suffix interval of $pc$ in $B$ and the suffix interval of $\overline{pc}$ in $\overline{B}$ provided that the intervals of $p$ and $\overline{p}$ are known. The operation $\text{contractleft}(cp)$ computes the suffix intervals of $p$ and $\overline{p}$ provided that the suffix intervals of factors $cp$ and $\overline{cp}$ are known\(^2\). It was demonstrated [41, 4] that both operations can be supported by answering $O(1)$ rank queries on $B$ and $\overline{B}$.

Belazzougui [1] proposed the following algorithm for consecutive computing of $\ell_0$, $\ell_1$, $\ldots$, $\ell_n$. Suppose that $\ell_{i-1}$ is already known. We already know the rank $r_{i-1}$ of $T[i-1..i]$, the interval of $T[i-1..i + \ell_{i-1} - 1]$ in $B$, and the interval of $\overline{T}[i-1..i + \ell_{i-1} - 1]$ in $\overline{B}$. We compute the rank $r_i$ of $T[i..i]$.

If $r_{i-1}$ is known, we can compute $r_i$ in $O(1)$ time by answering one select query on $B$; see Section A.1. Then we find the interval $[r_s, r_c]$ of $T[i..i + \ell_{i-1} - 1]$ in $B$ and the interval $[r'_s, r'_c]$ of $\overline{T}[i..i + \ell_{i-1} - 1]$ in $\overline{B}$. These two intervals can be computed by operation $\text{contractleft}$. In the special case when $i = 0$ or $\ell_i = 0$, we set $[r, r_c] = \overline{[r_s, r_c]} = [0, n-1]$. Then for $j = 1, 2, \ldots,$ we find the intervals for $T[i..i + (\ell_{i-1} - 1) + j]$ and $\overline{T}[i..i + (\ell_{i-1} - 1) + j]$. Every following pair of intervals is found by operation $\text{extendright}$. We stop when the interval of $T[i..i + \ell_{i-1} - 1 + j]$ is $[r_s, j, r_{c,j}]$ such that $r_{s,j} = r_i$. For all $j'$, such that $0 \leq j' < j$, we have $r_{s,j'} < r_i$. It can be shown that $\ell_i = \ell_{i-1} + j - 1$; see the proof of [1, Lemma 2]. Once $\ell_i$ is computed, we increment $i$ and find the next $\ell_i$ in the same way. All $\ell_i$ are computed by $O(n)$ $\text{contractleft}$ and $\text{extendright}$ operations.

**Implementing $\text{contractleft}$ and $\text{extendright}$.** We create the succinct representation of the suffix tree topology both for $T$ and $\overline{T}$; they will be denoted by $\mathcal{T}$ and $\overline{\mathcal{T}}$ respectively. We keep both $B$ and $\overline{B}$ in the data structure that supports access in $O(1)$ time. We also store $B$ in the data structure that answers select queries in $O(1)$ time. The array $\text{Acc}$ keeps information about accumulated frequencies of symbols: $\text{Acc}[i]$ is the number of occurrences of all symbols $a \leq i - 1$ in $B$. Operation $\text{contractleft}$ is implemented as follows. Suppose that we know the interval $[i, j]$ for a factor $cp$ and the interval $[i', j']$ for the factor $\overline{cp}$. We can compute the interval $[i_1, j_1]$ of $p$ by finding $l = \text{select}_c(i - \text{Acc}[c], B)$ and $r = \text{select}_c(j - \text{Acc}[c], B)$. Then we find the lowest common ancestor $x$ of leaves $l$ and $r$ in the suffix tree $\mathcal{T}$. We set $i_1 = \text{leftmostleaf}(x)$ and $j_1 = \text{rightmostleaf}(x)$. Then we consider the number of distinct symbols in $B[i_1..j_1]$. If $c$ is the only symbol that occurs in $B[i_1..j_1]$, then all factors $p$ in $T$ are preceded by $c$. Hence all factors $\overline{p}$ in $\overline{T}$ are followed by $c$ and all occurrences of $p$ in $T$ are followed by $c$. Hence the interval of $pc$ in $B$ is $[i_1, j_1]$. Otherwise there is at least one other symbol besides $c$ that can follow $p$. Let $x$ denote the lowest common ancestor of leaves $i$ and $j$. If $y$ is the child of $x$ that is labeled with $c$, then the interval of $pc$ is $[i_1, j_1]$ where $i_1 = \text{leftmostleaf}(y)$ and $j_1 = \text{rightmostleaf}(y)$.

We can find the child $y$ of $x$ that is labeled with $c$ by answering rank and select queries on two additional sequences, $L$ and $D$. The sequence $L$ contains labels of children for all nodes of $\mathcal{T}$; labels are ordered by nodes and labels of the same node are ordered lexicographically. We encode the degrees of all nodes in a sequence $D = d_1d_20d_1d_21d_3 \ldots d_n$, where $d_i$ is the degree of the $i$-th node. We compute $v = \text{select}_0(x, D) - x$, $p_1 = \text{rank}_0(v, L)$, $p_2 = \text{select}_c(p_1 + 1, L)$, and $j = p_2 - v$. Then $y$ is the $j$-th child of $x$. The bottleneck of $\text{extendright}$ are the computations of $p_1$, $i'$, and $j_1'$ because we need $\Omega(\log \log \log n)$ time to answer a rank query on $L$ (resp. on $\overline{B}$); all other calculations can be executed in $O(1)$ time.

**Our Approach.** Our algorithm follows the technique of [1] that relies on operations $\text{extendright}$ and $\text{contractleft}$ for building the PLCP. We implement these two operations as described above; hence we will have to perform $\Theta(n)$ rank queries on sequences $L$ and $\overline{B}$. Our method creates large batches of queries; each query in a batch is answered in $O(1)$ time using Theorem 3.1.

During the pre-processing stage we create the machinery for supporting operations $\text{extendright}$ and $\text{contractleft}$. We compute the BWT $B$ of $T$ and the BWT $\overline{B}$ for the reverse text $\overline{T}$. We also construct the suffix tree topologies $\mathcal{T}$ and $\overline{\mathcal{T}}$. When $B$ is constructed,
we record the positions in \( B \) that correspond to suffixes \( T[i \cdot \Delta'..] \) for \( i = 0, \ldots, \lfloor n/\Delta' \rfloor \). PLCP construction is divided into three stages: first we compute the values of \( \ell_i \) for selected evenly spaced indices \( i, j = j \cdot \Delta' \) and \( j = 0, 1, \ldots, \lfloor n/\Delta' \rfloor \). We use a slow algorithm for computing lengths that takes \( O(\Delta') \) extra time for every \( \ell_i \). During the second stage we compute all remaining values of \( \ell_i \). We use the method from [1] during Stage 2. The key to a fast implementation is “parallel” computation. We divide all lengths into groups and assign each group of lengths to a job. At any time we process a list containing at least \( 2n/\log^2 n \) jobs. We answer rank queries in batches: when a job \( J_i \) must answer a slow rank query on \( L \) or \( B \), we pause \( J_i \) and add the rank query to the corresponding pool of queries. When a pool of queries on \( L \) or the pool of queries on \( B \) contains \( n/\log^2 n \) items, we answer the batch of queries in \( O(n/\log^2 n) \) time. The third stage starts when the number of jobs becomes smaller than \( 2n/\log^2 n \). All lengths that were not computed earlier are computed during Stage 3 using the slow algorithm. Stage 2 can be executed in \( O(n) \) time because rank queries are answered in \( O(1) \) time per query. Since the number of lengths that we compute during the first and the third stages is small, Stage 1 and Stage 3 also take \( O(n) \). A more detailed description follows.

**Stage 1.** Our algorithm starts by computing \( \ell_i \) for \( i = j \cdot \Delta' \) and \( j = 0, 1, \ldots, \lfloor n/\Delta' \rfloor \). Let \( j = 0 \) and \( f = j \Delta' \). We already know the rank \( r_f \) of \( S_f = T[j \Delta'..] \) in \( B \) (\( r_f \) was computed and recorded when \( B \) was constructed). We can also find the starting position \( j' \) of the suffix \( S' \) of rank \( r_f - 1 \) in \( B \). Since \( f' \) can be found by employing the function \( LF \) at most \( \Delta' \) times, we compute \( f' \) in \( O(\Delta') \) time; see Section A.1\(^3\).

When \( f \) and \( f' \) are known, we scan \( T[f..] \) and \( T[f'..] \) until the first symbol \( T[f + p_f] \neq T[f' + p_f] \) is found. By definition of \( \ell_f \), \( \ell_0 = p_f - 1 \). Suppose that \( \ell_s \Delta' \) for \( s = 0, \ldots, j - 1 \) are already computed and we have to compute \( \ell_f \) for \( f = j \Delta' \) and some \( j \geq 1 \). We already know the rank \( r_f \) of \( S_f \). We find \( f' \) such that the suffix \( T[f'..] \) is of rank \( r_f - 1 \) in \( O(\Delta') \). We showed above that \( \ell_f \geq \ell_{(j-1) \Delta'} - \Delta' \). Hence the first \( \alpha_f \) symbols in \( T[f..] \) and \( T[f'..] \) are equal, where \( \alpha_f = \max(0, \ell_{(j-1) \Delta'} - \Delta') \). We scan \( T[f + \alpha_f..] \) and \( T[f' + \alpha_f..] \) until the first symbol \( T[f + \alpha_f + p_f] \neq T[f' + \alpha_f + p_f] \) is found. By definition, \( \ell_f = \alpha_f + p_f \). Hence we compute \( \ell_f \) in \( O(\Delta' + p_f) \) time for \( f = j \Delta' \) and \( j = 1, \ldots, \lfloor n/\Delta' \rfloor \). It can be shown that \( \sum_f p_f = O(n) \). Hence the total time needed to compute all selected \( \ell_f \) is \( O(n/\Delta') \Delta' + \sum_f p_f = O(n) \). For every \( f = j \Delta' \) we also compute the interval of \( T[j \Delta'..j \Delta' + \ell_f] \) in \( B \).

![Figure 2: Computing lengths during Stage 2. Groups corresponding to paused jobs are shown shaded by slanted lines. Only selected groups are shown. The i-th job \( J_i \) is paused because we have to answer a rank query on \( B \); the job \( J_i \) is paused because we have to answer a rank query on \( L \). When \( Q_1 \) or \( Q_2 \) contain \( n/\log^2 n \) queries, we answer a batch of rank queries contained in \( Q_1 \) or \( Q_2 \).]

A faster computation is possible, but we do not need it here.

---

\(^3\)A faster computation is possible, but we do not need it here.
extendright are implemented as described above. We must answer two rank queries on $B$ and one rank query on $L$ for every extendright. Ignoring the time for these three rank queries, extendright takes constant time. Rank queries on $B$ and $L$ are answered in batches, so that each rank query takes $O(1)$ time. Hence every operation extendright needs $O(1)$ time. The job $J_i$ needs $O(\ell_{i\Delta'+1} - \ell_{i\Delta'} + j)$ time to compute $\ell_{i\Delta'+1}, \ell_{i\Delta'+2}, \ldots, \ell_{i\Delta'+j}$. All $J_i$ are executed in $O(n)$ time.

Stage 3. “Parallel processing” of jobs terminates when the number of jobs in the pool becomes smaller than $2n/\log^2 n$. Since every job computes $\Delta'$ values of $\ell_i$, there are at most $2n(\log \log \sigma/(\log n \log \sigma)) < 2n/\log n$ unknown values of $\ell_i$ at this point. We then switch to the method of Stage 1 to compute the values of unknown $\ell_i$. All remaining $\ell_i$ are sorted by $i$ and processed in order of increasing $i$. For every unknown $\ell_i$ we compute the rank $r$ of $T[i\ldots]$ in $B$. For the suffix $S'$ of rank $r-1$ we find its starting position $f'$ in $T$, $S' = T[f'\ldots]$. Then we scan $T[f'+\ell_{i-1} - 1\ldots]$ and $T[i+\ell_{i-1} - 1\ldots]$ until the first symbol $T[f'+\ell_{i-1} + j - 1] \neq T[f+\ell_{i-1} + j - 1]$ is found. We set $\ell_i = \ell_{i-1} + j - 2$ and continue with the next unknown $\ell_i$. We spend $O(\Delta' + \ell_i)$ additional time for every remaining $\ell_i$; hence the total time needed to compute all $\ell_i$ is $O(n + (n/\log n)\Delta') = O(n)$.

Every job during Stage 2 uses $O(\log n)$ bits of workspace. The total number of jobs in the job list does not exceed $n/\Delta'$. The total number of queries stored at any time in lists $Q_i$ and $Q_b$ does not exceed $n/\log^2 n$. Hence our algorithm uses $O(n \log \sigma)$ bits of workspace.

Lemma 5.1. If the BWT of a string $T$ and the suffix tree topology for $T$ are already known, then we can compute the permuted LCP array in $O(n)$ time and $O(n \log \sigma)$ bits.

6 Conclusions

We have shown that the Burrows-Wheeler Transform (BWT), the Compressed Suffix Array (CSA), and the Compressed Suffix Tree (CST) can be built in deterministic $O(n)$ time by an algorithm that requires $O(n \log \sigma)$ bits of working space. Belazzougui independently developed an alternative solution, which also builds within the same time and space the simpler part of our structures, that is, the BWT and the CSA, but not the CST. His solution, that uses different techniques, is described in the updated version of his ArXiV report [2] that extends his conference paper [1].

Our results have many interesting applications. For example, we can now construct an FM-index [13, 14] in $O(n)$ deterministic time using $O(n \log \sigma)$ bits. Previous results need $O(n \log \log \sigma)$ time or rely on randomization [21, 1]. Furthermore Theorem A.4.1 enables us to support the function $LF$ in $O(1)$ time on an FM-index. In the extended version of this paper [29] we also describe a new index based on these ideas.

Another application is that we can now compute the Lempel-Ziv 77 and 78 parsings [27, 45, 46] of a string $T[0..n-1]$ in deterministic linear time using $O(n \log \sigma)$ bits: Köppl and Sadakane [25] recently showed that, if one has a compressed suffix tree on $T$, then they need only $O(n)$ additional (deterministic) time and $O(\log n)$ bits to produce the parsing, where $z$ is the resulting number of phrases. Since $z \leq n/\log_2 n$, the space is $O(n \log \sigma)$ bits. With the suffix tree, they need to compute in constant time any $\Psi(i)$ and to move in constant time from a suffix tree node to its $i$-th child. The former is easily supported as the inverse of the $LF$ function using constant-time select queries on $B$ [17]; the latter is also easily obtained with current topology representations using parentheses [35].

Yet another immediate application of our algorithm are index data structures for dynamic document collections. If we use our compressed index, described in the extended version of this paper [29], and apply Transformation 2 from [31], then we obtain an index data structure for a dynamic collection of documents that uses $nH_k + o(n \log \sigma) + O(n \log \sigma)$ bits where $H_k$ is the $k$-th order entropy and $s$ is a parameter. This index can count how many times a query pattern $P$ occurs in a collection in $O(|P| \log \log n + \log \log \sigma \log \log n)$ time; every occurrence can be then reported in time $O(s)$. An insertion or a deletion of some document $T_s$ is supported in $O(|T_s| \log^2 n)$ and $O(|T_s| \log^2 n + s)$ deterministic time respectively.

We believe that our technique can also improve upon some of the recently presented results on bidirectional FM-indices [41, 4] and other scenarios where compressed suffix trees are used [5].

Acknowledgment. The authors wish to thank an anonymous reviewer of this paper for careful reading and helpful comments.

References


A.1 Preliminaries

Rank and Select Queries

The following two kinds of queries play a crucial role in compressed indexes and other succinct data structures. Consider a sequence $B[0..n-1]$ of symbols over an alphabet of size $\sigma$. The rank query $\text{rank}_a(i, B)$ counts how many times $a$ occurs among the first $i+1$ symbols in $B$, $\text{rank}_a(i, B) = |\{ j | B[j] = a \text{ and } 0 \leq j < i \}|$. The select query $\text{select}_a(i, B)$ finds the position in $B$ where $a$ occurs for the $i$-th time, $\text{select}_a(i, B) = j$ where $j$ is such that $B[j] = a$ and $\text{rank}_a(j, B) = i$. The third kind of query is the access query, $\text{access}(i, B)$, which returns the $(i+1)$-th symbol in $B$, $B[i]$. If insertions and deletions of symbols in $B$ must be supported, then both kinds of queries require $\Omega(\log n / \log \log n)$ time [15]. If the sequence $B$ is static, then we can answer select queries in $O(1)$ time and the cost of rank queries is reduced to $\Theta(\log \sigma / \log \log n)$ [7].

One important special case of rank queries is the partial rank query, $\text{rank}_{B[i]}(i, B)$. Thus a partial rank query asks how many times $B[i]$ occurred in $B[0..i]$. Unlike general rank queries, partial rank queries can be answered in $O(1)$ time [7]. In Section A.4 we describe a data structure for partial rank queries that can be constructed in $O(n)$ deterministic time. Better results can be achieved in the special case when the alphabet size is $\sigma = \log^{O(1)} n$; in this case we can represent $B$ so that rank, select, and access queries are answered in $O(1)$ time [14].

Suffix Tree and Suffix Array

A suffix tree for a string $T[0..n-1]$ is a compacted tree on the suffixes of $T$. The suffix array is an array $SA[0..n-1]$ such that $SA[i] = j$ if and only if $T[j..]$ is the $(i+1)$-th lexicographically smallest suffix of $T$. All occurrences of a substring $p$ in $T$ correspond to suffixes of $T$ that start with $p$; these suffixes occupy a contiguous interval in the suffix array $SA$.

Compressed Suffix Array

A compressed suffix array (CSA) is a compact data structure that provides the same functionality as the suffix array. The main component of CSA is the function $\Psi$, defined by the equality $SA[\Psi(i+1)] = (SA[i]+1) \mod n$. It is possible to regenerate the suffix array from $\Psi$. We refer to [32] and references therein for a detailed description of CSA and for trade-offs between space usage and access time.

Burrows-Wheeler Transform and FM-index.

The Burrows-Wheeler Transform (BWT) of a string $T$ is obtained by sorting all possible rotations of $T$ and writing the last symbol of every rotation (in sorted order). The BWT is related to the suffix array as follows: $BWT[i] = T[(SA[i]-1) \mod n]$. Hence, we can build the BWT by sorting the suffixes and writing the symbols that precede the suffixes in lexicographical order. This method is used in Section 2.

The FM-index uses the BWT for efficient searching in $T$. It consists of the following three main components:

- The BWT of $T$.
- The array $\text{Acc}[0..\sigma-1]$ where $\text{Acc}[i]$ holds the total number of symbols $a \leq i-1$ in $T$ (or equivalently, the total number of symbols $a \leq i-1$ in $B$).
- A sampled array $\text{SAM}_b$ for a sampling factor $b$: $\text{SAM}_b$ contains values of $SA[i]$ if and only if $SA[i] \mod b = 0$ or $SA[i] = n-1$.

---

If we aim to use $n \log \sigma + o(n \log \sigma)$ bits, then either select or access must cost $\omega(1)$. If, however, $(1 + \epsilon)n \log \sigma$ bits are available, for any constant $\epsilon > 0$, then we can support both queries in $O(1)$ time.
The search for a substring $P$ of length $m$ is performed backwards: for $i = m - 1, m - 2, \ldots$, we identify the interval of $p[i..m]$ in the BWT. Let $B$ denote the BWT of $T$. Suppose we know the interval $B[i_1..j_1]$ that corresponds to $p[i+1..m-1]$. Then the interval $B[i_2..j_2]$ that corresponds to $p[i..m-1]$ is computed as $i_2 = \text{rank}_B(i_1 - 1) + \text{Acc}[c]$ and $j_2 = \text{rank}_B(j_1 - 1) + \text{Acc}[c] - 1$, where $c = P[i]$. Thus the interval of $p$ is found by answering $2m$ rank queries. We observe that the interval of $p$ in $B$ is exactly the same as the interval of $p$ in the suffix array $SA$.

Another important component of an FM-index is the function $LF$, defined as follows: if $SA[j] = i + 1$, then $SA[LF(j)] = i$. $LF$ can be computed by answering rank queries on $B$. Using $LF$ we can find the starting position of the $r$-th smallest suffix, $SA[r]$, in $O(b)$ applications of $LF$, where $b$ is the sampling factor; we refer to [32] for details. It is also possible to compute the function $\Psi$ by using select queries the BWT [26]. Therefore the BWT can be viewed as a variant of the CSA. Using $\Psi$ we can can consecutively obtain positions of suffixes $T[i..]$ in the suffix array: Let $r_i$ denote the position of $T[i..]$ in $SA$. Since $T[n-1..] = \$\text{ is the smallest suffix, } r_0 = \Psi(0)$. For $i \geq 1$, $r_i = \Psi(r_{i-1})$ by definition of $\Psi$. Hence we can consecutively compute each $r_i$ in $O(1)$ time if we have constant-time select queries on the BWT.

**Compressed Suffix Tree.** A compressed suffix tree consists of the following components:

- The compressed suffix array of $T$. We can use the FM-index as an implementation.
- The suffix tree topology. This component can be stored in $4n + o(n)$ bits [39].
- The permuted LCP array, or PLCP. The longest common prefix array $LCP$ is defined as follows: $LCP[r] = j$ if and only if the longest common prefix between the suffixes of rank $r$ and $r - 1$ is of length $j$. The permuted LCP array is defined as follows: $PLCP[r] = j$ if and only if the rank of $T[i..]$ is $r$ and $LCP[r] = j$. A careful implementation of $PLCP$ occupies $2n + o(n)$ bits [39].

### A.2 Monotone List Labelling with Batched Updates

A direct attempt to dynamize the data structure of Section 3 encounters one significant difficulty. An insertion of a new symbol $a$ into a chunk $C$ changes the positions of all the symbols that follow it. Since symbols are stored in pairs $(a, i)$ grouped by symbol, even a single insertion into $C$ can lead to a linear number of updates. Thus it appears that we cannot support the batch of updates on $C$ in less than $\Theta(|C|)$ time. In order to overcome this difficulty we employ a monotone labeling method and assign labels to positions of symbols. Every position $i$ in the chunk is assigned an integer label $lab(i)$ satisfying $0 \leq lab(i) \leq \sigma \cdot n^{O(1)}$ and $lab(i_1) < lab(i_2)$ if and only if $i_1 < i_2$. Instead of pairs $(a, i)$ the sequence $R$ will contain pairs $(a, lab(i))$.

When a new element is inserted, we have to change the labels of some other elements in order to maintain the monotonicity of the labeling. Existing labeling schemes [44, 10, 11] require $O(n \log n)$ changes of labels after every insertion. In our case, however, we have to process large batches of insertions. We can also assume that at most $\log n$ batches need to be processed. In our scenario $O(1)$ amortized modifications per insertion can be achieved, as shown below.

In this section we denote by $C$ an ordered set that contains between $\sigma$ and $2\sigma$ elements. Let $x_1 \leq x_2 \leq \ldots \leq x_\ell$ denote the elements of $C$. Initially we assign the label $lab(x_i) = i \cdot d$ to the $i$-th smallest element $x_i$, where $d = 4n$. We associate an interval $[\text{lab}(x_i), \text{lab}(x_{i+1}) - 1]$ with $x_i$. Thus initially the interval of $x_i$ is $[id, (i+1)d-1]$. We assume that $C$ also contains a dummy element $x_0 = -\infty$ and $\text{lab}(-\infty) = 0$. Thus all labels are non-negative integers bounded by $O(\sigma \cdot n)$.

Suppose that the $k$-th batch of insertions consists of $m$ new elements $y_1 \leq y_2 \leq \ldots \leq y_m$. Since at most $\log n$ batches of insertions must be supported, $1 \leq k \leq \log n$. We say that an element $y_i$ is in an interval $I = [\text{lab}(x_s), \text{lab}(x_e)]$ if $x_s < y_i < x_e$. We denote by $new(I)$ the number of inserted elements in $I$. The parameter $\rho(I)$ for an interval $I$ is defined as the ratio of old and new elements in $I = [\text{lab}(x_s), \text{lab}(x_e)]$, $\rho(I) = \frac{\text{lab}(x_e) - \text{lab}(x_s)}{new(I)}$. We identify the set of non-overlapping intervals $I_1, \ldots, I_r$ such that every new element $y_i$ is in some interval $I_j$, and $1 \leq \rho(I_j) \leq 2$ for all $j$, $1 \leq j \leq r$. (This is always possible if $m \leq |C|$; otherwise we simply merge the insertions with $C$ in $O(|C| + m) = O(m)$ time and restart all the labels.) We can find $I_1, \ldots, I_r$ in $O(m)$ time. For every $I_j$, $1 \leq j \leq r$, we evenly distribute the labels of old and new elements in the interval $I'_j \subseteq I_j$. Suppose that $f$ new elements $y_{p\cdot f}, \ldots, y_{p\cdot f+j-1}$ are inserted into interval $I_j = [\text{lab}(x_s), \text{lab}(x_e)]$ so that there now are $v = f + (e-s) + 1$ elements in this interval. We assign the label $\text{lab}(x_s) + d_j \cdot (i-1)$ to the $i$-th smallest element in $I_j$ where $d_j = \frac{\text{lab}(x_s) - \text{lab}(x_e)}{f}$. By our choice of $I_j$, $f \leq e - s + 1$ and the number of elements in $I_j$ increased at most by twofold. Hence the minimal distance between two consecutive labels does not decrease by more than a factor of 2 after insertion of new elements into $I_j$. We inserted $f$ new elements into $I_j$ and changed the labels of at most $2f$ old elements.
Hence the amortized number of labels that we must change after every insertion is $O(1)$. The initial distance between labels is $d = 4n$ and this distance becomes at most two times smaller after every batch of insertions. Hence the distance between consecutive labels is an integer larger than 2 during the first $\log n$ batches.

One remaining problem with our scheme is the large range of the labels. Since labels are integers bounded by $4|C|n$, we need $\Theta(\log \sigma + \log n)$ bits per label. To solve this problem, we will split the chunk $C$ into blocks and assign the same label to all the symbols in a block. A label assigned to the symbols in a block will be stored only once. Details are provided in Section A.3.

A.3 Batched Rank Queries and Insertions on a Sequence

In this section we describe a dynamic data structure that supports both batches of rank queries and batches of insertions. First we describe how queries and updates on a chunk $C$ are supported.

The linked list $L$ contains all the symbols of $C$ in the same order as they appear in $C$. Each node of $L$ stores a block of $\Theta(\log_\sigma n)$ symbols, containing at most $(1/4) \log_\sigma n$ of them. We will identify list nodes with the blocks they contain; however, the node storing block $b$ also stores the total number of symbols in all preceding blocks and a label lab$(b)$ for the block. Labels are assigned to blocks with the method described in Section A.2. The pointer to (the list node containing) block $b$ will be called $p_b$; these pointers use $O(\log \sigma)$ bits.

We also maintain a data structure that can answer rank queries on any block. The data structure for a block supports queries and insertions in $O(1)$ time using a look-up table: Since $\sigma \leq n^{1/4}$ and the block size is $(1/4) \log_\sigma n$, we can keep pre-computed answers to all rank queries for all possible blocks in a table $Tbl[0..n^{1/4} - 1][0..n^{1/4} - 1][0..\log_\sigma n - 1]$. The entry $Tbl[b][a][i]$ contains the answer to the query rank$_b(i, b)$ on a block $b$. Tbl contains $O(n^{1/2} \log_\sigma n) = o(n)$ entries and can be constructed in $o(n)$ time. Updates can be supported by a similar look-up table or by bit operations on the block $b$.

We also use sequences $R$ and $R'$, defined in Section 3, but we make the following modifications. For every occurrence $C[i] = a$ of a symbol $a$ in $C$, the sequence $R$ contains pair $(a, p_b)$, where $p_b$ is a pointer to the block $b$ of $L$ that contains $C[i]$. Pairs are sorted by symbol in increasing order, and pairs with the same symbol are sorted by their position in $C$. Unlike in Section 3, the chunk $C$ can be updated and we cannot maintain the exact position $i$ of $C[i]$ for all symbols in $C$; we only maintain the pointers $p_b$ in the pairs $(a, p_b) \in R$.

Note that we cannot use block pointers for searching in $L$ (or in $C$). Instead, block labels are monotonically increasing and lab$(b_1) < \text{lab}(b_2)$ if the block $b_2$ follows $b_1$ in $L$. Hence block labels will be used for searching and answering rank queries. Block labels lab$(b)$ use $\Theta(\log n)$ bits of space, so we store them only once with the list nodes $b$ and access them via the pointers $p_b$.

Groups $H_{a,j}$ are defined as in Section 3; each $H_{a,j}$ contains all the pairs of $R$ that are between two consecutive elements of $R'_a$ for some $a$. The data structure $D_{a,j}$ that permits searching in $H_{a,j}$ is defined as follows. Suppose that $H_{a,j}$ contains pairs $(a, p_{b_1}), \ldots, (a, p_{b_i})$. We then keep a Succinct SB-tree data structure [18] on lab$(b_1), \ldots, \text{lab}(b_j)$. This data structure requires $O(\log \log n)$ additional bits per label. For any integer $q$, it can find the largest block label lab$(b_i) < q$ in $O(1)$ time or count the number of blocks $b_j$ such that lab$(b_j) < q$ in $O(1)$ time (because our sets $H_{a,c}$ contain a logarithmic number of elements). The search procedure needs to access one block label, which we read from the corresponding block pointer.

**Queries.** Suppose that we want to answer queries rank$_a(i_1,C)$, rank$_a(i_2,C)$, $\ldots$, rank$_a(i_3,C)$ on a chunk $C$. We traverse the blocks of $L$ and find for every $i_j$ the label $l_j$ of the block $b_j$ that contains the $i_j$-th symbol, $l_j = \text{lab}(b_j)$. We also compute $r_{j,1} = \text{rank}_a(i_j, b_j)$ using Tbl, where $i_j$ is the relative position of the $i_j$-th symbol in $b_j$. Since we know the total number of symbols in all the blocks that precede $b_j$, we can compute $i_j$ in $O(1)$ time.

We then represent the queries by pairs $(a_j, l_j)$ and sort these pairs stably in increasing order of $a_j$. Then we traverse the list of query pairs $(a_j, l_j)$ and the sequence $R'$. For every query $(a_j, l_j)$ we find the rightmost pair $(a_j, p_j) \in R'$ satisfying lab$(b_j) < l_j$. Let $r_{j,2}$ denote the rank of $(a_j, p_j)$ in $R_{a_j}$, i.e., the number of pairs $(a_i, i) \in R$ preceding $(a_j, p_j)$. We keep this information for every pair in $R'$ using $O(\log \sigma)$ additional bits. Then we use the succinct SB-tree $D_{a,j,p_j}$, which contains information about the pairs in $H_{a,j,p_j}$ (i.e., the pairs in the group starting with $(a_j, p_j)$). The structure finds in constant time the largest lab$(b_g) \in D_{a,j,p_j}$ such that lab$(b_g) < l_j$, as well as the number $r_{j,3}$ of pairs from the beginning of $H_{a,j,p_j}$ up to the pair with label lab$(b_g)$. The answer to the $j$-th rank query is then rank$_a(i_j, C) = r_{j,1} + r_{j,2} + r_{j,3}$.

The total query time is then $O(\sigma/\log_\sigma n + t)$.

**Insertions.** Suppose that symbols $a_1, \ldots, a_t$ are to be inserted at positions $i_1, \ldots, i_t$, respectively. We traverse the list $L$ and identify the nodes where new symbols must be inserted. We simultaneously update the information about the number of preceding elements, for all nodes. All this is done in time $O(\sigma/\log_\sigma n + t)$. We also perform the insertions into the blocks. If, as
a result, some block contains more than \((1/4) \log_\sigma n\) symbols, we split it into an appropriate number of blocks, so that each block contains \(\Theta(\log_\sigma n)\) but at most \((1/4) \log_\sigma n\) symbols. Nodes for the new blocks are allocated\(^5\), linked to the list \(L\), and assigned appropriate labels using the method described in Section A.2. After \(t\) insertions, we create at most \(O(t/\log_\sigma n)\) new blocks (in the amortized sense, i.e., if we consider the insertions from the beginning). Each such new block \(b'\), coming from splitting an existing block \(b\), requires that we change all the corresponding pointers \(p_k\) from the pairs \((a_z, p_k)\) in \(R\) (and \(R'\)), so that they become \((a_z, p_k')\). To find those pairs efficiently, the list node holding \(\sigma(\log_\sigma n)\) pointers to those pairs (using \(O(\log \sigma)\) bits each); we can then update the required pointers in \(O(t)\) total time.

The new blocks also require creating their labels. Those \(O(t/\log_\sigma n)\) label insertions also trigger \(O(t/\log_\sigma n)\) changes of other labels, with the technique of Section A.2. If the label of a block \(b\) was changed, we visit all pairs \((a_z, p_k)\) in \(R\) that point to \(b\). Each such \((a_z, p_k)\) is kept in some group \(H_{a_z,k}\) and in some succinct SB-tree \(D_{a_z,k}\). We then delete the old label of \(b\) from \(D_{a_z,k}\) and insert the new modified label. The total number of updates is thus bounded by \(O(t)\). While not mentioned in the original paper \cite{18}, one can easily perform constant-time insertions and deletions of labels in a succinct SB-tree: The structure is a two-level B-tree of arity \(\sqrt{\log n}\) holding encoded Patricia trees on the bits of the keys, and storing at the leaves the positions of the keys in \(H_{a,r}\) using \(O(\log \log n)\) bits each. To insert or delete a label we follow the usual B-tree procedures. The insertion or deletion of a key in a B-tree node is done in constant time with a precomputed table that, in the same spirit of TB\(l\), yields the resulting Patricia tree if we delete or insert a certain node; this is possible because internal nodes store only \(O(\sqrt{\log n} \log \log n) = o(\log n)\) bits. Similarly, we can delete or insert a key at the leaves of the tree.

Apart from handling the block overflows, we must insert in \(R\) the pairs corresponding to the new \(t\) symbols we are actually inserting. We perform \(t\) rank queries \(\text{rank}_{a_z}(i_1, C), \ldots, \text{rank}_{a_z}(i_t, C)\), just as described above, and sort the symbols to insert by those ranks using radix sort. We then traverse \(R'\) and identify the groups \(H_{a_1,j_1}, \ldots, H_{a_t,j_t}\) where new symbols must be inserted; the counters of preceding pairs for the pairs in \(R'\) is easily updated in this way. We allocate the pairs \((a_{k}, p_{b_k})\) that will belong to \(H_{a_{k},j_{k}}\) and insert the labels \(\text{lab}(b_k)\) in the corresponding data structures \(D_{a_{k},j_{k}}\), for all \(1 \leq k \leq \tau\). If some groups \(H_{a_{i},j_{i}}\) become larger than permitted, we split them as necessary and insert the corresponding pairs in \(R'\). We can answer the rank queries, traverse \(R\), and update the groups \(H_{a_{k},j_{k}}\) all in \(O(\sigma/\log_\sigma n + t)\) time.

**Global Sequence.** In addition to chunk data structures, we keep a static bitvector \(M_a = 1^{d_1} \ldots 1^{d_t}\) for every symbol \(a\); \(d_i\) denotes the number of times \(a\) occurs in the \(i\)-th chunk.

Given a global sequence of \(m \geq n/\log_\sigma n\) queries, \(\text{rank}_{a_1}(i_1, B), \ldots, \text{rank}_{a_m}(i_m, B)\) on \(B\), we can assign them to chunks in \(O(m)\) time. Then we answer queries on chunks as shown above. If \(m_j\) queries are asked on chunk \(C_j\), then these queries are processed in \(O(m_j + \sigma/\log_\sigma n)\) time. Hence all queries on all chunks are answered in \(O(m + n/\log_\sigma n) = O(m)\) time. We can answer a query \(\text{rank}_{a_j}(i_k, B)\) by answering a rank query on the chunk that contains \(B[i_k]\) and \(O(1)\) queries on the sequence \(M_{a_k}\) \cite{17}. Queries on \(M_{a_k}\) are supported in \(O(1)\) time because the bitvector is static. Hence the total time to answer \(m\) queries on \(B\) is \(O(m)\).

When a batch of symbols is inserted, we update the corresponding chunks as described above. If some chunk contains more than \(4\sigma\) symbols, we split it into several chunks of size \(\Theta(\sigma)\) using standard techniques. Finally we update the global sequences \(M_a\), both because of the insertions and due to the possible chunk splits. We simply rebuild the bitvectors \(M_a\) from scratch; this is easily done in \(O(n_a/\log n)\) time, where \(n_a\) is the number of bits in \(M_a\); see e.g. \cite{30}. This adds up to \(O(m/\log n)\) time.

Hence the total amortized cost for a batch of \(m \geq n/\Delta\) insertions is \(O(m)\).

**Theorem A.3.1.** We can keep a sequence \(B[0..n-1]\) over an alphabet of size \(\sigma\) in \(O(n \log \sigma)\) bits of space so that a batch of \(m\) rank queries can be answered in \(O(m)\) time and a batch of \(m\) insertions is supported in \(O(m)\) amortized time, for \(\frac{n}{\log_\sigma n} \leq m \leq n\).

**A.4 Sequences with Partial Rank Operation**

If \(\sigma = \log^{O(1)} n\), then we can keep a sequence \(S\) in \(O(n \log \sigma)\) bits so that select and rank queries (including partial rank queries) are answered in constant time \cite{14}.

In the remaining part of this section we will assume that \(\sigma \geq \log^3 n\).

**Lemma A.4.1.** Let \(\sigma \leq m \leq n\). We can support partial rank queries on a sequence \(C[0..m-1]\) over an alphabet of size \(\sigma\) in time \(O(1)\). The data structure needs \(O(m \log \log m)\) additional bits and can be constructed in \(O(m)\) deterministic time.

**Proof.** Our method employs the idea of buckets intro-
duced in [3]. Our structure does not use monotone perfect hashing, however. Let $I_a$ denote the set of positions where a symbol $a$ occurs in $C$, i.e., $I_a$ contains all integers $i$ satisfying $C[i] = a$. If $I_a$ contains more than $2 \log^2 m$ integers, we divide $I_a$ into buckets $B_{a,s}$ of size $\log^2 m$. Let $p_{a,s}$ denote the longest common prefix of all integers (seen as bit strings) in the bucket $B_{a,s}$ and let $l_{a,s}$ denote the length of $p_{a,s}$.

For every element $C[i]$ in the sequence we keep the value of $l_{C[i],t}$, where $B_{C[i],t}$ is the bucket containing $i$. If $l_{C[i],t}$ was not divided into buckets, we assume $l_{C[i],t} = \text{null}$, a dummy value. We will show below how the index $t$ of $B_{C[i],t}$ can be identified if $l_{C[i],t}$ is known. For every symbol $C[i]$ we also keep the rank $r$ of $i$ in its bucket $B_{C[i],t}$. That is, for every $C[i]$ we store the value of $r$ such that $i$ is the $r$-th smallest element in its bucket $B_{C[i],t}$. Both $l_{C[i],t}$ and $r$ can be stored in $O(\log m)$ bits. The partial rank of $C[i]$ in $C$ can be computed from $t$ and $r$.

It remains to describe how the index $t$ of the bucket containing $C[i]$ can be found. Our method uses $o(m)$ additional bits. First we observe that $p_{a,i} \neq p_{a,j}$ for any fixed $a$ and $i \neq j$; see [3] for a proof. Let $T_w$ denote the full binary trie on the interval $[0..m-1]$. Nodes of $T_w$ correspond to all possible bit prefixes of integers $0, \ldots, m-1$. We say that a bucket $B_{a,j}$ is assigned to a node $u \in T_w$ if $p_{a,j}$ corresponds to the node $u$. Thus many different buckets can be assigned to the same node $u$. But for any symbol $a$ at most one bucket $B_{a,k}$ is assigned to $u$. If a bucket is assigned to a node $u$, then there are at least $\log^2 m$ leaves below $u$. Hence buckets can be assigned to nodes of height at least $2 \log \log m$; such nodes will be further called bucket nodes. We store all buckets assigned to bucket nodes of $T_w$ using the structure described below.

We order the nodes $u$ level-by-level starting at the top of the tree. Let $m_j$ denote the number of buckets assigned to $u_j$. The data structure $G_j$ contains all symbols $a$ such that some bucket $B_{a,k}$ is assigned to $u_j$. For every symbol $a$ in $G_j$ we can find in $O(1)$ time the index $k_a$ of the bucket $B_{a,k}$ that is assigned to $u_j$. We implement $G_j$ as deterministic dictionaries of Hagerup et al. [20]. $G_j$ uses $O(m_j \log \log m)$ bits and can be constructed in $O(m_j \log \log m)$ time. We store $G_j$ only for bucket nodes $u_j$ such that $m_j > 0$. We also keep an array $W[1..m]$, whose entries correspond to bucket nodes of $T_w$: $W[j]$ contains a pointer to $G_j$ or null if $G_j$ does not exist.

Using $W$ and $G_j$ we can answer a partial rank query $\text{rank}_{G_j}(i,C)$. Let $C[i] = a$. Although the bucket $B_{a,t}$ containing $i$ is not known, we know the length $l_{a,t}$ of the prefix $p_{a,t}$. Hence $p_{a,t}$ can be computed by extracting the first $l_{a,t}$ bits of $i$. We can then find the index $j$ of the node $u_j$ that corresponds to $p_{a,t}$, $j = (2^{l_{a,t}} - 1) + p_{a,t}$. Finally the index $t$ of the bucket $B_{a,t}$ is computed as $t = G_j[a]$.

A data structure $G_j$ consumes $O(m_j \log m)$ bits. Since $\sum_j m_j \leq \frac{m}{\log^2 m}$, all $G_j$ use $O(m/\log m)$ bits of space. The array $W$ also uses $O(m/\log m)$ bits. Hence our data structure uses $O(\log \log m)$ additional bits per symbol.

**Theorem A.4.1.** We can support partial rank queries on a sequence $B$ using $O(n \log \log \sigma)$ additional bits. The underlying data structure can be constructed in $O(n)$ deterministic time.

**Proof.** We divide the sequence $B$ into chunks of size $\sigma$ (except for the last chunk that contains $n - (\lfloor n/\sigma \rfloor \sigma)$ symbols). Global sequences $M_a$ are defined in the same way as in Section 3. A partial rank query on $B$ can be answered by a partial rank query on a chunk and two queries on $M_a$.

### A.5 Reporting All Symbols in a Range

We prove the following lemma in this section.

**Lemma A.5.1.** Given a sequence $B[0..n-1]$ over an alphabet $\sigma$, we can build in $O(n)$ time a data structure that uses $O(n \log \log \sigma)$ additional bits and answers the following queries: for any range $[i,j]$, report those distinct symbols that occur in $B[i..j]$ in $O(\text{occ})$ time, and for every reported symbol $a$, give its frequency in $B[i..j]$ and its frequency in $B[0..i-1]$.

The proof is the same as that of Lemma 3 in [1], but we use the result of Theorem A.4.1 to answer partial rank queries. This allows us to construct the data structure in $O(n)$ deterministic time (while the data structure in [1] achieves the same query time, but the construction algorithm requires randomization). For completeness we sketch the proof below.

Augmenting $B$ with $O(n)$ additional bits, we can report all distinct symbols occurring in $B[i..j]$ in $O(\text{occ})$ time using the idea originally introduced by Sadakane [40]. For every reported symbol we can find in $O(1)$ time its leftmost and rightmost occurrences in $B[i..j]$. Suppose $i_a$ and $j_a$ are the leftmost and rightmost occurrences of $a$ in $B[i..j]$. Then the frequencies of $a$ in $B[i..j]$ and $B[0..i-1]$ can be computed as $\text{rank}_a(j_a, B) - \text{rank}_a(i_a, B) + 1$ and $\text{rank}_a(i_a, B) - 1$ respectively. Since $\text{rank}_a(i_a, B)$ and $\text{rank}_a(j_a, B)$ are partial rank queries, they are answered in $O(1)$ time. The data structure that reports the leftmost and the rightmost occurrences can be constructed in $O(n)$ time. Details and references can be found in [8]. Partial rank
queries are answered by the data structure of Theorem A.4.1. Hence the data structure of Lemma A.5.1 can be built in $O(n)$ deterministic time. We can also use the data structure of Lemma A.4.1 to determine whether the range $B[i..j]$ contains only one distinct symbol in $O(1)$ time by using the following observation. If $B[i..j]$ contains only one symbol, then $B[i] = B[j]$ and $\text{rank}_g[i](j, B) - \text{rank}_g[i](i, B) = j - i + 1$. Hence we can find out whether $B[i..j]$ contains exactly one symbol in $O(1)$ time by answering two partial rank queries. This observation will be helpful in Section 5.

A.6 Computing the Intervals

The algorithm for constructing PLCP, described in Section 5, requires that we compute the intervals of $T[j\Delta'.j\Delta' + \ell_i]$ and $T[j\Delta'.j\Delta' + \ell_i]$ for $i = j\Delta'$ and $j = 0, 1, \ldots, n/\Delta'$. We will show in this section how all necessary intervals can be computed in linear time when $\ell_i$ for $i = j\Delta'$ are known. Our algorithm uses the suffix tree topology. We construct some additional data structures and pointers for selected nodes of the suffix tree $T$. First, we will describe auxiliary data structures on $T$. Then we show how these structures can be used to find all needed intervals in linear time.

Marking Nodes in a Tree. We use the marking scheme described in [33]. Let $d = \log n$. A node $u$ of $T$ is heavy if it has at least $d$ leaf descendants and light otherwise. We say that a heavy node $u$ is a special or marked node if $u$ has at least two heavy children. If a non-special heavy node $u$ has more than $d$ children and among them is one heavy child, then we keep the index of the heavy child in $u$.

We keep all children of a node $u$ in the data structure $F_u$, so that the child of $u$ that is labeled by a symbol $a$ can be found efficiently. If $u$ has at most $d + 1$ children, then $F_u$ is implemented as a fusion tree [16]; we can find the child of $u$ labeled by any symbol $a$ in $O(1)$ time. If $u$ has more than $d + 1$ children, then $F_u$ is implemented as the van Emde Boas data structure and we can find the labeled child in $O(\log \log \sigma)$ time. If the node $u$ is special, we keep labels of its heavy children in the data structure $D_u$. $D_u$ is implemented as a dictionary data structure [20] so that we can find any heavy child of a special node in $O(1)$ time. We will say that a node $u$ is difficult if $u$ is light but the parent of $u$ is heavy. We can quickly navigate from a node $u \in T$ to its child $u_i$ unless the node $u_i$ is difficult.

**Proposition A.6.1.** We can find the child $u_i$ of $u$ that is labeled with a symbol $a$ in $O(1)$ time unless the node $u_i$ is difficult. If $u_i$ is difficult, we can find $u_i$ in $O(\log \log \sigma)$ time.

**Proof.** Suppose that $u_i$ is heavy. If $u$ is special, we can find $u_i$ in $O(1)$ time using $D_u$. If $u$ is not special and it has at most $d + 1$ children, then we find $u_i$ in $O(1)$ time using $F_u$. If $u$ is not special and it has more than $d + 1$ children, then $u_i$ is the only heavy child of $u$ and its index $i$ is stored with the node $u$. Suppose that $u_i$ is light and $u$ is also light. Then $u$ has at most $d$ children and we can find $u_i$ in $O(1)$ time using $F_u$. If $u$ is heavy and $u_i$ is light, then $u_i$ is a difficult node. In this case we can find the index $i$ of $u_i$ in $O(\log \log \sigma)$ time using $F_u$.

**Proposition A.6.2.** Any path from a node $u$ to its descendant $v$ contains at most one difficult node.

**Proof.** Suppose that a node $u$ is a heavy node and its descendant $v$ is a light node. Let $u'$ denote the first light node on the path from $u$ to $v$. Then all descendants of $u'$ are light nodes and $u'$ is the only difficult node on the path from $u$ to $v$. If $u$ is light or $v$ is heavy, then there are apparently no difficult nodes between $u$ and $v$.

Weiner Links. A Weiner link (or w-link) $\text{wlink}(v, c)$ connects a node $v$ of the suffix tree $T$ labeled by the path $p$ to the node $u$, such that $u$ is the locus of $cp$. If $\text{wlink}(v, c) = u$ we will say that $u$ is the target node and $v$ is the source of $\text{wlink}(v, c)$ and $c$ is the label of $\text{wlink}(v, c)$. If the target node $u$ is labeled by $cp$, we say that the w-link is explicit. If $u$ is labeled by some path $cp'$, such that $cp'$ is a proper prefix of $cp'$, then the Weiner link is implicit. We classify Weiner links using the same technique that was applied to nodes of the suffix tree above. Weiner links that share the same source node are called sibling links. A Weiner link from $v$ to $u$ is heavy if the node $u$ has at least $d$ leaf descendants and light otherwise. A node $v$ is $w$-special iff there are at least two heavy w-links connecting $v$ and some other nodes. For every special node $v$ the dictionary $D_v'$ contains the labels $c$ of all heavy w-links $\text{wlink}(v, c)$. For every $c$ such that $\text{wlink}(v, c)$ is heavy, we also keep the target node $u = \text{wlink}(v, c)$. $D_v'$ is implemented as in [20] so that queries are answered in $O(1)$ time. Suppose that $v$ is the source node of at least $d + 1$ w-links, but $u = \text{wlink}(v, c)$ is the only heavy link that starts at $v$.

In this case we say that $\text{wlink}(v, c)$ is unique and we store the index of $u$ and the symbol $c$ in $v$. Summing up, we store only heavy w-links that start in a $w$-special node or unique w-links. All other w-links are not stored explicitly: if they are needed, we compute them using additional data structures that will be described below.

Let $B$ denote the BWT of $T$. We split $B$ into intervals $G_j$ of size $4d^2$. For every $G_j$ we keep the dictionary $A_j$ of symbols that occur in $G_j$. For each symbol $a$ that occurs in $G_j$, the data structure $G_j.a$ contains all positions of $a$ in $G_j$. Using $A_j$, we can
Proposition A.6.3. The total number of heavy w-links that start in w-special nodes is $O(n/d)$.

Proof. Suppose that $u$ is a w-special node and let $p$ be the label of $u$. Let $c_1, \ldots, c_s$ denote the labels of heavy w-links with source node $u$. This means that each $c_ip, c_2p, \ldots, c_sp$ occurs at least $d$ times in $T$. Consider the suffix tree $\overline{\pi}$ of the reverse text $\pi$. $\overline{\pi}$ contains the node $\overline{u}$ that is labeled with $\overline{p}$. The node $\overline{u}$ has (at least) $s$ children $\overline{u}_1, \ldots, \overline{u}_s$. The edge connecting $\overline{u}$ and $\overline{u}_i$ is a string that starts with $c_i$. In other words each $\overline{u}_i$ is the locus of $\overline{pc}_i$. Since $c_ip$ occurs at least $d$ times in $T$, $\overline{pc}_i$ occurs at least $d$ times in $\overline{\pi}$. Hence each $\overline{u}_i$ has at least $d$ descendants. Thus every w-special node in $T$ correspond to a special node in $\overline{\pi}$ and every heavy w-link outgoing from a w-special node corresponds to some heavy child of a special node in $\overline{\pi}$. Since the number of heavy children of special nodes in a suffix tree is $O(n/d)$, the number of heavy w-links starting in a w-special node is also $O(n/d)$.

Proposition A.6.4. The total number of unique w-links is $O(n/d)$.

Proof. A Weiner link $\text{wlink}(v,a)$ is unique only if $\text{wlink}(v,a)$ is heavy, all other w-links outgoing from $v$ are light, and there are at least $d$ light outgoing w-links from $v$. Hence there are at least $d$ w-links for every explicitly stored target node of a unique Weiner link.

We say that $\text{wlink}(v,a)$ is difficult if its target node $u = \text{wlink}(v,a)$ is light and its source node $v$ is heavy.

Proposition A.6.5. We can compute $u = \text{wlink}(v,a)$ of $u$ in $O(1)$ time unless $\text{wlink}(v,a)$ is difficult. If the $\text{wlink}(v,a)$ is difficult, we can compute $u = \text{wlink}(v,a)$ in $O(\log \log \sigma)$ time.

Proof. Suppose that $u$ is heavy. If $v$ is w-special, we can find $u$ in $O(1)$ time using $D_u$. If $v$ is not w-special and it has at most $d + 1$ w-children, then we find $u$ in $O(1)$ time using data structures on $B$. Let $[l_v, r_v]$ denote the suffix range of $v$. The suffix range of $u$ is $[l_u, r_u]$ where $l_u = \text{Acc}[a] + \text{rank}_a(l_v - 1, B) + 1$ and $r_u = \text{Acc}[a] + \text{rank}_a(r_v, B)$. We can find $\text{rank}_a(r_v, B)$ as follows. Since $v$ has at most $d$ light w-children, the rightmost occurrence of $a$ in $B[l_v, r_v]$ is within the distance $d^2$ from $r_v$. Hence we can find the rightmost $i_v \leq r_v$ such that $B[i_v] = a$ by searching in the interval $G_j$ that contains $r_v$ or the preceding interval $G_{j-1}$. When $i_v$ is found, $\text{rank}_a(r_v, B) = \text{rank}_a(i_v, B)$ can be computed in $O(1)$ time because partial rank queries on $B$ are supported in time $O(1)$. We can compute $\text{rank}_a(l_v - 1, B)$ in the same way. When rank queries are answered, we can find $l_u$ and $r_u$ in constant time. Then we can identify the node $u$ by computing the lowest common ancestor of $l_u$-th and $r_u$-th leaves in $T$.

If $v$ is not special and it has more than $d-1$ outgoing w-links, then $u$ is the only heavy target node of a w-link starting at $v$; hence, its index $i$ is stored in the node $v$. Suppose that $u$ is light and $v$ is also light. Then the suffix range $[l_v, r_v]$ of $v$ has length at most $d$. $B[l_v, r_v]$ intersects at most two intervals $G_j$. Hence we can find $\text{rank}_a(l_v - 1, B)$ and $\text{rank}_a(r_v, B)$ in constant time. Then we can find the range $[l_u, r_u]$ of the node $u$ and identify $u$ in time $O(1)$ as described above. If $v$ is heavy and $u$ is light, then $\text{wlink}(v,a)$ is a difficult w-link. In this case we need $O(\log \log \sigma)$ time to compute $\text{rank}_a(l_v - 1, B)$ and $\text{rank}_a(r_v, B)$. Then we find the range $[l_u, r_u]$ and the node $u$ is found as described above.

Proposition A.6.6. Any sequence of nodes $u_1, \ldots, u_n$ where $u_i = \text{wlink}(u_{i-1}, a_{i-1})$ for some symbol $a_{i-1}$ contains at most one difficult w-link.

Proof. Let $\pi$ denote the path of w-links that contains nodes $u_1, \ldots, u_n$. Suppose that a node $u_1$ is a heavy node and $u_n$ is a light node. Let $u_1$ denote the first light node on the path $\pi$. Then all nodes on the path from $u_1$ to $u_n$ are light nodes and $\text{wlink}(u_{i-1}, a_{i-1})$ is the only difficult w-link on the path from $u_1$ to $u_i$. If $u_1$ is light or $u_n$ is heavy, then all nodes on $\pi$ are light nodes (resp. all nodes on $\pi$ are heavy nodes). In this case there are apparently no difficult w-links between $u_1$ and $u_n$.

Pre-processing. Now we show how we can construct above described auxiliary data structures in linear time. We start by generating the suffix tree topology and creating data structures $F_u$ and $D_u$ for all nodes $u$. For every node $u$ in the suffix tree we create the list of its children $u_i$ and their labels in $O(n)$ time. For every tree node $u$ we can find the number of its leaf descendants using standard operations on the suffix tree topology. Hence, we can determine whether $u$ is a heavy or a light node and whether $u$ is a special node. When this infor-
We can create data structures necessary for navigating along w-links in a similar way. We visit all nodes \( \nu \) of \( \mathbf{T} \). Let \( \ell_a \) and \( r_a \) denote the indexes of leftmost and rightmost leaves in the subtree of \( u \). Let \( B \) denote the BWT of \( T \). Using the method of Lemma A.5.1, we can generate the list of distinct symbols in \( B[\ell_a..r_a] \) and count how many times every symbol occurred in \( B[\ell_a..r_a] \) in \( O(1) \) time per symbol. If a symbol \( a \) occurred more than \( d \) times, then \( w_{\text{link}}(u, a) \) is heavy. Using this information, we can identify \( w \)-special nodes and create data structures \( D'_a \). Using the method of [37], we can construct \( D'_a \) in \( O(n_a \log n_a) \) time. By Lemma A.6.3 the total number of target nodes in all \( D'_a \) is \( O(n/d) \); hence we can construct all \( D'_a \) in \( o(n) \) time. We can also find all nodes \( u \) with a unique \( w \)-link. All dictionaries \( D'_a \) and all unique \( w \)-links need \( O((n/d) \log n) = O(n) \) bits of space.

**Supporting a Sequence of Extendright Operations.**

**Lemma A.6.1.** If we know the suffix interval of a right-maximal factor \( T[i..i+j] \) in \( \mathbf{T} \) and the suffix interval of \( T[i..i+j] \) in \( \mathbf{B} \), we can find the intervals of \( T[i..i+j+l] \) and \( T[i..i+j+l+t] \) in \( O(t \log \log \sigma) \) time.

**Proof.** Let \( \mathbf{T} \) and \( \mathbf{B} \) denote the suffix tree for the text \( T \) and let \( \mathbf{T} \) denote the suffix tree of the reverse text \( T^r \).

We keep the data structure for navigating the suffix tree \( \mathbf{T} \), described in Proposition A.6.1 and the data structure for computing Weiner links described in Proposition A.6.5. We also keep the same data structures for \( \mathbf{B} \). Let \( \ell_{0,s}, \ell_{0,e} \) denote the suffix interval of \( T[i..i+j] \); let \( \ell_{0,s}, \ell_{0,e} \) denote the suffix interval of \( T[i..i+j] \).

We navigate down the tree following the symbols \( T[i+j+1], \ldots, T[i+j+t] \). Let \( a = T[i+j+k] \) for some \( k \) such that \( 1 \leq k \leq t \) and suppose that the suffix interval \( [\ell_{k-1,s}, \ell_{k-1,e}] \) of \( T[i..i+j+k-1] \) and the suffix interval \( [\ell'_{k-1,s}, \ell'_{k-1,e}] \) of \( T[i..i+j+k-1] \) are already known. First, we check whether our current location is a node of \( \mathbf{T} \). If \( \mathbf{B}[\ell'_{k-1,s}, \ell'_{k-1,e}] \) contains only one symbol \( T[i+j+k] \), then the range of \( T[i..i+j+k] \) is identical with the range of \( T[i..i+j+k-1] \). We can calculate the range of \( T[i..i+j+k-1] \) in a standard way by answering two rank queries on \( \mathbf{B} \) and \( O(1) \) arithmetic operations; see Section A.1. Since \( \mathbf{B}[\ell'_{k-1,s}, \ell'_{k-1,e}] \) contains only one symbol, rank queries that we need to answer are partial rank queries. Hence we can find the range of \( T[i..i+j+k] \) in time \( O(1) \). If \( \mathbf{B}[\ell'_{k-1,s}, \ell'_{k-1,e}] \) contains more than one symbol, then there is a node \( u \) in \( \mathbf{T} \) that is labeled with \( T[i..i+j+k-1] \); \( u = \text{lca}(\ell_{k-1,s}, \ell_{k-1,e}) \) where \( \text{lca}(f, g) \) denotes the lowest common ancestor of the \( f \)-th and the \( g \)-th leaves. We find the child \( u' \) of the node \( u \) in \( \mathbf{T} \) that is labeled with \( a = T[i+j+k] \).

We also compute the Weiner link \( \mathbf{w} = w_{\text{link}}(\mathbf{u}, a) \) for a node \( u' = \text{lca}(\ell'_{k-1,s}, \ell'_{k-1,e}) \) in \( \mathbf{T} \). Then \( \ell'_{k,s} = \text{leftmost leaf}(\mathbf{w}) \) and \( \ell'_{k,e} = \text{rightmost leaf}(\mathbf{w}) \). We need to visit at most \( t \) nodes of \( \mathbf{T} \) and at most \( t \) nodes of \( \mathbf{T} \) in order to find the desired interval by Proposition A.6.1 and Proposition A.6.2, the total time needed to move down in \( \mathbf{T} \) is \( O(t \log \log \sigma) \). By Proposition A.6.5 and Proposition A.6.6, the total time to compute all necessary w-links in \( \mathbf{T} \) is also \( O(t \log \log \sigma) \).

**Finding the Intervals.** The algorithm for computing PLCP, described in Section 5, assumes that we know the intervals of \( T[j \Delta..j \Delta' + \ell_i] \) and \( T[j \Delta'..j \Delta' + \ell_i] \) for \( i = j \Delta' \) and \( j = 0, 1, \ldots, n/\Delta' \).

These values can be found as follows. We start by computing the intervals of \( T[0..0] \) and \( T[0..0] \). Suppose that the intervals of \( T[j \Delta'..j \Delta' + \ell_i] \) and \( T[j \Delta'..j \Delta' + \ell_i] \) are known. We can compute \( \ell_{i+1} \) as shown in Section 5. We find the intervals of \( T[(j + 1) \Delta'..(j + 1) \Delta' + \ell_{i+1}] \) and \( T[(j + 1) \Delta'..(j + 1) \Delta' + \ell_{i+1}] \) in time \( O(\Delta') \) by executing \( \Delta' \) operations \text{contractleft}. Each operation \text{contractleft} takes constant time. Then we calculate the intervals of \( T[(j + 1) \Delta'..(j + 1) \Delta' + \ell_{i+1}] \) and \( T[(j + 1) \Delta'..(j + 1) \Delta' + \ell_{i+1}] \) in time \( O(\log \log \sigma + (\ell_{i+1} - \ell_i)) \) using Lemma A.6.1. We know from Section 5 that \( \sum(\ell_{i+1} - \ell_i) = O(n) \). Hence we compute all necessary intervals in time \( O(n + (n/\Delta') \log \log \sigma) = O(n) \).