Theoretical Computer Science 410 (2009) 4402–4413

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Dynamic rank/select structures with applications to run-length encoded texts☆

Sunho Lee, Kunsoo Park*

School of Computer Science and Engineering, Seoul National University, Seoul, 151-742, South Korea

ARTICLE INFO

Keywords:
Succinct data structures
Dynamic rank/select structures
Full-text index
Run-length encoding

ABSTRACT

Given an n-length text over a σ-size alphabet, we propose a framework for dynamic rank/select structures on the text and some of its applications. For a small alphabet with $\sigma \leq \log n$, we propose a two-level structure consisting of a counting scheme and a storing scheme that supports $O(\log n)$ worst-case time *rank*/*select* operations and $O(\log n)$ amortized time *insert*/*delete* operations. For a large alphabet with $\log n < \sigma \leq n$, we extend it to obtain $O((1+\frac{\log\sigma}{\log\log n})\log n)$ worst-case time *rank*/*select* and $O((1+\frac{\log\sigma}{\log\log n})\log n)$ amortized time *insert*/*delete*. Our structure provides a simple representation of an index for a collection of texts. In addition, we present rank/select structures on run-length encoding (RLE) of a text. For the n'-length RLE of an n-length text, our static version provides $O(1)$ time *select* and $O(\log\log\sigma)$ time *rank* using $n'\log\sigma + O(n)$ bits and our dynamic version gives $O((1+\frac{\log\sigma}{\log\log n})\log n)$ time operations in $n'\log\sigma + o(n'\log\sigma) + O(n)$ bits.

1. Introduction

Given a text T of length n over a σ-size alphabet, a rank/select structure answers the following queries.

- $rank_T(c, i)$: counts the number of character c's up to position i in T.
- $select_T(c, k)$: finds the position of the kth c in T.

For a dynamic structure, we consider the following insert and delete operations on T in addition to $rank_T$ and $select_T$.

- $insert_T(c, i)$: inserts character c between $T[i]$ and $T[i+1]$.
- $delete_T(i)$: deletes $T[i]$.

A rank/select structure is an essential ingredient of compressed full-text indices such as compressed suffix array (CSA) [1,2] and the FM-index [3]. The rank/select structure occupies only the same space as the text T, $n\log\sigma$ bits, plus a small extra space, $o(n\log\sigma)$ bits, and it can be compressed into an even smaller space, $nH_k + o(n\log\sigma)$ bits [4,5], where H_k is the empirical kth order entropy of T [6].

The rank/select structures on binary strings were first introduced by Jacobson [7] and further improved by Clark [8] and Munro [9]. See Table 1. Especially, Raman et al. achieved a compressed version of $nH_0 + o(n)$ bits with $O(1)$ time *rank*/*select* [10]. Golynski et al. [11] and Patrascu [12] improved the $o(n)$-bits term in space. Sadakane and Grossi [4], and Ferragina and

☆ A preliminary version of this paper appeared in *Proceedings of the 18-th Annual Symposium on Combinatorial Pattern Matching (CPM 2007)*. This work was supported by Grants FPR08-A1-021 of the 21C Frontier Functional Proteomics Project from the Korean Ministry of Education, Science & Technology.

* Corresponding author.
E-mail addresses: shlee@theory.snu.ac.kr (S. Lee), kpark@theory.snu.ac.kr (K. Park).

0304-3975/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2009.07.021

Table 1
Static rank/select structures.

Alphabet	Text	Time	Space	Reference
Binary ($\sigma = 2$)	Plain	$O(\log n)$	$n + o(n)$	Jacobson [7]
		$O(1)$	$n + o(n)$	Clark [8], Munro [9]
		$O(1)$	$nH_0 + o(n)$	Raman, Raman and Rao [10] Golynski, Grossi, Gupta, Raman and Rao [11] Patrascu[12]
		$O(1)$	$nH_k + o(n)$	Sadakane and Grossi [4] Ferragina and Venturini [5]
polylog(n)	Plain	$O(1)$	$nH_0 + o(n)$	Ferragina, Manzini, Mäkinen and Navarro [13]
$O(n^\beta)$, $\beta < 1$		$O(\frac{\log\sigma}{\log\log n})$	$nH_0 + o(n\log\sigma)$	
General	Plain	$O(\log\sigma)$	$nH_0 + o(n\log\sigma)$	Grossi, Gupta and Vitter [14]
		$O(1)$ select $O(\log\log\sigma)$ rank	$n\log\sigma + O(n)$	Hon, Sadakane and Sung [15]
		$O(1)$ select $O(\log\log\sigma)$ rank	$nH_0 + O(n)$	Golynski, Munro and Rao [16]
		$o((\log\log n)^3)$	$nH_k + o(n\log\sigma)$	Barbay, He, Munro and Rao [17]
	RLE	$O(\log\log\sigma)$ rank	$n'H_0' + O(n)$	Mäkinen and Navarro [18]
		$O(1)$ select $O(\log\log\sigma)$ rank	$n'H_0' + O(n)$	This paper

Table 2
Dynamic rank/select structures (for a large alphabet).

Text	Time	Space	Reference
Plain	$O(\log\sigma\frac{\log n}{\log\log n})$	$n\log\sigma + o(n\log\sigma)$	Raman, Raman and Rao [19]
	$O(\log\sigma\log_b n)$ rank/select $O(\log\sigma\, b)$ flip		
	$O(\log\sigma\log_b n)$ rank/select $O(\log\sigma\, b)$ insert/delete	$n\log\sigma + o(n\log\sigma)$	Hon, Sadakane and Sung [20]
	$O(\log\sigma\log n)$	$nH_0 + o(n\log\sigma)$	Mäkinen and Navarro [21]
	$O((1/\epsilon)\log\log n)$ rank/select $O((1/\epsilon)n^\epsilon)$ insert/delete	$n\log\sigma + o(n\log\sigma)$	Gupta, Hon, Shah and Vitter [22]
	$O((1+\frac{\log\sigma}{\log\log n})\log n)$	$nH_0 + o(n\log\sigma)$	González and Navarro [23]
	$O((1+\frac{\log\sigma}{\log\log n})\log n)$	$n\log\sigma + o(n\log\sigma)$	This paper
		$nH_0 + O(n) + o(n\log\sigma)$	Lee and Park [24]
RLE	$O((1+\frac{\log\sigma}{\log\log n})\log n)$	$n'\log\sigma + o(n'\log\sigma) + O(n)$	This paper

Venturini [5] proposed schemes to compress a general sequence into H_k entropy bounds and to access its subsequences in $O(1)$ time, so these schemes compress any rank/select structure into $nH_k + o(n)$ bits.

There are two ways of extending rank/select structures on binary strings. One is to provide *rank*/*select* on texts over a σ-size alphabet with $2 < \sigma \leq n$ and the other is to support updates of texts. For *rank*/*select* on texts over a σ-size alphabet, Grossi et al.'s wavelet tree [14] and its improvement by Ferragina et al. [13] are general frameworks which transform any k-size alphabet rank/select structures to a σ-size alphabet rank/select structures with $\log_k\sigma$ slowdown factor in query time. Not using wavelet trees, the structures by Golynski et al. [16] and by Hon et al. [15] gave $O(1)$ time *select* and $O(\log\log\sigma)$ time *rank*. Recently, Barbay et al. obtained $o((\log\log\sigma)^3)$ time *rank*/*select* in $nH_k + o(n\log\sigma)$ bits [17].

For supporting updates of bit strings, the problem was addressed as a special case of the dynamic partial sum problem by Raman et al. [19] and Hon et al. [20]. Their rank/select structures take $n + o(n)$ bits and provide $O(\frac{\log n}{\log\log n})$ worst-case time *rank*/*select* and $O(\log n)$ amortized time updates. The first entropy-bound structure of $nH_0 + o(n)$ bits was proposed by Mäkinen and Navarro [21], which guarantees $O(\log n)$ worst-case time for all operations.

For dynamic structures on σ-size alphabet texts, the extensions of dynamic versions on bit strings provide solutions with $O(\log\sigma)$ slowdown factor by binary wavelet trees as in Table 2. For example, the extension of Mäkinen and Navarro's structure supports $O(\log\sigma\log n)$ time operations in $nH_0 + o(n\log\sigma)$ bits. Recently, Gupta et al. presented a dynamic structure of $n\log\sigma + o(n\log\sigma)$ bits, which provides $O((1/\epsilon)\log\log n)$ time *rank*/*select* and $O((1/\epsilon)n^\epsilon)$ time updates without wavelet trees [22].

In this paper we propose a framework for dynamic rank/select structures on texts over a σ-size alphabet and some of its applications. Our contributions are as follows.

- For a small alphabet with $\sigma \leq \log n$: To obtain $O(\log n)$ time operations in $n\log\sigma + o(n\log\sigma)$ bits, we propose a two-level dynamic structure consisting of a counting scheme and a storing scheme. The counting scheme and the storing scheme were tightly coupled in Mäkinen and Navarro's dynamic structure [21]. By separating these two schemes, we

get a simple description of dynamic rank/select structures, and an independent improvement of each scheme is possible. We combine counting schemes in static structures [15,16] with a simplified Mäkinen and Navarro's structure only for the storing scheme, and obtain $O(\log n)$ worst-case time *rank*/*select* and $O(\log n)$ amortized time *insert*/*delete*.

- For a large alphabet with $\log n < \sigma \leq n$: We extend our $\log n$-size alphabet version by the k-ary wavelet tree and obtain $O((1+\frac{\log\sigma}{\log\log n})\log n)$ worst-case time *rank*/*select* and $O((1+\frac{\log\sigma}{\log\log n})\log n)$ amortized time *insert*/*delete* in $n\log\sigma + o(n\log\sigma)$ bits. This is the first application of the k-ary wavelet tree to dynamic structures, though it was a well-known technique in static structures [13].
- Application to an index for a collection of texts: We show that any dynamic rank/select structure on Burrows–Wheeler Transform (BWT) replaces Chan et al.'s index for a collection of texts [25]. Hence, our dynamic rank/select structures provide a simple and efficient representation of Chan et al.'s structure.
- Dynamic Run-Length based FM-index (RLFM): We apply our structures to the n'-length RLE of an n-length text so that the size of structures is reduced to $n'\log\sigma + o(n'\log\sigma) + O(n)$ bits. Thus, we obtain a full version (supporting *rank*, *select*, *insert*, and *delete*) of Mäkinen and Navarro's RLFM [18] that was a static *rank* structure with $O(n)$ bits. This technique also reduces the size of static structures to $n'\log\sigma + O(n)$ bits or $n'H_0' + O(n)$ bits depending on base rank/select structures, where H_0' is the entropy of the sequence of run characters in RLE. For the n'-length RLE of an n-length text, our static version provides $O(1)$ time *select* and $O(\log\log\sigma)$ time *rank* and our dynamic version gives $O((1+\frac{\log\sigma}{\log\log n})\log n)$ worst-case time *rank*/*select* and $O((1+\frac{\log\sigma}{\log\log n})\log n)$ amortized time *insert*/*delete*. See Tables 1 and 2.

The rest of this paper is organized as follows. Section 2 introduces some definitions and preliminaries. Section 3 gives our dynamic rank/select structure for a $\log n$-size alphabet and its extension for a σ-size alphabet. Section 4 shows how to simplify the index of texts collection by the rank/select structures. Section 5 applies the rank/select structures to RLE of texts, and we conclude in Section 6.

Note. Recently, González and Navarro have obtained an improved result which achieves worst-case $O((1+\frac{\log\sigma}{\log\log n})\log n)$ time in updates and compressed space of $nH_0 + o(n\log\sigma)$ bits [23]. However, our framework such as separation of the counting scheme from the storing scheme and application of the k-ary wavelet tree is used in [23]. González and Navarro proposed a novel counting scheme to guarantee worst-case time in updates.

They also point out that our counting scheme has $O(n)$-bits vectors, so $O(n)$ bits dominate the whole space when σ is a small constant or T is highly compressible such that $H_0 = o(1)$. However, our $O(n)$-bits vectors are conceptual and the actual sizes achieve the information theoretical lower bounds. We clearly show that the lower bound of our counting scheme is $o(n)$ for a $\log n$-size alphabet, and therefore an $o(n)$-bits term is automatically achieved by using Mäkinen and Navarro's compressed binary rank/select structures [21,26]. This is an advantage of separating the counting scheme from the storing scheme.

Recently, we also obtained a compressed storing scheme of $nH_0 + O(n) + o(n\log\sigma)$ bits [24], in which the $O(n)$-bits term is made by gap-encoding. González and Navarro use block-identifier encoding of $nH_0 + o(n\log\sigma)$ bits to remove $O(n)$ bits in space.

2. Definitions and preliminaries

We denote by $T = T[1]T[2]\ldots T[n]$ the text of length n over a σ-size alphabet $\Sigma = \{0, 1, \ldots, \sigma-1\}$. We assume that the characters in a σ-size alphabet are $0, 1, \ldots, \sigma-1$ and their lexicographic order is the order of their values. We assume the alphabet size σ is $o(n)$, because otherwise the size of text becomes $n\log\sigma = \Omega(n\log n)$ bits; $\Omega(n\log n)$ bits space makes the problem easy.

We assume the RAM model with a word of $w = \Theta(\log n)$ bits size, which supports addition, multiplication, and bitwise operations in $O(1)$ time. Our RAM model allows us to access the memory with pointers of $O(\log n)$ bits size. We regard a character $c \in \Sigma$ as a $\log\sigma$ bits number, so one word contains $w_\sigma = \Theta(\frac{\log n}{\log\sigma})$ characters.

Our rank/select structure for a σ-size alphabet is built over a rank/select structure on dynamic bit vectors. For this binary rank/select structure, we employ Mäkinen and Navarro's structure (MN-structure) [21,26] as a black box, which gives $O(\log n)$ worst-case time operations in $nH_0 + o(n)$ bits.

Theorem 1 ([21,26]). *Given an n-length bit vector B, there is a rank/select structure that supports rank, select, insert and delete in $O(\log n)$ worst-case time and $nH_0 + o(n)$ bits.*

Note that the nH_0 term is from the information theoretical lower bounds of an n-length bit vector with m 1s, i.e., $\log\binom{n}{m} \leq m\log\frac{n}{m} \leq nH_0$ [10]. The MN-structure achieves this lower bound with additional $o(n)$ bits.

3. Dynamic rank/select structures for texts

In this section we present our techniques for dynamic rank/select structures. For a small alphabet with $\sigma \leq \log n$, we present a two-level dynamic structure with a counting scheme and a storing scheme to support all operations in $O(\log n)$ time. In the MN-structure, the structure for counting bits and that for storing bits were tightly coupled. We separate these

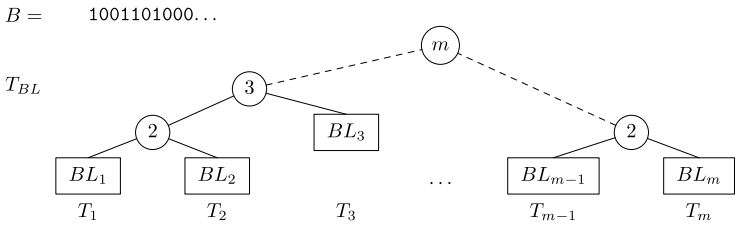

Fig. 1. Layout of our structure.

two structures to handle the operations more effectively. For a large alphabet with $\sigma > \log n$, we extend our small-alphabet structure by using a k-ary wavelet tree. Based on our techniques, we achieve the following result.

Theorem 2. *Given a text T over an alphabet of size σ, there is a dynamic structure that supports $O((1 + \frac{\log \sigma}{\log \log n}) \log n)$ worst-case time rank, select and access, while supporting $O((1+\frac{\log \sigma}{\log \log n}) \log n)$ amortized time insert and delete in $n \log \sigma + o(n \log \sigma)$ bits.*

3.1. Dynamic rank/select structures for a small alphabet

Our small-alphabet structure employs the memory layout of the MN-structure: word, sub-block, and block. For a binary alphabet, a word contains $w = \Theta(\log n)$ bits but our word contains $w_\sigma = \Theta(\frac{\log n}{\log \sigma})$ characters. A sub-block consists of $\sqrt{\log n}$ words. A block is a list of $\frac{1}{2}\sqrt{\log n}$ sub-blocks to $2\sqrt{\log n}$ sub-blocks, so that the block contains $\frac{1}{2}\log n$ to $2 \log n$ words.

Given an n-length text T, we partition T into m substrings with the lengths from $\frac{1}{2}w_\sigma \log n$ to $2w_\sigma \log n$ and store each substring in a block. We denote by T_b the bth substring and by BL_b the bth block storing T_b, for $1 \le b \le m$ (Fig. 1). For the set of starting positions of the substrings, $P = \{p_1, p_2, \ldots, p_m\}$, we represent the partition as an n-length bit vector I, where $I[p_b] = 1$ for $1 \le b \le m$ and all other bits are 0s. Then, I gives a mapping between a position and a block number. Given block number b, $select_I(1, b)$ is the starting position of T_b in T. For position i, there are block number b of the substring containing $T[i]$ and the offset r of $T[i]$ in T_b, which are computed by $b = rank_I(1, i)$ and $r = i - select_I(1, b) + 1$.

Using partition vector I, we divide a given operation into an over-block operation and an in-block one. For instance, given $rank_T(c, i)$, the over-block *rank* counts the number of occurrences of c before T_b and the in-block *rank* returns the number of c in $T_b[1..r]$. Then, the sum of these over-block *rank* and in-block *rank* answers $rank_T(c, i)$. We define the over-block operations as

- $rank\text{-}over_T(c, b)$: gives the number of c's in blocks preceding the bth block.
- $select\text{-}over_T(c, k)$: gives the number of the block containing the kth c.
- $insert\text{-}over_T(c, b)$: updates counting information by the insertion of c in T_b.
- $delete\text{-}over_T(c, b)$: updates counting information by the deletion of c from T_b.

The in-block operations for the bth block are defined as

- $rank_{T_b}(c, r)$: gives the number of c's in $T_b[1..r]$.
- $select_{T_b}(c, k)$: gives the position of the kth c in T_b.
- $insert_{T_b}(c, r)$: inserts c between $T_b[r]$ and $T_b[r + 1]$.
- $delete_{T_b}(r)$: deletes $T_b[r]$.

Note that the over-block updates just change the structure for counting occurrences of a character, and the in-block updates actually change the text itself.

3.1.1. Data structures and retrieving operations

Over-block structures. For an over-block counting scheme, we employ a bit string B used in static rank/select structures such as [15,16]. We here present it for a complete description and we will later show how to update it for our dynamic structures.

Given a text T, let $(T[i], b)$ be the character-and-block pair of $T[i]$, where the block number b of $T[i]$ is given by $rank_I(1, i)$. We define $CB(T)$ as the sorted sequence of $(T[i], b)$ by the order of its character $T[i]$ and block number b. See Fig. 2. Let a c-group be the sequence of pairs (c, b) with the same character c in $CB(T)$. Then, the block numbers in a c-group are non-decreasing from 1 to m. In other words, the c-group preserves the order of occurrences of c in T.

We encode $CB(T)$ into bit vector B as follows. For each pair (c, b), $0 \le c \le \sigma - 1$ and $1 \le b \le m$, $CB(T)$ has the consecutive occurrences of the same (c, b) pairs, which means the occurrences of c in T_b. The bit vector B represents each pair (c, b) as a 1 and the number of its occurrences as the following 0s. If there is no occurrence of (c, b), it is represented as a single 1. Note that the encoding of a c-group is also grouped in B, which starts from the $(cm + 1)$st 1 of B. The encoding of a c-group will also be called a c-group of B. See Fig. 2.

$$
\begin{array}{rcl}
T & = & \underline{\texttt{babc}}\ \underline{\texttt{abab}}\ \underline{\texttt{abca}} \\
I & = & \underline{1000}\ \underline{1000}\ \underline{1000} \\
 & & \underline{(\texttt{b},1)(\texttt{a},1)(\texttt{b},1)(\texttt{c},1)}\quad \underline{(\texttt{a},2)(\texttt{b},2)(\texttt{a},2)(\texttt{b},2)}\quad \underline{(\texttt{a},3)(\texttt{b},3)(\texttt{c},3)(\texttt{a},3)} \\
 & & \\
CB(T) & = & (\texttt{a},1)(\texttt{a},2)(\texttt{a},2)(\texttt{a},3)(\texttt{a},3)\quad (\texttt{b},1)(\texttt{b},1)(\texttt{b},2)(\texttt{b},2)(\texttt{b},3)\quad (\texttt{c},1)(\texttt{c},3) \\
B & = & 10100100\ 10010010\ 10110
\end{array}
$$

Fig. 2. Example of $CB(T)$ and B.

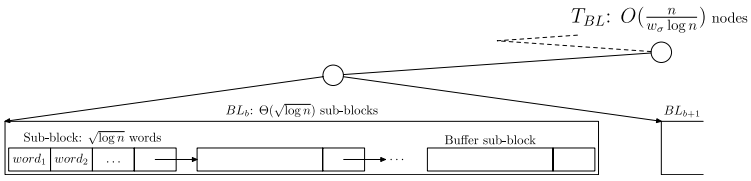

Fig. 3. Layout of in-block structures.

Now we show the lower bounds of our counting scheme, bit vectors I and B. Since we partition T into $\Omega(\sigma)$-size blocks, we will obtain $o(n)$ bits rather than $O(n)$ bits in static structures [15,16]. The block size varies from $\frac{1}{2}\log n$ to $2\log n$ words, so the number of substrings $m \leq \frac{2n}{w_\sigma \log n} = O(\frac{n \log \sigma}{\log^2 n})$. Thus, I is an n-length bit vector with m 1s. The lower bound of I is $o(n)$ as follows.

$$
\begin{aligned}
\log \tbinom{n}{m} &\leq m \log \frac{n}{m} \\
&= O\left(\frac{n \log \sigma}{\log^2 n}\right) \cdot \log\left(\frac{\log^2 n}{\log \sigma}\right) \\
&= O\left(\frac{n \log\log n}{\log^2 n} \cdot \log\log n\right) \quad \text{for } \sigma \leq \log n.
\end{aligned}
$$

In the bit vector B, the number of 0s is $n_0 = n$ and the number of 1s is the number of (c, b) pairs, i.e., $n_1 = \sigma m$. Hence, the lower bound of B is also $o(n)$ as follows.

$$
\begin{aligned}
\log \tbinom{n_0+n_1}{n_1} &\leq n_1 \log \frac{n_0 + n_1}{n_1} \\
&= O\left(\frac{\sigma n \log \sigma}{\log^2 n}\right) \cdot \log\left(1 + \frac{\log^2 n}{\sigma \log \sigma}\right) \\
&= O\left(\frac{n \log\log n}{\log n} \cdot \log\log n\right) \quad \text{for } \sigma \leq \log n.
\end{aligned}
$$

The sizes of the MN-structures on B and I achieve these lower bounds of $o(n)$ bits [21,26].

In-block structures. Our in-block structures keep the substrings of T and we process in-block rank/select queries on T_b by scanning T_b, word-by-word. We use a simplified MN-structure only as a storing scheme to keep blocks of texts over a $\log n$-size alphabet. We define the components of the storing scheme for a $\log n$-size alphabet as follows.

- BL_b: The bth block that is a linked list of $\frac{1}{2}\sqrt{\log n}$ to $2\sqrt{\log n}$ sub-blocks. A sub-block is a chunk of $\sqrt{\log n}$ words containing $w_\sigma \sqrt{\log n}$ characters. See Fig. 3. BL_b reserves one extra sub-block as a buffer for updates. We count additional space except the text as follows. Since there is an $O(\log n)$-bits pointer per sub-block in the linked list representation, we use total $\frac{2n}{w_\sigma \sqrt{\log n}} \cdot O(\log n) = O(\frac{n \log \sigma}{\sqrt{\log n}})$ bits for all lists. The total space of buffer sub-blocks is also $\frac{2n}{w_\sigma \log n} \cdot w\sqrt{\log n} = O(\frac{n \log \sigma}{\sqrt{\log n}})$ bits.
- T_{BL}: A red-black tree with $BL_1, BL_2, \ldots, BL_m$ as leaf nodes. An internal node v maintains $n(v)$, the number of blocks in the subtree rooted at v. Given block number b, we can access the bth block, BL_b, by traversing the path from the root node. The size of T_{BL} is $O(\frac{2n}{w_\sigma \log n}) \cdot O(\log n) = O(\frac{n \log \sigma}{\log n})$ bits, because T_{BL} has $O(\frac{2n}{w_\sigma \log n})$ nodes which contain $n(v)$ and pointers of $O(\log n)$ bits.
- J_R: The zero-rank table on a half word of all possible $\frac{1}{2}w_\sigma$ characters. Each entry of J_R returns the number of $0 \in \Sigma$ in each of $\frac{1}{2}w_\sigma$ prefixes of a half word. The size of J_R is $\sigma^{w_\sigma/2} \cdot w_\sigma/2 \cdot \log w_\sigma = O(\sqrt{n} \log n \log\log n) = o(n)$ bits. For the rank on a word, we look up J_R twice.

Rank/select queries. Using our over-block and in-block structures, we process rank/select queries in $O(\log n)$ worst-case time. The details are described in Fig. 4. We reduce *rank-over* and *select-over* to binary *rank*/*select* on B. The in-block

$rank_T(c,i)$	$select_T(c,k)$
1. Find the block number b and the offset r of i: $b = rank_I(1,i)$, $r = i - select_I(1,b)+1$	1. Get the number of characters less than c, $F[c]$: $F[c] = rank_B(0, select_B(1, cm+1))$
2. Get the number of characters less than c, $F[c]$: $F[c] = rank_B(0, select_B(1, cm+1))$	2. $select\text{-}over_T(c,k)$: count the 1s before the k-th 0 in the c-group of B: $select\text{-}over_T(c,k) = select_B(0,F[c]+k) - (F[c]+k) - cm$
3. $rank\text{-}over_T(c,b)$: count the 0s before the b-th 1 in the c-group of B. $p_b = select_B(1, cm+b)$, $rank\text{-}over_T(c,b) = rank_B(0,p_b) - F[c]$	3. Get the block number b' and the number k' of remaining characters: $b' = select\text{-}over_T(c,k)$, $k' = k - rank\text{-}over(c,b')+1$
4. $rank_{T_b}(c,r)$: scan the words in BL_b to count c in $T_b[1..r]$.	4. $select_{T_{b'}}(c,k')$: scan the words in $BL_{b'}$ to find the k'-th c in $T_{b'}$.
5. $rank_T(c,i) = rank\text{-}over_T(c,b) + rank_{T_b}(c,r)$	5. $select(c,k) = select_I(1,b') + select_{T_{b'}}(c,k')$

Fig. 4. The steps of rank/select queries.

operations of $rank_{T_b}$ and $select_{T_b}$ are processed by a simple scanning of $O(\log n)$ words in BL_b. We start from the following $O(1)$ time *rank* on a word-size string.

Lemma 1. *The rank of $c \in \Sigma$ on a word of $w_\sigma = \Theta(\frac{\log n}{\log \sigma})$ characters can be computed in $O(1)$ time and $o(n)$ bits.*

Proof. We first convert *rank* of c on a word x to *rank* of the zero character ($0 \in \Sigma$). That is, we convert the occurrences of c to zeros and other occurrences to non-zeros. We assume a pre-computed constant $M = 1^{w_\sigma}$, which is w_σ repetitions of character $1 \in \Sigma$. We multiply M by c and obtain $cM = c^{w_\sigma}$. We apply an XOR operation between c^{w_σ} and x. The value of c^{w_σ} XOR x contains $0 \in \Sigma$ from c and non-zeros from others. After this conversion, J_R answers *rank* of c by *rank* of $0 \in \Sigma$. □

The binary *rank*/*select* on B for over-block counting are the same as in the static structures. For $rank\text{-}over_T(c,b)$, we count the number of 0s in the c-group of B that represent the occurrences of the pairs (c,j) such that $j < b$. We first find the 1 that represents the pair (c,b) in the c-group. This 1 is the $(cm+b)$th 1 of B. Then, we count the number of 0s up to the $(cm+b)$th 1 and subtract $F[c]$ that is the number of characters less than c in T.

$$F[c] = rank_B(0, select_B(1, cm+1))$$
$$rank\text{-}over_T(c,b) = rank_B(0, select_B(1, cm+b)) - F[c].$$

Let us consider the example of $rank\text{-}over_T(\mathtt{b}, 3)$ in Fig. 2. In this example, $\Sigma = \{\mathtt{a},\mathtt{b},\mathtt{c}\} = \{0,1,2\}$ and $m = 3$. For $rank\text{-}over(\mathtt{b}, 3)$, we count the number of 0s that represent b before the third 1 in the b-group. This third 1 is given by $select_B(1, 1\cdot m+3) = 15$. Then, $rank\text{-}over_T(\mathtt{b}, 3) = 4$ is obtained from $rank_B(0, 15) = 9$ by subtracting the number of a in T, $F[\mathtt{b}] = 5$.

The $select\text{-}over(c,k)$ is the block number of the kth c pair. Recall that in the c-group of B the kth c is represented as the kth 0 and its block number as the number of 1s up to that 0. Because the number of 0s before the c-group is $F[c]$, we obtain the block number by counting 1s between the $F[c]$th 0 and the $(F[c]+k)$th 0. Note that the number of 1s before the xth 0 is $select_B(0,x) - x$ and the number of 1s before the $F[c]$th 0 is cm. Hence,

$$select\text{-}over_T(c,k) = select_B(0, F[c]+k) - (F[c]+k) - cm.$$

For the example of $select\text{-}over_T(\mathtt{b}, 4)$ in Fig. 2, we count the number of 1s before the fourth 0 of the b-group, which is $select_B(0, F[\mathtt{b}]+4) - (F[\mathtt{b}]+4) = 5$. Then, we get $select\text{-}over_T(\mathtt{b}, 4) = 5 - 1\cdot m = 2$ by subtracting the number of 1s that represent the a pairs.

Lemma 2. *Given an n-length text over a σ-size alphabet with $\sigma \le \log n$ and its partition of substrings with the lengths from $\frac{1}{2}w_\sigma \log n$ to $2w_\sigma \log n$, the rank-over and select-over are processed in $O(\log n)$ worst-case time and $o(n)$ bits.*

Finally, we obtain $rank_T(c,i)$ as the sum of $rank\text{-}over(c,b)$ and $rank_{T_b}(c,r)$, where b is the block number of position i and r is its offset. Similarly, $select_T(c,k)$ is the sum of the position of $T_{b'}$ and the offset $select_{T_{b'}}(c,k')$, where $b' = select\text{-}over(c,k)$ and $k' = k - rank\text{-}over(c,b') + 1$. We also process an accessing of the ith character, $access_T(i)$, in $O(\log n)$ time by scanning T_b that contains the ith character.

Lemma 3. *Given an n-length text over a σ-size alphabet with $\sigma \le \log n$, the rank, select, and access are processed in $O(\log n)$ worst-case time and $n\log\sigma + O(\frac{n\log\sigma}{\sqrt{\log n}})$ bits.*

In-block updates of $T_b[r]$	**Over-block updates** ($insert\text{-}over_T(c, b)$)
1. Find the BL_b by traversing T_{BL}: For each node v, compare $n(v)$ and b to choose the subtree containing BL_b.	1. Update the length of T_b: $$insert_I(0, select_I(1, b))$$
2. Propagate carry characters in the words in BL_b: – Shift carry to the last sub-block (insertion) or from the last sub-block (deletion) – Check whether the last sub-block is full or empty, and allocate or remove sub-blocks.	2. Update the c-group in B: $$insert_B(0, select_B(1, cm + b))$$
3. If BL_b becomes out of the range, $\frac{1}{2}\sqrt{\log n}$ to $2\sqrt{\log n}$ sub-blocks: – Split BL_b (insertion) or merge it with its neighbor (deletion). – Traverse T_{BL} from the leaf of BL_b to the root and update $n(v)$ in internal nodes.	3. If BL_b overflows and splits, reflect this split of T_b at position sp: – Split the length of T_b: mark 1 at position $select_I(1, b) + sp$ – Split the b-th blocks for all c-group: $$insert_B(1, select(1, cm + b) + rank_{T_b}(c, sp))$$

Fig. 5. The steps of in-block/over-block updates.

3.1.2. Updating operations

For insert and delete operations, we present updates of the in-block and over-block structures in amortized $O(\log n)$ time. Our description is bottom-up: word updates, in-block updates, and over-block updates. An insertion or deletion on a word triggers a carry character to the neighbor words in BL_b. The process of in-block updates is propagation of the carry characters in BL_b. This change of BL_b induces the updates of the over-block encoding, B. We first show the following $O(1)$ time insert/delete operations on a word-size text.

Lemma 4. *It takes $O(1)$ time to process insert or delete on a word of w_σ characters.*

Proof. We combine shifting and masking to insert or delete c in word x of w_σ characters. Given *insert* or *delete* of c at position r, we split x into two parts: x_1 of the left r characters and x_2 of the other characters. We use a constant $M = z^{w_\sigma}$ that is w_σ repetitions of a mask z consisting of $\log\sigma$ 1 bits. $M_1 = z^r 0^{(w_\sigma - r)}$ is a left-shift of M by $w_\sigma - r$ characters and $M_2 = 0^r z^{(w_\sigma - r)}$ is a right-shift of M by r characters. Then, we split x into $x_1 = M_1$ AND x and $x_2 = M_2$ AND x. We update x_1 by adding c or deleting the rth character. The other part x_2 is shifted by the case of insertion or deletion, which triggers a carry character to neighbor words. Then, the updated x_1 merges with the shifted x_2 by the OR operation. □

In-block updates. We show how to update the components, BL_b and T_{BL}, in $O(\log n)$ worst-case time. For $insert_{T_b}(c, r)$ or $delete_{T_b}(r)$, carry characters propagate from the word containing $T_b[r]$ to a word in the buffer sub-block of BL_b. This propagation updates $\frac{1}{2}\log n$ to $2\log n$ words of BL_b in $O(\log n)$ worst-case time by Lemma 4. If the buffer sub-block is full, then we create a new sub-block and add it to BL_b as a new buffer. If the last two sub-blocks become empty, then we remove one of empty sub-blocks from BL_b. Note that we assume $O(1)$ time memory allocator, which is essential for dynamic data structures (Fig. 5).

After the carry propagation, we check whether BL_b is in the range of $\frac{1}{2}\log n$ to $2\log n$ words, and split BL_b or merge it with its neighbor when it is out of the range. If the merged list exceeds $2\log n$ words, the list re-splits into two equal-size lists. All the list manipulations take $O(\sqrt{\log n})$ time, because BL_b has $O(\sqrt{\log n})$ sub-blocks. For the split of BL_b, we mark the split position sp of T_b.

We update T_{BL} to reflect the split or merge of T_b. If T_b remains in the range of $\frac{1}{2}\log n$ to $2\log n$ words, it is not necessary to update T_{BL}, because the number of blocks are unchanged. Otherwise, T_{BL} is updated by inserting or deleting $O(1)$ leaf nodes and by changing $n(v)$ for each node v on the path from updated leaves to the root. This update of a path takes $O(\log n)$ time.

Over-block updates. Now we consider $insert\text{-}over(c, b)$ or $delete\text{-}over(c, b)$, which is to update I and B in amortized $O(\log n)$ time. For the case that T_b is in the range, the over-block updates are simple. We change the length of T_b by $insert_I(0, select_I(1, b))$ or $delete_I(select_I(1, b)+1)$. To change the number of c's in T_b, we insert or delete 0 after the $(cm+b)$th 1 in B, e.g., $insert_B(0, select_B(1, cm + b))$ or $delete_B(select_B(1, cm + b) + 1)$.

The main problem is to handle the split or merge of T_b when T_b is out of the range. We split or merge the length bits of T_b in I by $O(\log n)$ time operations of the MN-structure. The split of T_b is processed by inserting 1 at the split position sp, and the merge of T_b with T_{b+1} is done by deleting the first 1 of length bits of T_{b+1}.

The update of B requires total $O(\sigma)$ queries on B, $O(1)$ queries for each c-group. Given split of T_b at position sp, we divide 0s representing the occurrences of each c in T_b into two parts, the occurrences in $T_b[1..sp]$ and the others, by inserting a new 1. The position of this insertion is computed by skipping 0s representing c in $T_b[1..sp]$ from the $(cm + b)$th 1 representing

$$\begin{array}{rcl} T & = & \underline{\texttt{babc}}\ \underline{\texttt{abbaaaba}}\ \underline{\texttt{abca}} \\ I & = & \underline{1000}\ \underline{1\mathbf{0000}000}\ \underline{1000} \\ B & = & 101\underline{\mathbf{000}00}100\ 1001\underline{\mathbf{0}00}10\ 10110 \end{array}$$

(split)

$$\rightarrow$$

$$\begin{array}{rcl} T & = & \underline{\texttt{babc}}\ \underline{\texttt{abba}}\ \underline{\texttt{aaba}}\ \underline{\texttt{abca}} \\ I & = & \underline{1000}\ \underline{1000}\ \underline{\mathbf{1}000}\ \underline{1000} \\ B & = & 101\underline{00}\underline{\mathbf{1}}\underline{000}100\ 1001\underline{00}\underline{\mathbf{1}}\underline{0}10\ 10110 \end{array}$$

Fig. 6. Example of over-block insertions.

$T = \texttt{abb bbc abc cab abb acc cab baa}$, $\Sigma = \{\texttt{aaa}, \texttt{aab}, \ldots \texttt{ccc}\}$

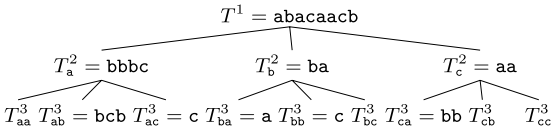

Fig. 7. Example of k-ary wavelet tree.

the bth block in the c-group. The update query for each c is $insert_B(1, select_B(1, cm + b) + rank_{T_b}(c, sp))$. The merging of c-groups is similar. Hence, the total cost is worst-case $O(\sigma \log n)$ time.

In the example of Fig. 6, suppose that four characters were inserted into T_2, which were three a's and one b. We changed the length bits of T_2 by inserting four 0s after the second 1 of I. The a-group and b-group of B were updated by inserting 0s into their second blocks. If T_2 splits at its fourth position, we process I and B to reflect this split. We update I by deleting the old 0 and inserting a new 1 at the split position, and we divide the second blocks of the a-group and b-group of B. Since the new T_2 has two a's and two b's, the second block of the a-group is divided by inserting a new 1 after two 0s, and that of the b-group is divided similarly. The c-group is not changed.

For $\sigma \leq \log n$, however, we amortize the $O(\sigma)$ queries over the in-block updates of BL_b. If $\sigma \leq \log n$, then the size of T_b becomes $\Theta(\frac{\log^2 n}{\log \sigma}) = \Omega(\sigma)$ characters, i.e., $\frac{1}{2}\Omega(\sigma)$ to $2\Omega(\sigma)$ characters. So there were $\Omega(\sigma)$ character updates in BL_b before the split or merge of BL_b occurs. This makes $O(\log n)$ amortized time per insertion or deletion of a character.

Lemma 5. *Given an n-length text over a σ-size alphabet with $\sigma \leq \log n$ and its partition of substrings with the lengths from $\frac{1}{2}w_\sigma \log n$ to $2w_\sigma \log n$, the insert-over and delete-over are processed in $O(\log n)$ amortized time and $o(n)$ bits.*

Note that we fix $\log n$ in the above descriptions. That is, w_σ is fixed as $\Theta(\frac{\log n}{\log \sigma})$. If n becomes σn or n/σ, then we need to change the rank table J_R, word-size w_σ, and all our structures. In the MN-structure, partial structures are built for the values $\log n - 1$, $\log n$, and $\log n + 1$ to avoid amortized $O(1)$ update [21]. In this paper we simply re-build all structures in $O(n)$ time whenever the value of $\log n / \log \sigma$ changes. This is amortized over all update operations and makes $O(1)$ costs per update operation.

Finally, we process $insert_T(c, i)$ by $insert\text{-}over(c, b)$ and $insert_{T_b}(c, r)$, where block number $b = rank_I(1, i)$ and offset $r = i - select_I(1, b) + 1$. In the same way, the process of $delete_T(i)$ is done by $delete\text{-}over(c, b)$ and $delete_{T_b}(r)$, where $c = access_T(i)$. We obtain the following dynamic structure for a small alphabet.

Lemma 6. *Given an n-length text over a σ-size alphabet with $\sigma \leq \log n$, there is a dynamic structure that provides $O(\log n)$ worst-case time access, rank, and select while supporting $O(\log n)$ amortized-time insert and delete in $n \log \sigma + O(\frac{n \log \sigma}{\sqrt{\log n}})$ bits.*

3.2. Dynamic rank/select structures for a large alphabet

Now we show a dynamic rank/select structure for a large alphabet by extending the above rank/select structure for a $\log n$-size alphabet. Our extension uses a k-ary wavelet tree [14,13]. Given a text T over a σ-size alphabet Σ, we regard $T[i]$ of $\log \sigma$ bits as $\lceil \frac{\log \sigma}{\log \log n} \rceil$ digits from a $\log n$-size alphabet Σ', i.e., $c = c_1 c_2 \ldots c_l$ where $c_i \in \Sigma'$ and $l = \lceil \frac{\log \sigma}{\log \log n} \rceil$. Let T^j be the concatenation of the jth digit of $T[i]$ for all i. T^j_s denotes a subsequence of T^j such that the jth digit of $T[i]$ belongs to T^j_s if and only if $T[i]$ has the prefix s of $(j-1)$ digits. For example, let $T = \texttt{abb bbc abc cab abb acc cab baa}$ be a text over an alphabet $\Sigma = \{\texttt{aaa}, \texttt{aab}, \ldots, \texttt{ccc}\}$. Using $\Sigma' = \{\texttt{a}, \texttt{b}, \texttt{c}\}$, $T^3 = \texttt{bccbbcba}$ and $T^3_{\texttt{ab}} = \texttt{bcb}$ (Fig. 7).

The k-ary wavelet tree represents T^j grouped by the prefix digits. The root of the tree contains T^1 and each of its children contains T^2_c for $0 \leq c < \log n$. If a node of the jth level contains T^j_s, then its children contain T^j_{sc} for $0 \leq c < \log n$. At the level $j \leq \lceil \frac{\log \sigma}{\log \log n} \rceil$, each node T^j_s represents a group of $T[i]$ by the order of prefix s. Instead of making the actual tree, we maintain it implicitly.

For the jth level, we concatenate all T^j_s by the lexicographic order of s and build our rank/select structure for the $\log n$-size alphabet on this concatenation. We also encode the length of each T^j_s by unary codes and concatenate the code bits

by the same order of s. This bit vector, F^j, takes $O(n+\sigma)$ bits and *rank*/*select* on F^j gives each position of T^j_s in the jth level concatenation. The sum of the sizes of F^j is $O(n\lceil\frac{\log\sigma}{\log\log n}\rceil)$, which is $o(n\log\sigma)$ for $\sigma > \log n$. Because T^j has a $\log n$-size alphabet and its character size is $\log\log n$ bits, the total size of the rank/select structures is $\lceil\frac{\log\sigma}{\log\log n}\rceil\cdot(n\log\log n + O(\frac{n\log\log n}{\sqrt{\log n}})) = n\log\sigma + o(n\log\sigma)$.

Then, $rank_T(c,i) = k_l$ and $select_T(c,k) = p_1$ are given by the following steps [14,13]:

$$\begin{aligned} k_1 &= rank_{T^1}(c_1, i) & p_l &= select_{T^l_{c_1c_2\ldots c_{l-1}}}(c_l, k)\\ k_2 &= rank_{T^2_{c_1}}(c_2, k_1) & p_{l-1} &= select_{T^{l-1}_{c_1c_2\ldots c_{l-2}}}(c_{l-1}, p_l)\\ &\ldots & &\ldots\\ k_l &= rank_{T^l_{c_1c_2\ldots c_{l-1}}}(c_l, k_{l-1}) & p_1 &= select_{T^1}(c_1, p_2). \end{aligned}$$

To process $access_T$, we should find the path of the character $T[i]$ from the root. This path starts from $c_1 = access_{T^1}(i)$ and we can find the next node by $rank_{T^1}(c_1, i)$:

$$\begin{aligned} c_1 &= access_{T^1}(i) & k_1 &= rank_{T^1}(c_1, i)\\ c_2 &= access_{T^2_{c_1}}(k_1) & k_2 &= rank_{T^2_{c_1}}(c_2, k_2)\\ &\ldots & &\ldots\\ c_l &= access_{T^l_{c_1c_2\ldots c_{l-1}}}(k_{l-1}). \end{aligned}$$

We process $insert_T$ and $delete_T$ in the same ways as $access_T$ and update the character of each level. Finally, we obtain a dynamic rank/select for a large alphabet with $\log n < \sigma \le n$.

Lemma 7. *Given an n-length text over a σ-size alphabet with $\log n < \sigma \le n$, there is a dynamic structure that gives $O(\lceil\frac{\log\sigma}{\log\log n}\rceil\log n)$ worst-case time access, rank, and select while supporting $O(\lceil\frac{\log\sigma}{\log\log n}\rceil\log n)$ amortized-time insert and delete in $n\log\sigma + o(n\log\sigma)$ bits.*

4. Indices for text collection

In this section we describe the relation between the rank/select structure and the compressed full-text index to apply our structure to the index for a collection of texts by Chan et al. [25]. Hon et al. [15] presented the duality between Burrows–Wheeler Transform (BWT) and the Ψ-function, which is a linear time transform between BWT and Ψ. Here we give a simple way of representing Ψ as a *select* query on BWT. This representation provides a simple version of the index for a text collection, which will be described in Section 4.2.

4.1. Suffix array, BWT and Ψ-function

The suffix array, denoted by SA, is an array of the starting positions of suffixes sorted by their lexicographic order. That is, $SA[i]$ is the starting position of the lexicographically ith suffix of T and $SA^{-1}[i]$ is the lexicographic order of the suffix with starting position i, $T[i..n]$. The compressed full-text indices such as CSA and FM-index replace SA of $O(n\log n)$ bits size by other compressed forms which have the equivalent information of suffix arrays.

There are two substitutions for SA: one is the Burrows–Wheeler Transform (BWT) of a text T and the other is the Ψ-function. The BWT of T, T^{bwt}, is a permutation of T made from the preceding characters of sorted suffixes. Given T and SA, T^{bwt} is defined as

$$T^{bwt}[i] = \begin{cases} T[SA[i]-1] & \text{if } SA[i] \neq 1\\ T[n] = \$ & \text{otherwise.} \end{cases}$$

Note that T is assumed to be terminated by a unique endmarker, $T[n] = \$$, which has the smallest lexicographic order in Σ. Because T^{bwt} can be compressed into $O(nH_k)$ bits [6], it is employed by compressed full-text indices explicitly or implicitly. The *rank* queries on T^{bwt} are the key steps of backward search of the FM-index [3].

The Ψ-function gives the lexicographic order of the next suffix. Given T and SA, the Ψ-function is defined as

$$\Psi[i] = \begin{cases} SA^{-1}[SA[i]+1] & \text{if } SA[i] \neq n\\ SA^{-1}[1] & \text{otherwise.} \end{cases}$$

The Ψ-function enables the compression of SA in CSA. In fact, from the duality between T^{bwt} and Ψ [15] we can directly obtain Ψ by the following *select*. Recall that $F[c]$ is the number of occurrences of characters less than c in the text T. Let $C[i]$ denote the first character of the ith suffix, $T[SA[i]..n]$. Then, $\Psi[i]$ is given by

$$\Psi[i] = select_{T^{bwt}}(C[i], i - F[C[i]]). \tag{1}$$

For the example of $\Psi[4] = SA^{-1}[6] = 11$ in Fig. 8, $\Psi[4]$ can be given by $select_{T^{bwt}}(C[4], 4 - F[C[4]]) = select_{T^{bwt}}(\mathtt{i}, 3)$.

i	SA	Ψ	T^{bwt}	Suffix
1	12	6	i	$
2	11	1	p	i$
3	8	8	s	ippi$
4	5	11	s	issippi$
5	2	12	m	ississippi$
6	1	5	$	mississippi$
7	10	2	p	pi$
8	9	7	i	ppi$
9	7	3	s	sippi$
10	4	4	s	sissippi$
11	6	9	i	ssippi$
12	3	10	i	ssissippi$

Fig. 8. Example of T^{bwt} and Ψ for $T =$ mississippi$.

4.2. Application to text-collection indexing

Given a collection of texts $C = \{T_1, T_2, \ldots, T_k\}$, Chan et al. proposed a dynamic BWT of the concatenation of texts, $T_C = \$T_1\$T_2 \ldots \$T_k$, to provide a pattern search with collection updates [25]. The BWT of T_C means an implicit suffix array of T_C. Their index has two components, *COUNT* and *PSI*. In fact, *COUNT* is a dynamic rank structure for the backward search algorithm, and *PSI* is a dynamic structure for the Ψ-function of the suffixes of T_C. To insert T_i into T_C^{bwt}, we compute the lexicographic order of each suffix of T_i by the backward search using *COUNT*. Then, the lexicographic order of each suffix $T_i[x..n_i]$ gives the position of $T_i[x-1]$ in T_C^{bwt}. The final $T_i[n_i]$ is inserted to T_C^{bwt} and its position denotes the lexicographic order of $T_i[1..n_i]$. To delete T_i from T_C, we assume that the lexicographic order of $T_i[1..n_i]$ is known and we follow Ψ to delete $T_i[n_i], T_i[1], \ldots, T_i[n_i - 1]$.

These two components, *COUNT* and *PSI*, can be replaced by *rank* and *select* on T_C^{bwt}, respectively. Chan et al. build *COUNT* and *PSI* as independent structures, so the updates of T_C^{bwt} mean the independent updates of *COUNT* and *PSI*. For simplicity, they present a removing of one component and a converting between *rank* and Ψ by $O(\log n)$ time binary search. However, this conversion can be discarded by our Ψ representation of Eq. (1). Hence, our rank/select structure on T_C^{bwt} provides a simple version of the index for a text collection. By substituting our rank/select for *COUNT* and *PSI*, we immediately obtain an index of $n \log \sigma + o(n \log \sigma)$ bits and $O((1 + \frac{\log \sigma}{\log \log n}) \log n)$ time per character. For texts over a small alphabet with $\sigma \leq \log n$, the operations takes $O(\log n)$ time per character.

Theorem 3. *Given a collection of texts $C = \{T_1, T_2, \ldots, T_k\}$, a dynamic rank/select structure with $O(X)$ time rank, select, access, insert, and delete operations on the BWT of $T_C = \$T_1\$T_2 \ldots \$T_k$ supports a searching for pattern P in $O(|P| \cdot X)$ time and an insertion/deletion of T_i in $O(|T_i| \cdot X)$ time.*

Recently, Mäkinen and Navarro [26] extended Chan et al.'s index for a collection of texts by locating pattern occurrences with small space and by removing the assumption of lexicographic order of T_i. We here presented a simple representation of *COUNT* and *PSI*, which can also be applied to Mäkinen and Navarro's extension [26].

Because any dynamic rank/select structure replaces *COUNT* and *PSI* by Theorem 3, González and Navarro's compressed rank/select structure [23] provides a compressed index of $nH_k + o(n \log \sigma)$ bits and $O((1 + \frac{\log \sigma}{\log \log n}) \log n)$ time per character for a collection of texts.

5. Space reduction by run-length encoding

In this section we reduce the space of our structure by run-length encoding (RLE). This method is an extension of Mäkinen and Navarro's Run-Length based FM-index (RLFM) [18], in which they addresses only *rank* query. For applications such as the Ψ-function and the dynamic structures, we consider other queries: *select*, *insert* and *delete*.

Given RLE of $T = (c_1, l_1)(c_2, l_2) \ldots (c_{n'}, l_{n'})$, where each pair (c_1, l_1) denotes an l_i-length run of a same character c_i, we represent $RLE(T)$ by the sequence of run characters, $T' = c_1 c_2 \ldots c_{n'}$, and the bit vector of run lengths, $L = 10^{l_1-1}10^{l_2-1} \ldots 10^{l_{n'}-1}$. See Fig. 9. In addition to L, we use some additional bit vectors: a grouped length vector L' and a frequency table F'. The length vector L' represents the lengths of runs in the order of their characters and positions, so that the lengths are grouped by the characters. L' is similar to the over-block encoding B. However, the lower bounds of L and L' are $O(n)$ because the number of runs is $O(n)$.

The frequency table F' provides $F'[c]$ that is the number of occurrences of characters less than c in text T'. F' is represented by $O(n' + \sigma)$ bits vector. Using L' and F', we compute $F[c]$ by $select_{L'}(1, F'[c] + 1) - 1$. Then, there is a mapping between the kth c of T and the $(F[c] + k)$th bit of L'.

For the number of runs in T^{bwt}, Mäkinen and Navarro showed that the number of runs is less than $nH_k + \sigma^k$, so the size of T' for $RLE(T^{bwt})$ is $nH_k \log \sigma + O(\sigma^k \log \sigma)$ bits.

$$\begin{array}{rcl@{\qquad}c@{\qquad}rcl} T & = & (\mathtt{b},2)(\mathtt{a},2)(\mathtt{b},4)(\mathtt{c},2)(\mathtt{a},3) & & & & (\mathtt{a},2)(\mathtt{a},3)(\mathtt{b},2)(\mathtt{b},4)(\mathtt{c},2) \\ L & = & \underline{10}\,\underline{10}\,\underline{1000}\,\underline{10}\,\underline{100} & \rightarrow & L' & = & \underline{10}\,\underline{100}\,\underline{10}\,\underline{1000}\,\underline{10} \\ T' & = & \mathtt{babca} & & F' & = & 00100101 \end{array}$$

Fig. 9. Additional vectors for rank/select structure on $RLE(T)$.

Lemma 8 ([18]). *The number of runs in T^{bwt} is at most $n \cdot \min(H_k(T), 1) + \sigma^k$ for any $k \geq 0$. In particular, this is $nH_k(T) + o(n)$ for any $k \leq \alpha \log_\sigma n$ and any constant $0 < \alpha < 1$.*

The rank/select structure on $RLE(T)$ uses the structures on (plain) text T' and on bit vectors including L. Therefore, the total size of a structure on $RLE(T)$ is bounded by the size of a structure on T' plus $O(n)$ bits. We start from RLFM's *rank* on RLE and extend RLFM to support *select* and dynamic updates.

Lemma 9 ([18]). *Given $RLE(T)$, if there are $O(t_S)$ time $select_{T'}$ and $O(t_R)$ time $rank_{T'}$ using $s(T')$ bits, then $rank_T(c, i)$ is processed in $O(t_S + t_R) + O(1)$ time and $s(T') + O(n)$ bits.*

Lemma 10. *Given $RLE(T)$, if there is $O(t_S)$ time $select_{T'}$ using $s(T')$ bits, then $select_T(c, k)$ is processed in $O(t_S) + O(1)$ time and $s(T') + O(n)$ bits.*

Proof. For query $select_T(c, k)$, our first step is to find the starting position of the run that contains the kth c. We use the mapping between the kth c of T and the $(F[c] + k)$th bit of L'. Let k' be the number of c runs containing the kth c. The value k' is obtained from the number of 1s that represent c runs in L':

$$k' = rank_{L'}(1, F[c] + k) - F'[c].$$

Then, we get the starting position of the k'th c run from T' and L, and add its difference to the position of the kth c. The difference between the position of the kth c and the starting position is computed on L' by using the fact that the k'th c run is mapped from $(F'[c] + k')$th 1 in L':

$$\begin{aligned} pos &= select_L(1, select_{T'}(c, k')) \\ select_T(c, k) &= pos + (F[c] + k - select_{L'}(1, F'[c] + k')). \end{aligned}$$

In addition to $select_{T'}$, we use only $O(1)$ operations on binary dictionaries. □

From Lemmas 9 and 10, we obtain a static rank/select structure on $RLE(T)$ by employing the static rank/select structure on T' such as [16,15,17] and the bit rank/select structure on L' such as [8,10].

Theorem 4. *Given $RLE(T)$ of a text T with n' runs, there is a rank/select structure that supports $O(\log\log\sigma)$ time rank and $O(1)$ time select, using $n'H_0(T') + O(n)$ bits. This structure can be constructed in deterministic $O(n)$ time.*

For dynamic versions, we use our structure in Theorem 2 for *rank*/*select* on T' and the MN-structure for *rank*/*select* on bit vectors. The retrieving operations, $rank_T$ and $select_T$, follow the same steps as in Lemmas 9 and 10, and the updating operations, $insert_T(c, i)$ and $delete_T(i)$, are described as follows.

Lemma 11. *Given $RLE(T)$, if there are $O(t_A)$ time $access_{T'}$, $O(t_R)$ time $rank_{T'}$, and $O(t_I)$ time $insert_{T'}$ using $s(T')$ bits, then $insert_T$ is processed in $O(t_A + t_R + t_I) + O(\log n)$ time and $s(T') + O(n)$ bits.*

Proof. The i'th run that contains $T[i]$ is obtained by $i' = rank_L(1, i)$. We describe only the case that $T[i+1]$ is also a character of the i'th run. The case that $T[i+1]$ is the first character of the next run is similar but simpler.

For the insertion of c into the i'th run there are two cases whether c is equal to $T'[i']$ or not. If $c = T'[i']$, we just lengthen the length of the i'th run by updating L and L'. Otherwise, we split the i'th run and insert a new run by $insert_{T'}$ (Fig. 10).

(1) $c = T'[i']$: We simply update L by $insert_L(i, 0)$. To update L', we find the length of the i'th run in L', which is represented as the $(F'[c] + rank_{T'}(c, i'))$th 1 and following 0s. Then, we update the length by $insert_{L'}(select_{L'}(1, F'[c] + rank_{T'}(c, i')), 0)$.

(2) $c \neq T'[i']$: We split the i'th run and create the lengths for a new run in L and L'. We update L by inserting 11 and deleting 0 at position i. To update L', we find the split position of the i'th run and the new position of c. The split position of the i'th run is computed by $F'[T[i']] + rank_{T'}(T'[i'], i')$ plus the offset, $i - select_L(1, i')$. The new position of c is found by $F'[c] + rank_{T'}(c, i')$ as in case 1. Let $change(x, c)$ denote consecutive calls of $insert(x, c)$ and $delete(x)$. The split and create operations are as follows:

- $change_L(i, 1)$
- $insert_L(i, 1)$
- $change_{L'}(select_{L'}(1, F'[T'[i']] + rank_{T'}(T'[i'], i')) + i - select_L(1, i'), 1)$
- $insert_{L'}(select_{L'}(1, F'[c] + rank_{T'}(c, i') + 1), 1)$.

After updating L and L', we call $insert_{T'}(T[i'], i')$ and $insert_{T'}(c, i')$. Updating F' is done by $insert_{F'}(0, select_{F'}(1, c))$ and $insert_{F'}(0, select_{F'}(1, T[i']))$.

In the above cases, we use a constant number of operations on bit vectors and on T'. □

We process $delete_T$ in a way similar to $insert_T$.

$$\begin{array}{rcl} T &=& \texttt{bbaabbbbccaaa} \\ L &=& 1010100010100 \\ L' &=& 1010010100010 \\ T' &=& \texttt{babca} \\ F' &=& 00100101 \end{array} \quad \rightarrow \quad \begin{array}{rcl} T &=& \texttt{bbaab}\mathbf{\texttt{c}}\texttt{bbccaaa} \\ L &=& 10101011010100 \\ L' &=& 10100101010110 \\ T' &=& \texttt{bab}\mathbf{\texttt{c}}\mathbf{\texttt{b}}\texttt{ca} \\ F' &=& 00\mathbf{1}000\mathbf{1}001 \end{array}$$

Fig. 10. Example of insertion into T, $insert_T(\mathtt{c}, 6)$.

Lemma 12. *Given RLE*(T)*, if there are* $O(t_A)$ *time* $access_{T'}$, $O(t_R)$ *time* $rank_{T'}$, *and* $O(t_D)$ *time* $delete_{T'}$ *using* $s(T')$ *bits, then* $delete_T$ *is processed in* $O(t_A + t_R + t_D) + O(\log n)$ *time and* $s(T') + O(n)$ *bits.*

Theorem 5. *Given RLE*(T) *of a text* T *with* n' *runs, there is a dynamic rank/select structure that supports* $O((1 + \frac{\log\sigma}{\log\log n})\log n)$ *worst-case time rank and select, while supporting* $O((1+\frac{\log\sigma}{\log\log n})\log n)$ *amortized time insert and delete in* $n'\log\sigma + o(n'\log\sigma) + O(n)$ *bits.*

6. Conclusion

In this paper we proposed dynamic rank/select structures of $n\log\sigma + o(n\log\sigma)$ bits that support $O((1 + \frac{\log\sigma}{\log\log n})\log n)$ worst-case time *rank*/*select* and $O((1 + \frac{\log\sigma}{\log\log n})\log n)$ amortized time *insert*/*delete*. We started with a small-alphabet structure providing $O(\log n)$ worst-case time *rank*/*select* and $O(\log n)$ amortized time *insert*/*delete* for a $\log n$-size alphabet and extended it for a large alphabet with $\log n < \sigma \leq n$. Our rank/select structure can be applied to an index of texts collection and we obtained a simple version of that index. Applying to RLE of texts, we presented static and dynamic versions of RLFM.

References

[1] R. Grossi, J.S. Vitter, Compressed suffix arrays and suffix trees with applications to text indexing and string matching, SIAM Journal on Computing 35 (2) (2005) 378–407.
[2] K. Sadakane, New text indexing functionalities of the compressed suffix arrays, Journal of Algorithms 48 (2) (2003) 294–313.
[3] P. Ferragina, G. Manzini, Opportunistic data structures with applications, in: Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science, 2000, pp. 390–398.
[4] K. Sadakane, R. Grossi, Squeezing succinct data structures into entropy bounds, in: Proceedings of the 17th Annual ACM–SIAM Symposium on Discrete Algorithms, 2006, pp. 1230–1239.
[5] P. Ferragina, R. Venturini, A simple storage scheme for string achieving entropy bounds, Theoretical Computer Science 372 (2007) 115–121.
[6] G. Manzini, An analysis of the Burrows–Wheeler transform, Journal of ACM 48 (3) (2001) 407–430.
[7] G. Jacobson, Space-efficient static trees and graphs, in: Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science, 1989, pp. 549–554.
[8] D.R. Clark, Compact pat trees, Ph.D. Thesis, Univ. Waterloo, 1998.
[9] J.I. Munro, Tables, in: Proceedings of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science, 1996, pp. 37–42.
[10] R. Raman, V. Raman, S.S. Rao, Succinct indexable dictionaries with applications to encoding k-ary trees and multisets, in: Proceedings of the 13th Annual ACM–SIAM Symposium on Discrete Algorithms, 2002, pp. 233–242.
[11] A. Golynski, R. Grossi, A. Gupta, R. Raman, S.S. Rao, On the size of succinct indexes, in: Proceedings of the 15th Annual European Symposium on Algorithms, 2007, pp. 371–382.
[12] M. Patrascu, Succincter, in: Proceedings of the 49th Annual IEEE Symposium on Foundations of Computer Science, 2008, pp. 305–313.
[13] P. Ferragina, G. Manzini, V. Mäkinen, G. Navarro, Compressed representations of sequences and full-text indexes, ACM Transactions on Algorithms 3 (2) (2007).
[14] R. Grossi, A. Gupta, J.S. Vitter, High-order entropy-compressed text indexes, in: Proceedings of the 14th Annual ACM–SIAM Symposium on Discrete Algorithms, 2003, pp. 841–850.
[15] W.-K. Hon, K. Sadakane, W.-K. Sung, Breaking a time-and-space barrier in constructing full–text indices, in: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science, 2003, pp. 251–260.
[16] A. Golynski, J.I. Munro, S.S. Rao, Rank/select operations on large alphabets: a tool for text indexing, in: Proceedings of the 17th Annual ACM–SIAM Symposium on Discrete Algorithms, 2006, pp. 368–373.
[17] J. Barbay, M. He, J.I. Munro, S.S. Rao, Succinct indexes for strings, binary relations and multi-labeled tree, in: Proceedings of the 18th Annual ACM–SIAM Symposium on Discrete Algorithms, 2007, pp. 680–689.
[18] V. Mäkinen, G. Navarro, Succinct suffix arrays based on run-length encoding, in: Proceedings of the 16th Annual Symposium on Combinatorial Pattern Matching, 2005, pp. 45–56.
[19] R. Raman, V. Raman, S.S. Rao, Succinct dynamic data structures, in: Proceedings of the 7th International Workshop on Algorithms and Data Structures, 2001, pp. 426–437.
[20] W.-K. Hon, K. Sadakane, W.-K. Sung, Succinct data structures for searchable partial sums, in: Proceedings of the 14th Annual Symposium on Algorithms and Computation, 2003, pp. 505–516.
[21] V. Mäkinen, G. Navarro, Dynamic entropy-compressed sequences and full-text indexes, in: Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching, 2006, pp. 306–317.
[22] A. Gupta, W.-K. Hon, R. Shah, J.S. Vitter, A framework for dynamizing succinct data structures, in: Proceedings of the 34th International Colloquium on Automata, Languages and Programming, 2007, pp. 521–532.
[23] R. González, G. Navarro, Improved dynamic rank-select entropy-bound structures, in: Proceedings of the 8th Latin American Symposium on Theoretical Informatics, 2008, pp. 374–386.
[24] S. Lee, K. Park, Dynamic compressed representation of texts with rank/select, in: Proceedings of the 11th Korea–Japan Joint Workshop on Algorithms and Computation, 2008, pp. 131–138.
[25] H.-L. Chan, W.-K. Hon, T.-W. Lam, Compressed index for a dynamic collection of texts, in: Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching, 2004, pp. 445–456.
[26] V. Mäkinen, G. Navarro, Dynamic entropy-compressed sequences and full-text indexes, ACM Transactions on Algorithms 4 (3) (2008).