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 have a construction time that depends on the alphabet size or need $\omega(n \log \sigma)$ bits of working space.
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 tree and suffix array are two most popular solutions of this fundamental problem. 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 array does not support all queries that can be answered by the suffix tree, it uses less space and is 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 fifteen years there was a significant increase of interest in compressed indexes, i.e., data structures that keep $T$ in compressed form and support string matching queries. Compressed suffix array (CSA) [16, 10] and compressed suffix tree (CST) [32] 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 [26] for an overview of compressed indexes.

It is well known that both suffix array and suffix tree can be constructed in $O(n)$ time [23, 35, 36]. The first algorithm that constructs the suffix tree in linear time independently of the alphabet size was presented by Farach [9]. There are also algorithms that directly construct the suffix array of $T$ in $O(n)$ time [19, 20]. 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 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. The algorithm of Hon et al. [18] constructs the CST in $O(n \log^2 n)$ time for an arbitrarily small constant $\varepsilon > 0$. In the same paper the authors also showed that CSA can be constructed in $O(n \log \log \sigma)$ time. The algorithm of Okanohara and Sadakane constructs the CSA in linear time, but needs $O(n \log \sigma \log \log n)$ space [30]. In a recent breakthrough Belazzougui [2] 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 $O(n \log \log \sigma)$ [3]. 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.

Overview. 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 $j$ steps and during the $j$-th step we process all suffixes that start at position $i\Delta - j - 1$ for a fixed $j$, $0 \leq j \leq \Delta - 1$ and all $i$, $1 \leq i \leq n/\Delta$. 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 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. Then we must update the sequence and insert new symbols into $B$. Unfortunately we need $\Omega(\log n / \log \log n)$ time to answer rank queries on a dynamic sequence [12]. Even if we do not have to update the sequence, we need $\Omega(\log \log \sigma)$ time to answer a rank query. 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 a 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. We believe that this result is of independent interest.

The data structures that answer batches of queries and supports batched updates are described in Sections 3, A.2, and A.3. This is the most technically involved part 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. Finally we show how the static data structure can be dynamized in Section A.3. Then we turn to the problem of constructing the compressed suffix tree. First we describe a data structure that answers partial \( \Delta \) meta-symbols. Let \( \Sigma \) be the alphabet size and \( \delta \) is smaller than all other symbols in \( T \). We assume that the text \( T \) ends with a special symbol $ and $ is smaller than all other symbols in \( T \). The alphabet size is \( \sigma \) and symbols are integers in \([0..\sigma - 1]\). 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 \([n/\Delta] \Delta - n \) $-symbols). The BWT of \( T \) is a sequence \( B \) defined as follows: if \( T[k..] \) is the \( i \)-th lexicographically smallest suffix, then \( B[i] = T[k - 1] \). Thus the symbols of a sequence \( B \) are 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 \( O(\Delta) \) steps. We say that a suffix \( S \) is a \( j \)-suffix for \( 0 \leq j < \Delta \) if \( S = T[i\Delta - j - 1..] \) and denote by \( S_j \) the set of all \( j \)-suffixes, \( S_j = \{ T[i\Delta - j - 1..] \mid 1 \leq i \leq n/\Delta \} \). 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

---

1 The exact formula is \( B[i] = T[(k - 1) \mod n] \). We will write \( B[i] = T[k - 1] \) to avoid tedious details.
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. Then we apply a linear-time and linear-space suffix array construction algorithm [19] 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 starting with meta-symbol $T[i \Delta - 1]T[i \Delta - 2] \ldots$ in $T_3$ and the suffix $T[i \Delta - 2..]$ corresponds to the suffix starting with $T[i \Delta - 2]T[i \Delta - 3] \ldots$. 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 $T[i \Delta..]$ is stored in the entry $W[i]$ of an auxiliary array $W$. We also keep an auxiliary array $P$ of size $\sigma$. $P[a]$ is equal to the number of occurrences of symbols $i \leq a - 1$ in the 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 $\cup_{i=0}^{j-1} S_t$. For every suffix $T[i \Delta - (j - 1)..]$, we know its position in $B$. Every suffix $S_i = T[i \Delta - j..] \in S_j$ can be represented as $S_i = aS_i'$ for some symbol $a$ and the suffix $S_i' = T[i \Delta - (j - 1)..] \in S_{j-1}$. We look up the position $t_i$ of $S_i'$ and answer rank query $r_i = \text{rank}_a(t_i, B)$. We need $\Omega(\log \frac{\log \sigma}{\log \log n})$ 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 a query [12]. However in our case the scenario is different: we ask 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 $\cup_{t=1}^{j} S_t$: there are exactly $p_i$ suffixes in $\cup_{t=1}^{j} S_t$ that are smaller than $S_i$, where $p_i = P[a] + r_i$. Correctness of this computation can be proved as follows.

**Proposition 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 $\cup_{t=1}^{j} S_t$, such that $S_p$ starts with $a'$. Further, there are exactly $r_i$ suffixes $S_v$ in $\cup_{t=1}^{j} S_t$ such that $S_v \leq S_i$ and $S_v$ starts with $a$.

**Proof:** Suppose that a suffix $S_p$ from $S_t$, such that $j \geq t \geq 1$, 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 $S'_{i_1}$. Now suppose that a suffix $S_i \in \mathcal{S}_t$, such that $j \geq t \geq 1$, starts with $a$ and $S_{i_1} \leq S_i$. Then $S_{i_1} = aS'_{i_1}$ for $S'_{i_1} \in \mathcal{S}_{t-1}$ and $S'_{i_1} \leq S'_{i_1}$. For every such $S'_{i_1}$ there is exactly one occurrence of the symbol $a$ in $B[1..t_1]$, where $t_1$ is the position of $S'_{i_1}$ in $B$.

The above calculation did not take into account the suffixes from $\mathcal{S}_0$. We compute the number of suffixes $S_k \in \mathcal{S}_0$ 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 $\mathcal{S}_0$ and $\mathcal{S}_j$ 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 $\mathcal{S}_0 \cup \mathcal{S}_j$ are sorted, we traverse the sorted list of suffixes; for every suffix $S_i \in \mathcal{S}_j$ we know the number $q_i$ of lexicographically smaller suffixes from $\mathcal{S}_0$.

We then modify the sequence $B$: We sort new suffixes $S_i$ by $o_i = p_i + q_i$. Then we insert the symbol $T[i\Delta − j − 1]$ at position $o_i$ in $B$; 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 $\mathcal{S}_j$ in the sequence $B$. Since the positions of suffixes from $\mathcal{S}_{j-1}$ are not needed any more, we use the entry $W[i]$ of $W$ to store the position of $T[i\Delta − j..]$. The array $P$ is also updated.

When Step $\Delta − 1$ is completed, the sequence $B$ contains $n$ symbols and $B[i]$ is the symbol that precedes the $i$-th smallest suffix of $T$. Thus we obtained a 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 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 $\frac{n}{\log^2 n} \leq m \leq n$ 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[11] can be used to answer rank queries in time $O(1)$.

Following previous work [14], 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 $a$’s occurrences 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 $R'$. For all pairs in $H_{a,r}$ we keep a data structure $D_{a,r}$ that contains the second components of all
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, then we keep all pairs starting with symbol \(a\) in one group \(H_{a,0}\). 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 [13]. 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 [28]; details will 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 \(\text{rank}_a(i, C)\) 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 \(\text{rank}_a(i, C)\) the rightmost pair \((a, j')\) in \(R'\), such that \(j' \leq i\). Pair \((a, j')\) provides us with an approximate answer to \(\text{rank}_a(i, C)\) (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 \(\text{rank}_{a_1}(i_1, C), \text{rank}_{a_2}(i_2, C), \ldots, \text{rank}_{a_v}(i_v, C)\) on a chunk \(C\). We sort the sequence of queries by pairs \((a_j, i_j)\) in increasing order. 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 \(O(\sigma/\log^2 n)\) time; if \(v \geq \sigma/\log^3 n\), we sort in \(O(v \log n) = O(\sigma/\log^2 n)\) 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 \((a_t, p_t)\) 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) \(p_t \leq i_j, a_j = a_t, \text{ and } a_t \neq a_{t+1}\). That is, we find the largest \(p_t \leq i_j\) such that \((a_j, p_t) \in R'\) 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 \(H_{a_j,0}\). Using \(D_{a_t,p_t}\) (resp. \(D_{a_j,0}\)), we find the largest position \(x_t \in H_{a_t,p_t}\) such that \(x_t \leq i_j\). Thus \(x_t\) is the largest position in \(C\) satisfying \(x_t \leq i_j\) and \(C[x_t] = a_j\). We can compute \(\text{rank}_{a_t}(x_t, C)\) as follows: Let \(n_1\) be the partial rank of \(C[p_t]\), \(n_1 = \text{rank}_{C[p_t]}(p_t, C)\). 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 \(C[i_j]\), \(\text{rank}_{a_j}(i_j, C) = \text{rank}_{a_t}(x_t, C)\). 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 \(v\) queries on a chunk is answered in \(O(v + \sigma/\log^2 n)\) 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 \((i/\sigma) + 1\) 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 \(O(m + n/\log^2 n) = O(m)\) time. Now we know the rank \(n_{j,2} = \text{rank}_{a_j}(i_{j}', C)\), where \(i_{j}' = i_j - \lfloor i_j/\sigma \rfloor \sigma\) 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 \(n + \lfloor n/\sigma \rfloor \sigma\) bits; hence they use \(O(n)\) bits of space. We can compute the number of \(a\)’s occurrences in the first \(j\) chunks in \(O(1)\) time by answering one select query. Consider a rank query \(\text{rank}_{a_j}(i_j, B)\) and suppose that \(n_{j,2}\) is already known. We compute \(n_{j,1},\)
where \( n_{j,1} = \text{select}_0([i_j/\sigma], M_{a_j}) - M_{a_j} \) is the number of times \( a_j \) occurs in the first \( [i/\sigma] \) chunks. Then we compute \( \text{rank}_{a_j}(i_j, B) = n_{j,1} + n_{j,2} \).

**Theorem 2** 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 \( \frac{n}{\log \sigma} \leq m \leq n \).

The static data structure of Theorem 2 can be dynamized so that batched queries and batched insertions are supported. We describe the dynamic data structure in Sections A.2 and A.3.

### 4 Building the Suffix Tree Topology

Belazzougui proved the following result in [2]: 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 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 suffix tree intervals in \( O(n) \) time, then we can build the suffix tree topology 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 [2] for details.

In Section 5 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 \( O(n \log \log n) \) bits. The details are given in Section A.4. Therefore, we obtain the following result.

**Lemma 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 6 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 3** 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 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 [11]. In the remaining part of this section we will assume that \( \sigma \geq \log^3 n \).

**Lemma 2** 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 introduced in [4]. 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
in the bucket $B_{a,s}$ and let $l_{a,s}$ denote the length of $p_{a,s}$. For every elements $C[i]$ in the sequence we keep the value of $l_{C[i],t}$ where $B_{C[i],t}$ is the bucket containing $i$. If $I_{C[i]}$ was not divided into buckets, we assume $l_{C[i],t} = \textit{null}$, 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 $O(\log \log m)$ bits. The partial rank of $C[i]$ in $C$ can be computed from $t$ and $r$, $\text{rank}_{C[i]}(i, C) = t \log^2 m + r$.

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 [4] for the 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 of $u$ 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 $B_{a,k_a}$ 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 $B_{a,k_a}$ that is assigned to $u_j$. We implement $G_j$ as deterministic dictionaries of Hagerup et al. [17]. $G_j$ uses $O(m_j \log \sigma)$ bits and can be constructed in $O(m_j \log \sigma)$ time. We store $B_j$ only for bucket nodes $u_j$ such that $m_j > 0$. We also keep an array $W[1..\frac{m}{\log^2 m}]$ whose entries correspond to bucket nodes of $T_w$: $W[j]$ contains a pointer to $B_j$ or null if $B_j$ does not exist.

Using $W$ and $B_j$ we can answer a partial rank query $\text{rank}_{C[i]}(i, C)$. 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 find the index $j$ of the node $u_j$ that corresponds to $p_{a,t}$, $j = (2^{l_{a,t}} - 1) + p_{a,t}$. We lookup the address of the data structure $B_j$ in $W[j]$. Finally the index $t$ of the bucket $B_{a,t}$ is computed as $t = G[i]$ where $i = \text{rank}_1(a, R)$.

A data structure $G_j$ consumes $O(m_j \log m)$ bits. Since $\sum_j m_j \leq \frac{m}{\log^2 m}$, all $B_j$ 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. \hfill $\square$

**Theorem 4** 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 $M_a$. \hfill $\square$

## 6 Constructing the Permuted LCP Array

The permuted LCP array is defined as $\text{PLCP}[i] = j$ if and only if $SA[r] = i$ and the longest common prefix of the $r$-th and $(r - 1)$-st suffixes is $j$. In other words $\text{PLCP}[i]$ is the longest common prefix of $T[i..]$ and the suffix that precedes $T[i..]$ 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 \( q - 1 \). By the same argument \( \ell_i \leq \ell_{i+\Delta} - \Delta \) for \( \Delta = \log \log n \). To simplify the description we will further assume that \( \ell_{i-1} = 0 \). 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 \( T = T[n-1]T[n-2] \ldots T[1]T[0] \). 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. It was demonstrated in [34, 5] that both operations can be supported by answering \( O(1) \) rank queries on \( B \) and \( \overline{B} \).

Belazzougui [2] proposed the following algorithm for consecutively 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 \( T[i-1..] \), 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 \( T[i..] \). Then we find the interval \( [r_s, r_e] \) of \( T[i..i+\ell_{i-1} - 1] \) in \( B \) and the interval \( [r'_s, r'_e] \) of \( \overline{T[i..i+\ell_{i-1} - 1]} \) in \( \overline{B} \). These two intervals can be computed by \( \text{contractleft} \). In the special case when \( i = 0 \) or \( \ell_{i-1} = 0 \), we set \( [r_s, r_e] = [r'_s, r'_e] = [0, n-1] \). 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 \( [r_{s,j}, r_{e,j}] \) 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 [2, Lemma 2]. When \( \ell_j \) is computed, we increment \( j \) and find the next \( \ell_j \) in the same way. All \( \ell_j \) 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 \( T \) and \( \overline{T} \) 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 \( P \) keeps information about accumulated frequencies of symbols: \( P[i] \) is the number of occurrences of all symbols \( a < i \) 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 \( l = \text{select}_c(i-P[c], B) \) and \( r = \text{select}_c(j-P[c], B) \). Then we find the lowest common ancestor \( x \) of leaves \( l \) and \( r \). We set \( i_1 = \text{leftmost_leaf}(x) \) and \( j_1 = \text{rightmost_leaf}(x) \). 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 \( p \) in \( T \) are followed by \( c \) and \( [i_1, j_1] = [i', j'] \). Otherwise we find the lowest common ancestor \( y \) of leaves \( i' \) and \( j' \) in \( \overline{T} \). Then we identify \( y' = \text{parent}(y) \) in \( \overline{T} \) and \( i'_1 = \text{leftmost_leaf}(y') \) and \( j'_1 = \text{rightmost_leaf}(y') \). Thus \( \text{contractleft} \) can be supported in \( O(1) \) time. Now we consider \( \text{extendright} \). Suppose that \( [i, j] \) and \( [i', j'] \) are intervals of \( p \) and \( \overline{p} \) in \( B \) and \( \overline{B} \) respectively. We compute the interval of \( \overline{cp} \) by using the standard BWT machinery. Let \( i'_1 = \text{rank}_c(i'_1 - 1, \overline{B}) + P[c] + 1 \) and \( j'_1 = \text{rank}_c(j'_1, \overline{B}) + P[c] \). We check whether \( c \) is the only symbol in \( B[i..j] \). If this is the case, then all occurrences of \( p \) in \( T \) are preceded by \( c \) and all occurrences of \( p \) in \( T \) are followed by \( c \). Hence the interval of \( pc \) in \( B \) is \( [i_1, j_1] = [i, j] \). 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 \( j \). We find the child \( y \) of \( x \) that is labeled by \( c \) and set \( i_1 = \text{leftmost_leaf}(y), j_1 = \text{rightmost_leaf}(y) \). We can find the child of \( x \) labeled
with c by searching the sequence L. L contains the labels of children for all nodes of T. We also
encode the degrees of all nodes in a sequence D = 1^{d_1}01^{d_2}0 . . . 1^{d_n}, where d_i is the degree of the
i-th node. We compute v = rank_1<select_0(f + 1, D), D) and v_1 = rank_1<select_0(f, D), D). Then we
compute p_1 = rank_c(v, L), p_2 = select_c(p_1, L), and j = p_2 − v_1. Then y is the j-th child of x. The
bottleneck of extendright is the computation of p_1 because we need Ω(log log log_σ n) time to answer
a rank query on L; all other calculations can be executed in O(1) time.

Our Approach. Our algorithm follows the technique of [2] that relies on operations extendright and
contractleft for building the LCP. We implement these two operations as described above;
thus we will have to perform Θ(n) rank queries on sequences L and B. We must create large
batches of queries in order to apply Theorem 2 and answer each query in O(1) time.

During the pre-processing stage we compute the BWT B of T and the BWT B for the reverse
text T. We create the machinery for supporting operations extendright and contractleft. We
also record the positions in B that correspond to suffixes T[iΔ..] for i = 0, . . . , ⌊n/Δ⌋. LCP
construction is divided into two stages: first we compute the values of ℓ_i for selected evenly spaced
indices i, i = jΔ and j = 0, 1, . . . , ⌊n/Δ⌋. This can be done as described in Section A.5. During the
second stage, we compute all remaining values of ℓ_i. The key to a fast algorithm is to “parallelize”
the computation. At any time we process a list containing at least n/log^2 n jobs. We answer
rank queries in batches: when a slow rank query on L or B must be answered, we send it 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. A detailed description follows.

Stage 1. Our algorithm starts by computing ℓ_i for i = jΔ and j = 0, 1, . . . , ⌊n/Δ⌋. Let j = 0 and
f = jΔ. We already know the rank r_f of S_f = T[jΔ..] in B. We can also find the starting position
f' of the suffix S'_f of rank r_f − 1, S'_f = T[f'..]. Since f' can be found by employing the function
LF at most Δ times, we can compute f' in O(Δ) time. When f and f' are known, we scan T[f..]
and T[f'..] until the first symbol T[f + pf] ̸= T[f' + pf'] is found. By definition of ℓ_j, ℓ_0 = pf − 1.
Suppose that ℓ_sΔ for s = 0, . . . , j − 1 are already computed and we have to compute ℓ_f for f = jΔ
now. We already know the rank r_f of suffix T[f..]. We find f' such that the suffix T[f'..] is of rank
r_f − 1 in time O(Δ). We showed above that ℓ_f ≥ ℓ_(j−1)Δ − Δ. Hence the first of symbols in T[f..]
and T[f'..] are equal, where of = max(0, ℓ_(j−1)Δ − Δ). We scan T[f + of..] and T[f' + of..] until
the first symbol T[f + of + pf] ̸= T[f' + of + pf'] is found. By definition, ℓ_f = of + pf. Then we
increment j and repeat the same procedure. It can be shown that ℓ_f is O((n/Δ)Δ + ∑ pf) = O(n).
Hence the total time needed to compute all selected ℓ_f is

Stage 2. We divide ℓ into groups of size Δ − 1 and compute the values of ℓ_k in every group using
a job. The i-th group contains lengths ℓ_k+1, ℓ_k+2, . . . , ℓ_k+Δ−1 for k = iΔ and i = 0, 1, . . . . All ℓ_k
in the i-th group will be computed by the i-th job J_i. Every J_i is either active or paused. Thus
originally we start with a list of n/Δ jobs and all of them are active. All active jobs are executed
at the same time. That is, we scan the list of active jobs, spend O(1) time on every active job, and
then move on to the next job. When a job must answer a rank query, we pause it and insert the
query into a query list. There are two query lists: Q_l contain rank queries on sequence L and Q_b
contains rank queries on B. When Q_l or Q_b contains n/log^2 n queries, we answer all queries in Q_l
(resp. in Q_b). The batch of queries is answered using Theorem 2, so that every query is answered

\footnote{A faster computation is possible, but we do not need it here.}

9
in $O(1)$ time. Answers to queries are returned to jobs, corresponding jobs are re-activated, and we continue scanning the list of active jobs. When all $\ell_k$ for $i\Delta < k < (i+1)\Delta$ are computed, the $i$-th job is finished; we remove this job from the pool and decrement by 1 the number of jobs.

Every job $J_i$ computes $\ell_{k+1}, \ell_{k+2}, \ldots, \ell_{k+\Delta-1}$ for $k = i\Delta$ using the algorithm of Belazzougui [2]. When the interval of $T[i+\ell_{k..}]$ in $B$ and the interval of in $B$ are known, we compute $\ell_{k+1}$. The procedure for computing $\ell_{k+1}$ must execute one operation contractleft and $\ell_{k+1} - \ell_{k+1}$ operations extendright. Operations contractleft and extendright are implemented as described above. We must answer one rank query on $B$ and one rank query on $L$ for every extendright. Ignoring the time for these two 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 $O(\ell_{i\Delta+j} - \ell_{i\Delta} + j)$ time to compute $\ell_{i\Delta+1}, \ell_{i\Delta+1}, \ldots, \ell_{i\Delta+j}$.

“Parallel processing” of jobs is stopped 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 $n/\log n$ unknown $\ell_i$ at this point. We switch to method of Stage 1 to compute the values of unknown $\ell_i$. We sort all remaining $\ell_i$ and process them in order of increasing $i$. For every unknown $\ell_i$ we compute its rank $r$ in $B$. For the suffix $S'$ of rank $r-1$ we find its starting position $f'$ in $T$, $s' = T[f'..]$. Then we scan $T[f' + \ell_i-1 - 1..]$ and $T[f + \ell_i-1 - 1..]$ 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 $O(\Delta)$ 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. Total number of jobs in the job list does not exceed $n/\Delta$. Total number of queries in lists $Q_i$ and $Q_b$ does not exceed $n/\log^2 n$. Hence our algorithm uses $O(\log \sigma)$ bits of workspace.

**Lemma 3** 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.

## 7 Conclusions

We have shown that the Burrows-Wheeler Transform and the Compressed Suffix Tree can be built in deterministic $O(n)$ time by an algorithm that requires $O(n \log \sigma)$ bits of working space.

These results have many interesting applications. For example, we can now construct an FM-index [10, 11] in $O(n)$ deterministic time using $O(n \log \sigma)$ bits. Theorem 4 enables us to support function LF in $O(1)$ time on an FM-index. Previous results need $O(n \log \log \sigma)$ time or rely on randomization [18, 2]. In Section A.6 we describe a new index based on these ideas.

Another application is that we can now compute the Lempel-Ziv parse [22] of a string $T[1..n]$ in deterministic linear time $O(n)$ using $O(n \log \sigma)$ bits: Köppel and Sadakane [21] recently showed that, if one has a compressed suffix tree on $T$, then they need only $O(n)$ additional (deterministic) time and $O(z \log n)$ bits to produce the parsing, where $z$ is the resulting number of phrases. Since $z \leq n/\log \sigma n$, the space is $O(n \log \sigma)$ bits. From 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 $B$ [14]; the latter is also easily obtained with current topology representations using parentheses [29].

We believe that our result can also improve upon some of the recently presented data structures for dynamic document collections [24] and bidirectional FM-indices [34, 5].
 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 structure. 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 $i$ symbols in $B$, $\text{rank}_a(i, B) = |\{j \mid B[j] = a \text{ and } 1 \leq j \leq i\}|$. 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 queries is the access query, $\text{access}(i, B)$, that returns the $i$-th symbol $B[i]$ in $B$. If insertions and deletions of symbols in $B$ must be supported, then both kinds of queries require $\Omega(\log n / \log \log n)$ time [12]. 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 $\Theta(\log \log \sigma)^3$. 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 4 we describe a data structure for partial rank queries that can be constructed in $O(n)$ deterministic time. Better results can be also achieved in the special case when the alphabet size $\sigma = \log O(1) n$; in this case we can represent $B$ so that rank, select, and access queries are answered in $O(1)$ time [11].

**Suffix Tree and Suffix Array.** A suffix tree for a string $T[0..n-1]$ is a compacted tree on 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 $i$-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 $SA$.

**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). BWT is related to the suffix array as follows: $BWT[i] = T[(SA[i] - 1) \mod n]$. Hence, we can build BWT by sorting suffixes and writing symbols that precede the suffixes in lexicographical order. This method is used in Section 2.

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

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

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 that 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 $i_2 = \text{rank}_c(i_1 - 1, B) + P[c]$ and $j_2 = \text{rank}_c(i_2, B) + P[c]$ where $c = P[i]$. Thus the interval of $p$ is found by answering $m$ rank queries. We observe that the interval of $p$ in $B$ is exactly the same as the interval of $p$ in the suffix array $SA$.

---

3If we aim to use $n \log \sigma + o(n)$ bits, then there are different trade-offs between costs of select and access. If, however, $O(n \log \sigma)$ bits are available, then we can support both queries in $O(1)$ time.
Another important component of an FM-index is the function LF, defined as follows: if $SA[j] = i + 1$, then $SA[LF(j)] = i$. Using $LF$, we can find the starting position of the $r$-th suffix in $O(b)$ time, where $b$ is the sampling factor. We refer to [26] for details.

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

- The compressed suffix array of $T$. We can also use FM index as alternative implementation.
- The suffix tree topology. This component can be stored in $O(n)$ bits[32].
- The permutated 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 equal to $j$. The permutated LCP array is defined as follows: $PLCP[i] = j$ 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 [32].

## 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 positions of all symbols that follow it. Since symbols are stored in pairs $(a_j, i)$ 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 $0 \leq lab(i) \leq \sigma^{O(1)}$ 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 require $O(\log^2 n)$ 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 $x_1 \leq x_2 \leq \ldots \leq x_t$ 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 $[lab(x_i), lab(x_{i+1}) - 1]$ 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 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 $y_j$ is in an interval $I = [lab(x_s), lab(x_e)]$ if $x_s < y_j < x_e$. 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 $I = [lab(x_s), lab(x_e)]$, $\rho(I) = \frac{x_{e} + 1 - x_{s} - 1}{new(I)}$. 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 $j$, $1 \leq j \leq r$. We can find $I_1, \ldots, I_r$ in $O(|C|/\log n)$ time. For every $I_j$, $1 \leq j \leq r$, we evenly distribute the labels of old and new elements in the interval $I_j \subset I_j$. Suppose that $f$ new elements $y_p, \ldots, y_{p+f-1}$ are inserted into interval $I_j = [lab(x_s), lab(x_e)]$ so that now there are $v = f + (e - s) + 1$ elements in this interval. We assign the label $x_s + d_j \cdot (i - 1)$ to the $i$-th smallest element in $I_j$ where $d_j = \frac{\text{lab}(x_e) - \text{lab}(x_s)}{v - 1}$.
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 did 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 two during the first $\log n$ batches.

One remaining problem with our scheme is the large range of labels. Since labels are integers bounded by $4C|n|$, 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 symbols in a block. A label assigned to 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 sequence $L$ contains all symbols of $C$ in the same order as they appear in $C$. We split $L$ into $l$-blocks of $\Theta(\log \sigma n)$ symbols so that each block contains at most $\log \sigma n/4$ symbols. For every $l$-block $b$ we keep the total number of symbols in all preceding blocks and its label $\text{lab}(b)$. Labels are assigned to $l$-blocks with the method described in Section A.2. We also maintain a data structure that can answer rank queries on every block. The data structure for an $l$-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 $\log \sigma n/4$, we can keep pre-computed answers to all rank queries for all possible blocks in the table $\text{Tbl}[[0..n^{1/4} - 1][0..n^{1/4} - 1][j]]$. The entry $\text{Tbl}[b][a][i]$ contains the answer to the query $\text{rank}_a(i, b)$ on an $l$-block $b$. $\text{Tbl}$ 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 for 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 of $C$. Therefore we only store the pointer $p_b$ in each pair $(a, p_b) \in R$.

For every pair $(a, p_b) \in R'$ we also keep the label $\text{lab}(b)$ of the block that $p_b$ points to; we will denote this label $\text{lab}(p_b)$. Each pointer uses $O(\log \sigma)$ bits and it uniquely identifies the block of $L$. However we cannot use block pointers for searching in $L$ (resp. searching in $C$). Block labels are monotonously increasing and $\text{lab}(b_1) < \text{lab}(b_2)$ if the block $b_2$ follows $b_1$ in $L$. Hence block labels can be used for searching and answering rank queries. But block labels need $\Theta(\log n)$ bits of space and we only store them for pairs in $R'$. We employ compact data structures in order to search in $R$ between two consecutive pairs from $R'$.

Groups $H_{a,j}$ are defined as in Section 3; each $H_{a,j}$ contains all 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 $(a, p_{b_1}), \ldots, (a, p_{b_f})$ and pointers in these pairs point to blocks $b_1, \ldots, b_f$. We keep the compact data structure from [15] on $\text{lab}(b_1), \ldots, \text{lab}(b_f)$. This data structure requires $O(\log \log n)$ additional bits per label; for any integer $q$ we can find the
largest block label \( \text{lab}(b_i) \leq q \) in \( O(1) \) time or count the number of blocks \( b_i \) such that \( \text{lab}(b_i) \leq q \) in \( O(1) \) time; the search procedure needs to access one block label.

**Queries.** Suppose that we want to answer queries \( \text{rank}_{a_1}(i_1, C), \text{rank}_{a_2}(i_2, C), \ldots, \text{rank}_{a_t}(i_t, C) \) on a chunk \( C \). We traverse all 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 \( r_{j,1} = \text{rank}_{a_j}(i'_j, b_j) \) where \( i'_j \) is the relative position of the \( i_j \)-th symbol in \( b_j \). We represent queries by pairs \((a_j, l_j)\) and sort these pairs in increasing order. 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 \( \text{lab}(p_j) \leq l_j \). Let \( r_2 \) denote the rank of \((a_j, p_j)\) in \( R_{a_j} \), i.e. the number of pairs \((a_j, i)\) \( \in R \) preceding \((a_j, p_j)\). We can keep this information for every pair in \( R' \) using \( O(\log \sigma) \) additional bits. Then we use the data structure \( D_{a_j, p_j} \) that contains information about pairs in \( H_{a_j, p_j} \) (i.e., pairs in the group starting with \((a_j, p_j)\)). We find the largest \( \text{lab}(b_j) \in D_{a_j, p_j} \) such that \( \text{lab}(b_j) \leq l_j \) and count the number \( r_{j,3} \) of blocks \( b_j \in D_{a_j, p_j} \) such that \( \text{lab}(b_j) \leq \text{lab}(b_j) \). The answer to the \( j \)-th rank query is \( \text{rank}_{a_j}(i_j, C) = r_{j,1} + r_{j,2} + r_{j,3} \).

**Insertions.** Suppose that symbols \( a_1, \ldots, a_t \) are to be inserted at positions \( i_1, \ldots, i_t \) respectively. We traverse the sequence \( L \) and identify locations where new symbols must be inserted. Then we update the information about the number of preceding elements for all \( l \)-blocks. If some block contains more than \( \log \sigma n \) symbols, we split it into an appropriate number of blocks, so that each block contains \( \Theta(\log \sigma n) \) at most \( \frac{1}{2} \log \sigma n \) symbols. After \( t \) insertions, we create at most \( t/\log \sigma n \) new blocks. New blocks are assigned labels using the method described in Section A.2. Hence we change labels for \( O(t/\log \sigma n) \) blocks. If the label of a block \( b \) was changed, we visit all pairs \((a_z, p_b)\) in \( R \) that point to \( b \). Each \((a_z, p_b)\) is kept in some group \( H_{a_z, k} \) and in some data structure \( D_{a_z, k} \); we delete the old label of \( b \) from \( D_{a_z, k} \) and insert the correct new label. The total number of updates is thus bounded by \( O(t) \). Since every update takes \( O(1) \) time, the total cost of updating all block labels is \( O(t) \). When symbols are inserted into \( L \), we also insert them (resp. pointers to their \( l \)-blocks) into \( R \). We answer \( t \) rank queries \( r_1 = \text{rank}_{a_1}(i_1, C), \ldots, \text{rank}_{a_t}(i_t, C) \) as described above. Then we traverse \( R' \) and identify groups \( H_{a_1, j_1}, \ldots, H_{a_t, j_t} \), where new symbols must be inserted. We update the corresponding data structures \( D_{a_k, j_k} \) for \( 1 \leq k \leq t \). If necessary, we split some groups into several groups; the sequence \( R' \) is modified accordingly. We can answer rank queries, traverse \( R' \), and update groups \( H_{a_k, j_k} \) in \( O(\sigma/\log \sigma n + t) \) time.

**Global Sequence.** In addition to chunk data structures, we keep global sequence \( M_a = 1^{d_1}0 \ldots 1^{d_s} \) 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 \( m_j \) 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 \( \text{rank}_{a_k}(i_k, B) \) by answering a rank query on the chunk that contains \( B[i_k] \) and \( O(1) \) queries on the sequence \( M_a \). Queries on \( M_a \) are supported in \( O(1) \) time. 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 \( M_a \). We can insert new symbols into
We use the marking scheme described in [27]. Let  

\[ M_a \]  

in \( O(1) \) time per symbol. When the symbol sequence \( M_a \) is updated, we re-build the data structure supporting rank and select queries on \( M_a \). It is known that we can create such a data structure in \( O(n_a / \log n) \) time, where \( n_a \) is the number of bits in \( M_a \); see e.g. [25]. On average we insert \( O(1) \) bits into \( M_a \) per insertion of symbol \( a \). Hence the total amortized cost for a batch of \( m \geq n/\Delta \) insertions is \( O(n/\Delta) \).

**Theorem 5** 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 \( O(m) \) time for \( \frac{n}{\log \sigma} \leq m \leq n \).

### A.4 Reporting All the Symbols in a Range

We prove the following lemma in this section.

**Lemma 4** Given a sequence \( B[0..n - 1] \) over an alphabet \( \sigma \), we can build in \( O(n) \) time the data structure that uses \( O(n \log \log \sigma) \) additional bits and answers the following queries: for any range \([i..j]\), we can report \( \text{occ} \) distinct symbols that occur in \( B[i..j] \) in \( O(\text{occ}) \) time; for every reported symbol \( a \), we can report 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 [2], but we use the result of Theorem 4 to answer partial rank queries. This allows us to construct the data structure in \( O(n) \) deterministic time (while the data structure in [2] 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 [33]. For every reported symbol we can find in \( O(1) \) time its leftmost and its 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 4. Hence the data structure of Lemma 4 can be built in \( O(n) \) deterministic time. We can also use the data structure of Lemma 4 to determine whether the range \( B[i..j] \) contains only one distinct symbol in \( O(1) \) time. Let \( i_{a_1} \) and \( j_{a_2} \) be the first leftmost and the first rightmost indices returned for the range \( B[i..j] \). If \( B[i_{a_1}] \neq B[j_{a_2}] \) then at least two different symbols occur in \( B[i..j] \). Otherwise let \( a = B[i_{a_1}] = B[j_{a_2}] \) and let \( f_a \) be the frequency of \( a \) in \( B[i..j] \). If \( f_a = j - i + 1 \), then \( a \) is the only symbol in \( B[i..j] \). Otherwise there are other symbols besides \( a \) in \( B[i..j] \). This observation will be helpful in Section 6.

### A.5 Computing the Intervals

**Marking Nodes in a Tree.** We use the marking scheme described in [27]. 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 \([13]\); we can find the child 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 child labeled by \( a \) 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 \([17]\) 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 \( u \in \mathcal{T} \) to its child \( u_i \) unless the node \( u_i \) is difficult.

**Proposition 2** 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 3** 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 \( \mathcal{T} \) 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. The node \( u \) will be called the target node of the Weiner link and the node \( v \) will be called the source node. Suppose that the Weiner link connects the node \( v \) labeled by \( p \) to the node \( u \) labeled by \( cp \). In this case we say that \( v \) is the source node of a Weiner link, \( u \) is the target node, and the Weiner link is labeled by the symbol \( c \). 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 edges. 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 \([17]\) 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 \( v \).
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 find out whether a symbol \( a \) occurs in \( G_j \). Using \( G_{j,a} \), we can find for any position \( i \) the smallest \( i' \geq i \) such that \( B[i'] = a \) and \( B[i'] \) is in \( G_j \) (or the largest \( i'' \leq i \) such that \( B[i''] = a \) and \( B[i''] \) is in \( G_j \)). We implement both \( A_j \) and \( G_{j,a} \) as fusion trees[13] so that queries are answered in \( O(1) \) time. We also keep the data structure from [14] that supports partial rank queries in \( O(1) \) time and the data structure from Theorem 4 that supports partial rank queries in \( O(1) \) time.

**Proposition 4** 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 \( c_1p, c_2p, \ldots, c_sp \) occurs at least \( d \) times in \( T \). Consider the suffix tree \( \mathcal{T} \) of the reverse text \( \overline{T} \). \( \mathcal{T} \) 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 \( c_sp \) occurs at least \( d \) times in \( T \), \( \overline{pc}_1 \) occurs at least \( d \) times in \( \mathcal{T} \). Hence each \( \overline{u}_i \) has at least \( d \) descendants. Thus every w-special node in \( \mathcal{T} \) correspond to a special node in \( \mathcal{T} \) and every heavy w-link outgoing from a w-special node corresponds to some heavy child of a special node in \( \mathcal{T} \). 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 5** The total number of unique w-links is \( O(n/d) \).

**Proof:** We keep target nodes of A Weiner links \( wlink(v,a) \) is heavy only if \( wlink(v,a) \) 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 \( wlink(v,a) \) is difficult if its target node \( u = wlink(v,a) \) is light and its source node \( v \) is heavy.

**Proposition 6** We can compute \( u = wlink(v,a) \) of \( u \) in \( O(1) \) time unless \( wlink(v,a) \) is difficult. If the w-link is difficult, we can compute \( u = wlink(v,a) \) 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 \( l_u = P[a] + \text{rank}_a(l_v - 1, B) + 1 \) and \( r_u = P[a] + \text{rank}_a(r_v, B) \). 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 \( i_a \leq r_v \) such that \( B[i_a] = a \) by searching in the interval \( G_j \) that contains \( r_v \) or the preceding interval \( G_{j-1} \). When \( i_a \) is found, \( \text{rank}_a(r_v, B) = \text{rank}_a(i_a, B) \) 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 \( \mathcal{T} \).
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 \([u,v]\) 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 \([u,v]\) 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 \([u,v]\) and the node \( u \) is found as described above.

\[ \square \]

**Proposition 7** Any sequence of nodes \( u_1, \ldots, u_t \) 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:* 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 \( v \).

**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 information is available, we generate the data structures \( F_u \) and \( D_u \).

We can create data structures necessary for navigating along w-links in a similar way. We visit all nodes \( u \) of \( T \). Let \( l_u \) and \( r_u \) 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 4, we can generate the list of distinct symbols in \( B[l_u..r_u] \) and count how many times every symbol occurred in \( B[l_u..r_u] \) in \( O(1) \) time per symbol. If a symbol \( a \) occurred more than \( d \) times, then \( \text{wlink}(u,a) \) is heavy. Using this information, we can identify w-special nodes and create data structures \( \mathcal{D}_u \). Using the method of [31], we can construct \( \mathcal{D}_u \) in \( O(n_d \log \log n_a) \) time. By Lemma 4 the total number of target nodes in all \( \mathcal{D}_u \) is \( O(n_d) \); hence we can construct all \( \mathcal{D}_u \) in \( o(n) \) time. We can also find all nodes \( u \) with a unique w-link. All dictionaries \( \mathcal{D}_u \) and all unique w-links need \( O((n/d) \log n) = O(n \log \sigma) \) bits of space.

**Supporting a Sequence of extendright Operations.**

**Lemma 5** If we know the suffix interval of a right-maximal factor \( T[i..i+j] \) in \( B \) and the suffix interval of \( T[i..i+j] \) in \( B \), then we can find the intervals of \( T[i..i+j+t] \) and \( T[i..i+j+t] \) in \( O(t + \log \log \sigma) \) time.

*Proof:* Let \( T \) and \( \overline{T} \) denote the suffix tree for the text \( T \) and let \( \overline{T} \) denote the suffix tree of the reverse text \( \overline{T} \). We keep the data structure for navigating the suffix tree \( T \), described in Proposition 2. We also keep the data structure for computing Weiner links described above. Let \([l_{0,s},l_{0,e}]\) denote the suffix interval of \( T[i..i+j] \); let \([l_0,s',l_0,e']\) denote the suffix interval of \( \overline{T}[i..i+j] \). 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 $T$. If $B[\ell'_{k-1,s}, \ell'_{k-1,e}]$ 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 $T[i..i+j+k]$ in a standard way. Since rank queries are partial rank queries, we can find $T[i..i+j+k]$ in time $O(1)$. If $B[\ell'_{k-1,s}, \ell'_{k-1,e}]$ contains more than one symbol, then there is a node $u \in T$ that is labeled with $T[i..i+j+k-1]$; $u = lca(\ell_{k-1,s}, \ell_{k-1,e})$ where $lca(f,g)$ denotes the lowest common ancestor of the $f$-th $g$-th leaves. We find the child $u'$ of the node $u$ in $T$ that is labeled with $a$. We also compute the Weiner link $\overline{w} = wlink(\overline{u}, a)$ for a node $\overline{u}' = lca(\ell'_{k-1,s}, \ell'_{k-1,e})$. Then $\ell'_{k,s} = \text{leftmost} - \text{leaf}(\overline{w})$ and $\ell'_{k,e} = \text{rightmost} - \text{leaf}(\overline{w})$. \hfill $\square$

**Finding the Intervals.** The algorithm for computing PLCP, described in Section 6, 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 $T[0..\ell_0]$ and $T[0..\ell_0]$. 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 6. We find the intervals of $T[(j+1)\Delta..j\Delta + \ell_i]$ and $T[(j+1)\Delta..j\Delta + \ell_i]$ in time $O(\Delta)$ by executing $\Delta$ operations $\text{contractleft}$. 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 $O(\log \log \sigma + (\ell_{i+1} - \ell_i + \Delta))$ time using Lemma 5. We know from Section 6 that $\sum (\ell_{i+1} - \ell_i) = O(n)$. Hence we compute all necessary intervals in time $O(n + (n/\Delta) \log \log \sigma) = O(n)$.

### A.6 Compressed Index

In this section we show how our algorithms can be used to construct a compact index in deterministic linear time. We prove the following result.

**Theorem 6** We can construct an index for a text $T[0..n-1]$ over an alphabet of size $\sigma$ in $O(n)$ deterministic time using $O(n \log \sigma)$ bits of working space. This index occupies $nH_k + o(n \log \sigma) + O(n^{\log n/d})$ bits of space for a parameter $d > 0$. All occurrences of a query substring $P$ can be counted in $O(|P| + \log \log \sigma)$ time; all occurrences of $P$ can be reported in $O(|P| + \log \log \sigma + \text{occ} \cdot d)$ time. An arbitrary substring $P$ of $T$ can be extracted in $O(|P| + d)$ time.

**Interval Rank Queries.** We start by showing how a compressed data structure that supports select queries can be extended to support a new kind of queries that we dub small interval rank queries. An interval rank query $\text{rank}_a(i, j, B)$ asks for $\text{rank}_a(i', B)$ and $\text{rank}_a(j', B)$, where $i'$ and $j'$ are the leftmost and rightmost occurrences of the symbol $a$ in $B[i..j]$; if $a$ does not occur in $B[i..j]$, we return $\text{null}$. An interval query $\text{rank}_a(i, j, B)$ is a small interval query if $j - i \leq 2\log^2 \sigma$. Our compressed index relies on the following result.

**Lemma 6** Suppose that we are given a data structure that supports access queries on a sequence $C[0..m]$ in time $t_{\text{select}}$. Then, using $O(m \log \log \sigma)$ additional bits, we can support small interval rank queries on $C$ in $O(t_{\text{select}})$ time.

**Proof:** We split $C$ into groups $G_i$ so that every group contains $2\log^2 \sigma$ consecutive symbols of $S$, $G_i = C[i \log^2 \sigma..(i + 1) \log^2 \sigma - 1]$. Let $A_i$ denote the set of symbols that occur in $G_i$. We would
need $\log \sigma$ bits per symbol to store $A_i$. Therefore we keep only a dictionary $A'_i$ implemented as a succinct SB-tree [15]. Succinct SB-tree needs $O(\log \log m)$ bits per symbol; it can answer queries $a \in A_i$ in constant time if we can access elements of $A_i$. We can identify every $a \in A_i$ by its leftmost position in $G_i$. Since $G_i$ consists of $2\log^2 \sigma$ consecutive symbols, a position within $G_i$ can be specified using $O(\log \log \sigma)$ bits. Hence we can find access any symbol of $A_i$ in $O(1)$ time. For each $a \in A_i$ we also keep a data structure $I_{a,i}$ that stores all positions where $a$ occurs in $G_i$. Positions are stored as differences with the left border of $G_i$: if $C[j] = a$, we store the difference $j - i \log^2 \sigma$. Hence elements of $I_{a,i}$ can be stored in $O(\log \log \sigma)$ bits per symbol.

Using data structures $A'_i$ and $I_{a,i}$, we can answer small interval rank queries. Consider a group $G_t = C[t \log^2 \sigma..(t + 1) \log^2 \sigma - 1]$, an index $i$ such that $t \log^2 \sigma \leq i \leq (t + 1) \log^2 \sigma$, and a symbol $a$. We can find the largest $j \leq i$ such that $C[j] = a$ and $C[j] \in G_i$: first we look for the symbol $a$ in $A'_i$; if $a \in A'_i$, we find the predecessor of $j$ in $I_{a,t}$. An interval $C[i..j]$ of size $d \leq \log^2 \sigma$ intersects at most two groups $G_t$ and $G_{t-1}$. We can find the rightmost occurrence of a symbol $a$ in $[i,j]$ as follows. First we look for the rightmost occurrence $j' \leq j$ of $a$ in $G_t$; if $a$ does not occur in $C[t \log^2 \sigma..j]$, we look for the rightmost occurrence $j' \leq t \log^2 \sigma - 1$ of $a$ in $G_{t-1}$. We can find the leftmost occurrence $i'$ of $a$ in $C[i..j]$ using a symmetric procedure. When $i'$ and $j'$ are found, we can compute $\text{rank}_a(i', C)$ and $\text{rank}_a(j', C)$ in $O(1)$ time by answering partial rank queries. Using the result of Theorem 4 we can support partial rank queries in $O(1)$ time and $O(m \log \log \sigma)$ bits.

Our data structure takes $O(m \log \log m)$ additional bits: Dictionaries $A'_i$ need $O(\log \log m)$ bits per symbol. Data structures $I_{a,t}$ and the structure for partial rank queries need $O(m \log \log \sigma)$ bits. We can reduce the space usage from $O(m \log \log m)$ to $O(m \log \log \sigma)$ using the same method as in Theorem 4.

\textbf{Compressed Index.} We mark nodes of the suffix tree $T$ using the method of Section A.5, but we set $d = \log \sigma$. Thus every $d$-th leaf is marked. Nodes of $T$ are classified into heavy, light, and special as defined in Section A.5. For every special node $u$, we construct a dictionary data structure $D_u$ that contains the labels of all heavy children of $u$. If there is child $u_j$ of $u$, such that the first symbol on the edge from to $u$ to $u_j$ is $a_j$, then we keep $a$ in $D_u$. For every $a_j \in D_u$ we store the index $j$ of the child $u_j$. If a heavy node $u$ has only one heavy child $u_j$ and more than $d$ light children, then we also store data structure $D_u$ for such a node $u$. If a heavy node has less $d$ children and one heavy child, then we keep the index of the heavy child using $O(\log d) = O(\log \log \sigma)$ bits.

The second component of our index is the Burrows-Wheeler Transform $B$ of the reverse text $\overline{T}$. We keep the data structure that supports partial rank, select, and access queries on $\overline{B}$. Using e.g., the result from [1], we can support access queries in $O(1)$ time while rank and select queries are answered in $O(\log \log \sigma)$ time. Moreover we construct a data structure, described in Lemma 6, that supports rank queries on a small interval in $O(1)$ time. We keep the data structure of Lemma 4 on $\overline{B}$; using this data structure, we can find in $O(1)$ time whether an arbitrary interval $\overline{B}[l..r]$ contains exactly one symbol. Finally we explicitly store answers to selected rank queries in the dictionary $D_u$. Let $\overline{B}[l_u..r_u]$ denote the range of $\overline{P}_u$ where $P_u$ is the string that corresponds to a node $u$ and $\overline{P}_u$ is the reverse of $P_u$. For all data structures $D_u$ and for every symbol $a \in D_u$ we store the values of $\text{rank}_a(l_u - 1, \overline{B})$ and $\text{rank}_a(r_u, \overline{B})$.

We will show later in this section that each rank value can be stored in $O(\log \sigma)$ bits. Thus $D_u$ needs $O(\log \sigma)$ bits per element. The total number of elements in all $D_u$ is equal to the number of special nodes plus the number of heavy nodes with one heavy child and at least $d$ light
children. Hence all $D_u$ contain $O((n/d))$ symbols and use $O((n/d) \log \sigma) = O(n)$ bits of space. Indexes of heavy children for nodes with only one heavy child can be kept in $O(\log \log \sigma)$ bits. Data structure that supports select, rank, and access queries on $B$ uses $nH_k(T) + o(n \log \sigma)$ bits. Auxiliary data structures on $B$ need $O(n) + O(n \log \log \sigma)$ bits. Finally we need $O(n \log n/d)$ bits to retrieve the position of a substring in $T$ in $O(d)$ time. Hence the space usage of our data structure is $nH_k(T) + o(n \log \sigma) + O(n) + O(n \log n/d)$.

It was already shown that we can compute the suffix tree and BWT in $O(n)$ time. We can also construct data structures for $B$ and data structures $D_u$ for special nodes of $T$ in linear time. We can traverse the tree and generate ranges $\overline{B}[l_u..r_u]$ in $O(1)$ time per node. If we store a data structure $D_u$ for some node, then we also answer queries $\text{rank}_u(l_u - 1, B)$ and $\text{rank}_u(r_u - 1, B)$ for all $a \in D_u$. Since a query is answered in time $O(\log \log \sigma)$ and the total number of elements in all $D_u$ is $O(n/d)$ all rank queries are answered in $O((n/d) \log \log \sigma) = o(n)$ time.

**Queries.** Given a query string $P$, we will find in time $O(|P| + \log \log \sigma)$ the range of the reversed string $P$ in $\overline{B}$. We will show below how to find the range of $\overline{P}[0..i]$ if the range of $\overline{P}[0..i-1]$ is known. Let $[l_j..r_j]$ denote the range of $\overline{P}[0..j]$, i.e., $\overline{P}[0..j]$ is the longest common prefix of all suffixes in $\overline{B}[l_j..r_j]$. We can compute $l_j$ and $r_j$ from $l_{j-1}$ and $r_{j-1}$ as $l_j = \text{Acc}[a] + \text{rank}_u(l_{j-1} - 1, B) + 1$ and $r_j = \text{Acc}[a] + \text{rank}_u(r_{j-1} - 1, B)$ for $a = P[j]$ and $j = 0, \ldots, |P|$. Here $\text{Acc}[f]$ is the accumulated frequency of the first $f - 1$ symbols. Using our auxiliary data structures of $B$ and additional information stored in nodes of the suffix tree $T$, we can answer necessary rank queries in constant time (with one exception). At the same time we traverse a path in the suffix tree $T$ until the locus of $P$ is found or a light node is reached. Additional information stored in selected tree nodes will help us answer rank queries in constant time. A more detailed description is given below.

Our procedure starts at the root node of $T$ and we set $l_{-1} = 0$, $r_{-1} = n - 1$, and $i = 0$. We compute the ranges $\overline{B}[l_i..r_i]$ that correspond to $\overline{P}[0..i]$ for $i = 0, \ldots, |P|$. Simultaneously we move down in the suffix tree until we reach a light node. Let $u$ denote the last visited node of $T$ and let $a = P[i]$. We denote by $u_a$ the next node that we must visit in the suffix tree, i.e., $u_a$ is the locus of $P[0..i]$. We can compute $l_i$ and $r_i$ in $O(1)$ time if $\text{rank}_u(r_{i-1} - 1, B)$ and $\text{rank}_u(l_{i-1} - 1, B)$ are known. We will show below that these queries can be answered in constant time because either (a) the answers to rank queries are explicitly stored in $D_u$ or (b) the rank query that must be answered is a small interval rank query. The only exception is the situation when we move from a heavy node to a light node in the suffix tree; in this situation the rank query takes $O(\log \log \sigma)$ time. For ease of description we distinguish between the following four cases.

(i) Node $u$ is a heavy node and $a \in D_u$. In this case we identify the heavy child $u_j$ of $u$ that is labeled with $a$. We can also find $l_i$ and $r_i$ in time $O(1)$ because $\text{rank}_u(l_{i-1} - 1, B)$ and $\text{rank}_u(r_{i-1} - 1, B)$ are stored in $D_u$.

(ii) Node $u$ is a heavy node and $a \notin D_u$ or we do not keep the dictionary $D_u$ for the node $u$. In this case $u$ has at most one heavy child and at most $d$ light children. If $u_a$ is a heavy node (case iia), then the leftmost occurrence of $a$ in $\overline{B}[l_{i-1}..r_{i-1}]$ is within $d^2$ symbols of $l_{i-1}$ and the rightmost occurrence of $a$ in $\overline{B}[l_{i-1}..r_{i-1}]$ is within $d^2$ symbols of $r_{i-1}$. Hence we can find $l_i$ and $r_i$ by answering small interval rank queries $\text{rank}_u(l_{i-1} - 1, B) + d^2$ and $\text{rank}_u(r_{i-1} - d^2, r_{i-1})$ respectively. If $u_a$ is a light node (case iib), we answer two standard rank queries on $\overline{B}$ in order to compute $l_i$ and $r_i$.

(iii) If $u$ is a light node, then $P[0..i - 1]$ occurs at most $d$ times. Hence $\overline{P}[0..i - 1]$ also occurs at
most $d$ times and $r_{i-1} - l_{i-1} \leq d$. Therefore we can compute $r_i$ and $l_i$ in $O(1)$ time by answering small interval rank queries.

(iv) We are on an edge of the suffix tree between a node $u$ and some child $u_j$ of $u$. In this case all occurrences of $P[0..i-1]$ are followed by the same symbol $a = P[i]$. Hence all occurrences of $P[0..i-1]$ are preceded by $P[i]$ in the reverse text. Therefore $B[l_{i-1}..r_{i-1}]$ contains only one symbol $a = P[i]$. In this case $\text{rank}_a(r_{i-1}, \overline{B})$ and $\text{rank}_a(l_{i-1} - 1, \overline{B})$ are partial rank queries and can be computed in $O(1)$ time.

In all cases, except for the case (iia), we can answer rank queries and compute $l_i$ and $r_i$ in $O(1)$ time. In case (iia) we need $O(\log \log \sigma)$ time answer rank queries. However case (iia) occurs only once when the pattern $P$ is processed: case (iia) only takes place when the node $u$ is heavy and its child $u_a$ is light. Hence the total time to find the range of $\overline{P}$ in $\overline{B}$ is $O(|\overline{P}| + \log \log \sigma)$ time. When the range is known, we can count and report all occurrences of $P$.

Construction Algorithm. We can construct the suffix tree $T$ and the BWT $\overline{B}$ in $O(n)$ deterministic time. Then we can visit all nodes of $T$ and identify all nodes $u$ for which the data structure $D_u$ must be constructed. We keep information about nodes for which $D_u$ will be constructed in a bit vector. For every such node we also store the list of its heavy children with their labels. To compute additional information for $D_u$, we traverse the nodes of $T$ one more time using a variant of depth-first search. When a node $u \in cT$ is reached, we know the interval $[l_u, r_u]$ of $\overline{s_u}$ in $\overline{B}$, where $s_u$ is the string that labels the path from the root to a node $u \in T$. We generate the list of all children $u_i$ of $u$ and their respective labels $a_i$. If we store a data structure $D_u$ for the node $u$, we identify labels $a_h$ of heavy children $u_h$ of $u$. For every $a_h$ we compute $\text{rank}_{a_h}(l_u - 1, \overline{B})$ and $\text{rank}_{a_h}(r_u, \overline{B})$ and add this information to $D_u$. Then we generate the intervals that correspond to all strings $\overline{s_u a_i}$ in $\overline{B}$ and keep them in a list $\text{List}(u)$. Since intervals in $\text{List}(u)$ are disjoint, we can store $\text{List}(u)$ in $O(\sigma \log n)$ bits.

We can organize our last traversal in such way that only $O(\log n)$ lists $\text{List}(u)$ need to be stored. Let $\text{num}(u)$ denote the number of leaves in the subtree of a node $u$. We say that a node is small if $\text{num}(u_i) \leq \text{num}(u)/2$ and big otherwise. Every node can have at most one big child. When a node $u$ processed and $\text{List}(u)$ is generated, we visit small children $u_i$ of $u$ in arbitrary order. When all small children $u_i$ are visited and processed, we discard the list $L(u)$. Finally, if $u$ has a big child $u_b$, we visit $u_b$. If a node $u$ is not the root node and we keep $\text{List}(u)$, then $\text{num}(u) \leq \text{num}(\text{parent}(u))/2$. Therefore we keep $\text{List}(u)$ for at most $O(\log n)$ nodes $u$. Thus the space we need to store all $\text{List}(u)$ is $O(\sigma \log^2 n) = o(n)$ for $\sigma \leq n^{1/2}$. Hence the total workspace used of our algorithm is $O(n \log \sigma)$. The total number of rank queries that we need to answer is $O(n/d)$ because all $D_u$ contain $O(n/d)$ elements. We need $O((n/d) \log \log \sigma)$ time to construct all $D_u$ and to answer all rank queries. The total time needed to traverse $T$ and collect necessary data about heavy nodes and special nodes is $O(n)$. Therefore our index can be constructed in $O(n)$ time.

It remains to show how we can store selected precomputed answers to rank queries in $O(\log \sigma)$ bits per query. We divide the sequence $\overline{B}$ into chunks of size $\sigma^2$. For each chunk and for every symbol $a$ we encode the number of $a$’s occurrences per chunk in a binary sequence $A_a$, $A_a = 1^{d_1}01^{d_2}0\ldots1^{d_1}0\ldots$ where $d_i$ is equal to the number of times $a$ occurs in the $i$-th chunk. If a symbol $\overline{B}[i]$ is in the chunk $Ch$, then we can answer $\text{rank}_a(i, \overline{B})$ by $O(1)$ queries on $A_a$ and a rank query on $Ch$; see e.g., [14]. Suppose that we need to store a pre-computed answer to a query $\text{rank}_a(i, \overline{B})$;
we store the answer to rank$_a(i', Ch)$ where $Ch$ is the chunk that contains $i$ and $i'$ is the relative position of $\mathcal{B}[i]$ in $Ch$. Since a chunk contains $\sigma^2$ symbols, rank$_a(i' Ch) \leq \sigma^2$ and we can store the answer to rank$_a(i', Ch)$ in $O(\log \sigma)$ bits. When the answer to the rank query on $Ch$ is known, we can compute the answer to rank$_a(i, \mathcal{B})$ in $O(1)$ time.