Dynamic Compressed Representation of Texts with Rank/Select

Sunho Lee and Kunsoo Park
School of Computer Science and Engineering, Seoul National University, Korea
shlee@theory.snu.ac.kr; kpark@theory.snu.ac.kr

Received 26 September 2008; Accepted 6 February 2009

Given an n-length text T over a σ-size alphabet, we present a compressed representation of T which supports retrieving queries of rank/select/access and updating queries of insert/delete. For a measure of compression, we use the empirical entropy $H(T)$, which defines a lower bound $nH(T)$ bits for any algorithm to compress T of $n \log \sigma$ bits. Our representation takes this entropy bound of T, i.e., $nH(T) \leq n \log \sigma$ bits, and an additional bits less than the text size, i.e., $o(n \log \sigma) + O(n)$ bits. In compressed space of $nH(T) + o(n \log \sigma) + O(n)$ bits, our representation supports $O(\log n)$ time queries for a $\log n$-size alphabet and its extension provides $O((1+\frac{\log \sigma}{\log \log n}) \log n)$ time queries for a σ-size alphabet.

Categories and Subject Descriptors: Succinct Data Structures [**Algorithms and Complexity**]
General Terms: Compressed Full-Text Index, Rank/Select Structure
Additional Key Words and Phrases: compression, data structure, index, rank, select, text

1. INTRODUCTION

For text processing data structures, we consider two basic functions: *rank*(c, i) which counts the number of occurrences of character c up to position i and *select*(c, k) which finds the position of the k-th character c. These rank/select functions are simple but powerful so that these functions coupled with Burrows-Wheeler Transform (BWT) directly support compressed full-text indices such as FM-index [Ferragina and Manzini 2005] and Compressed Suffix Arrays [Grossi and Vitter 2005]. Given an n-length text over a σ-size alphabet, the compressed full-text indices provide pattern searching in only $O(n \log \sigma)$ bits of text itself. In the compressed full-text indices, the rank function is a key of the pattern searching algorithms and the select function is essential to reduce the sizes of indices.

Moreover, the space of rank/select functions defines the sizes of compressed full-text

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

	Time	Space	Reference
Uncompressed	$O(\log\sigma\log_b n)$ rank/select $O(\log\sigma b)$ insert/delete	$n\log\sigma + o(n\log\sigma)$	[Hon et al. 2003]
	$O((1/e)\log\log n)$ rank/select $O((1/e)n^e)$ insert/delete	$n\log\sigma + o(n\log\sigma)$	[Gupta et al. 2007]
	$O((1+\frac{\log\sigma}{\log\log n})\log n)$	$n\log\sigma + o(n\log\sigma)$	[Lee and Park 2007]
Compressed	$O(\log\sigma\log n)$	$nH(T) + o(n\log\sigma)$	[Mäkinen and Navarro 2006]
	$O((1+\frac{\log\sigma}{\log\log n})\log n)$	$nH(T) + o(n\log\sigma)$	[González and Navarro 2008]
			This paper

indices. A rank/select function with the space of text itself makes compressed full-text indices of $O(n\log\sigma)$ bits, and a compressed function with $O(nH(T))$ bits, where $H(T)$ is the empirical entropy that is a lower bound of any compression algorithm for T [Manzini 2001], produces more compressed indices of $O(nH(T)) \leq O(n\log\sigma)$ bits. In this paper we consider a dynamic and compressed rank/select function which reflects update of texts while keeps texts in a compressed form. This is a basic component of a dynamic compressed full-text indices such as [Chan et al. 2004].

The dynamic rank/select structure was first considered on binary strings as a special case of the dynamic partial sum problem by Raman et al. [Raman et al. 2001] and Hon et al. [Hon et al. 2003]. These binary rank/select structures can be extended for a σ-size alphabet by binary wavelet trees [Grossi et al. 2003] with $O(\log\sigma)$ slowdown factor. Mäkinen and Navarro 2006; 2008] first proposed a compressed structure of $nH(T) + o(n\log\sigma)$ bits with $O(\log\sigma\log n)$ worst-case time operations. Lee and Park [Lee and Park 2007] proposed an uncompressed but faster structure, which provides $O((1+\frac{\log\sigma}{\log\log n})\log n)$ worst-case time rank/select queries and $O((1+\frac{\log\sigma}{\log\log n})\log n)$ amortized time insert/delete queries in space of $n\log\sigma + o(n\log\sigma)$ bits. These improved operations were the results of extending $O(\log n)$ time operations for a $\log n$-size alphabet by k-ary wavelet trees [Ferragina et al. 2007] with $O(\log_k\sigma)$ slowdown factor. Recently, González and Navarro [González and Navarrow 2008] achieved both compressed space of $nH(T) + o(n\log\sigma)$ bits and $O((1+\frac{\log\sigma}{\log\log n})\log n)$ worst-case time for all operations. Note that there are rank/select structures not based on wavelet trees such as Gupta et al.'s [Gupta et al. 2007].

In this paper we propose a compressed structure of our previous result [Lee and Park 2007], and this is a simple alternative to González and Navarro's structure [Gonzalez and Navarro 2008]. González and Navarro use a block-identifier encoding of $nH(T) + o(n)$ bits [Ferragina et al. 2007; Raman et al. 2002] to compress texts and propose a theoretical counting structure to guarantee worst-case time updates. Instead of the complex block-identifier encoding, we employ a gap encoding of $nH(T) + O(n)$ bits [Grossi et al. 2004; Mäkinen and Navarro 2007; Sadakane 2003], which takes slightly more space but supports practical implementations of compressed full text indices [Grossi et al. 2004]. We also propose a simple counting structure with amortized updates. Then, we obtain a compressed structure providing the queries in the same time as [Lee and Park 2007]. In compressed space of $nH(T) + o(n\log\sigma) +$

$O(n)$ bits, our rank/select/access queries take worst-case $O(\log n)$ time and insert/delete queries use amortized $O(\log n)$ time for a $\log n$-size alphabet. For an $O(n^{\varepsilon})$-size alphabet, we obtain $O((1+\frac{\log \sigma}{\log \log n})\log n)$ time queries by using $\log n$-ary wavelet trees.

This paper is organized as follows. Section 2 introduces preliminaries such as the empirical entropy of texts and the gap encoding. Section 3 shows how to organize gap encoded texts for supporting rank/select functions and updates of texts. In Section 4, we describe a simple counting structure for rank/select functions. Section 5 presents an extension for a large alphabet and an application to the BWT of texts for a high-order entropy compression. Section 6 finally concludes.

2. DEFINITIONS AND PRELIMINARIES

Let $T = T[1]T[2]...T[n]$ be an n-length text and $\Sigma = \{0, 1, ..., \sigma - 1\}$ be a σ-size alphabet. We first assume a $\log n$-size alphabet, i.e., $\sigma \leq \log n$ and then show how to handle a general alphabet. We assume the RAM model with constant time arithmetic and bitwise operations on a word of $\Theta(\log n)$ bits. In the RAM model, we can access its memory by the pointers of $O(\log n)$ bits.

Empirical entropy. To measure the efficiency of compression, we define the empirical entropy $H(T)$ of T which is a lower bound of any compression algorithm for input text T [Manzini 2001]. The zeroth order empirical entropy $H_0(T)$ is a lower bound of compression algorithms considering numbers of occurrences of characters. Let n_c be the total number of occurrences of c in T. The zeroth order empirical entropy $H_0(T)$ is defined by

$$H_0(T) = -\frac{1}{n}\sum_{c\in\Sigma} n_c \log\frac{n_c}{n}$$

From the concavity of the entropy, $H_0(T) \leq \log \sigma$. The high-order empirical entropy is a lower bound of compression algorithms using a context that is k characters preceding each character of T. Let Σ^k be the set of all k-length words and w_T denote the concatenation of the characters following w in T. The k-th order empirical entropy $H_k(T)$ is defined by

$$H_k(T) = \frac{1}{n}\sum_{w\in\Sigma^k} |w_T|\, H_0(w_T)$$

Note that $H_{k+1}(T) \leq H_k(T) \leq \log \sigma$ for any text T. We simply use H_0 or H_k instead of $H_0(T)$ or $H_k(T)$ if the text T is not ambiguous. To represent T in nH_0 bits with updates of T, we can't use global information n_c/n, so we employ an encoding scheme depending on local information.

Gap encoding. The gap encoding achieves the entropy bound of $nH_0(T)$ bits by using only local information [Grossi et al. 2004; Mäkinen and Navarro 2007; Sadakane 2003]. Let $p_1^c, p_2^c, ..., p_t^c$ be the positions of occurrences of c in T. The gap encoding of T, $G(T)$, encodes the distances of adjacent occurrences, $p_i^c - p_{i-1}^c$. Then, the size of $G(T)$ is $|G(T)| = \sum_{c\in\Sigma}\sum_{i=1}^{t} |C(p_i^c - p_{i-1}^c)|$

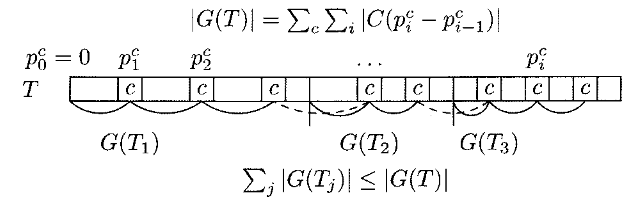

$|G(T)| = \sum_c \sum_i |C(p_i^c - p_{i-1}^c)|$
$p_0^c = 0$
p_1^c
p_2^c
...
p_i^c
T
c
$G(T_1)$
$G(T_2)$
$G(T_3)$
$\sum_j |G(T_j)| \le |G(T)|$

Figure 1. Gap encoding of a partition of the text.

where $p_0^c = 0$ and $C(p_i^c - p_{i-1}^c)$ is the code of $p_i^c - p_{i-1}^c$ (called the C-code) by a self-delimiting code. The self-delimiting code is a prefix coding of integer values which represents a positive integer x in $a\lceil \log x \rceil + b$ bits, where a and b are some constants.

The occurrences distance $p_i^c - p_{i-1}^c$ is encoded in $a\lceil \log(p_i^c - p_{i-1}^c) \rceil + b$ bits. Then, the total size of the gap encoding takes $a(H_0 + 1)n + bn$ bits, i.e., $|G(T)| \le a(H_0 + 1)n + bn$.

On the RAM model, it takes constant time to encode x or decode $C(x)$ by using $o(n)$ bits table. Examples of self-delimiting codes are γ-code and δ-code by Elias [Elias 1975]. The γ-code represents x as $1^{|b(x)|-1}0b(x)$, where $b(x)$ is a simple binary representation of x. The γ-code size of x is $2\lceil \log x \rceil + 1$ bits. If we use δ-code, the code size is reduced to $\lceil \log x \rceil + o(\log x)$ bits, where the δ-code representation of x is $1^{b(|b(x)|)-1}0b(|b(x)|)0b(x)$.

Lemma 2.1. *Given a text T, the total size of gap encodings of T with a self-delimiting code is $|G(T)| \le nH_0 + o(nH_0) + O(n)$ bits and it takes $O(|T|)$ time to encode T and to decode $G(T)$ by using additional tables of $o(n)$ bits.*

Problem definition. Our problem is to represent T in a compressed form which supports updates of T by insertions or deletions of a character. The size of this compressed form should achieve the empirical entropic space, $nH_0(T)$ bits. We also provide the following rank/select queries on T in the compressed form.

- *$rank_T(c, i)$*: gives the number of character c in $T[1..i]$.
- *$select_T(c, k)$*: gives the position of the k-th c in T.

3. DYNAMIC COMPRESSED REPRESENTATION OF TEXTS

In this section, we describe a gap-encoded representation of T over a $\log n$-size alphabet which provides the $O(\log n)$ time queries of access, insert, and delete in space of $nH_0 + o(nH_0) + O(n)$ bits. Our key observation is that the gap encoding can be applied to a partition of T. Therefore, we partition T into m substrings $T_1, T_2, \ldots, T_m$ and encode each T_j by the gap encoding with a self-delimiting code. Our representation also considers rank/select queries on a substring T_j and the complete rank/select queries on T will be given in Section 4.

We first show that the gap encoding of a partition of T has a smaller size than the whole encoding of T, i.e., $\sum_{j=1}^{m} |G(T_j)| \le |G(T)|$. The occurrence distances of $G(T_1)$, $G(T_2)$, ..., $G(T_m)$ have smaller values than those of $G(T)$, because we encode the first c-occurrence of each T_j by the distance from the starting position of T_j, not the distance from the previous c-occurrence in T_{j-1}. See Figure 1. Since the size of $|C(x)| = a\lceil \log x \rceil + b$ has the property that $|C(x)| \leq |C(y)|$ for any integer $1 \leq x \leq y$, the size of the encoded partition becomes $\Sigma_{j=1}^{m} |G(T_j)| \leq |G(T)|$.

Lemma 3.1. *Given a partition of text $T = T_1T_2 \ldots T_m$, the total size of gap encodings of T_j with a self-delimiting code becomes $\Sigma_{j=1}^{m} |G(T_j)| \leq |G(T)| \leq nH_0 + o(nH_0) + O(n)$ bits.*

Now we present how to organize encoded T_j for rank/select and update queries. The partitioning of T is made so that the size of a code block, $|G(T_j)|$, to be $\log^2 n$ to $4\log^2 n$ bits. We encode T from beginning to end and make a new code block $G(T_j)$ whenever the encoding of the current substring of T exceeds $2\log^2 n$ bits. By Lemma 3.1, the total size of the code blocks is $\Sigma_{j=1}^{m} |G(T_j)| \leq |G(T)| \leq nH_0 + o(nH_0) + O(n)$ bits.

The length of T_j satisfies $\frac{\log^2 n}{2\log\sigma + O(1)} \leq |T_j| \leq 4\log^2 n$ as follows. The number of C-codes in $G(T_j)$, which is the same as $|T_j|$, cannot exceed $|G(T_j)|$. Since $|G(T_j)| \leq 4\log^2 n$, we get the upper bound. Since $\log^2 n \leq |G(T_j)| \leq (1 + o(1))|T_j|H_0(T_j) + O(|T_j|)$, we have $|T_j| \geq \frac{\log^2 n}{2H_0(T_j) + O(1)}$. Because $H_0(T_j) \leq \log\sigma$, we get the lower bound $|T_j| \geq \frac{\log^2 n}{2\log\sigma + O(1)}$. Then, the total number, m, of code blocks is $O(\frac{n\log\sigma}{\log^2 n}) = O(\frac{n\log\log n}{\log^2 n})$ for $\sigma \leq \log n$.

For rank/select on T_j with a $\log n$-size alphabet, our idea is to store the C-codes of $G(T_j)$ in order of characters so that the C-codes of a same character c can be scanned in sequential. As in the example of Figure 2, $G(T_j)$ is the concatenation of $G_c(T_j)$ for all characters c. To access the C-codes of c, we reserve $O(\sigma\log\log n)$ bits for each $G(T_j)$ to mark the position of the starting C-code for each c. Its total overhead becomes $m \cdot O(\sigma\log\log n) = O(\frac{n\log\log^2 n}{\log n})$ bits for $\sigma \leq \log n$.

Our representation is illustrated in Figure 3. We build a red-black tree where a leaf node is $G(T_j)$ and an internal node maintains the number of code blocks in its subtree. By traversing this tree, we can find $G(T_j)$ for given block number j. We use a dynamic bit vector I that represents the length of T_j. I has total n bits, where the j-th 1 denotes the starting position of T_j and the following $|T_j|-1$ 0s indicate the length of T_j. By using I, we divide a rank/select query to two level queries: an over-block query which counts the number of c in the code blocks before T_j and an in-block query which

$$\begin{aligned} T_j &= \text{abbc ccba bbca bccc caca bbca} \\ G_{\text{a}}(T_j) &= C(1)C(7)C(4)C(6)C(2)C(4) \\ G_{\text{b}}(T_j) &= C(2)C(1)C(4)C(2)C(1)C(3)C(8)C(1) \\ G_{\text{c}}(T_j) &= C(4)C(1)C(1)C(5)C(3)C(1)C(1)C(1)C(2)C(4) \end{aligned}$$

Figure 2. Example of a code block $G(T_j)$.

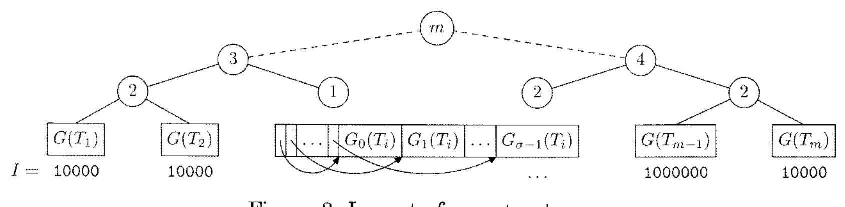


Figure 3. Layout of our structures.

counts the number of c in T_j.

Now we describe the processing of rank/select queries on T_j by using $o(n)$ bits table. For complete rank/select on T, we describe auxiliary structures handling rank/select over T_j in the next section. We define the following tables for all bit patterns of a $\log n/2$ bits code, $x = C(d_1)C(d_2) \ldots C(d_l)$ and position t of x [Makinen and Navarro 2007].

- $G[x][t]$: the maximum number, k, of C-codes in $x[1..t]$.
- $Pos[x][t]$: the total length of the k C-codes in $x[1..t]$, i.e., $\Sigma_{i=1}^{k} \mid C(d_i) \mid$.
- $DPos[x][t]$: the decoded values sum of the k C-codes in $x[1..t]$, i.e., $\Sigma_{i=1}^{k} d_i$.
- $H[x][p]$: the maximum k' such that $\Sigma_{i=1}^{k'} d_i \leq p$, where $p \leq \log^2 n$.

Using these tables, we process rank/select on T_j by scanning $G(T_j)$ of $c \log^2 n$ bits in $O(\log n)$ time. The detailed steps use the binary rank/select by Mäkinen and Navarro [Mäkinen and Navarro 2007].

Rank/select queries. To answer $rank_{T_j}(c, i)$, we first find the starting C-code for c. Then, we sums up the decoded distances of the C-codes in a code of $\log n/2$ bits by using table *DPos*. To find the boundaries of C-codes, we compute the number of C-codes and the length of C-codes by using tables G and *Pos*, respectively. These tables enable us to scan $G(T_j)$ by $\log n/2$ bits at once. If the sum of decoded distances is greater than i, we get the final code of $\log n/2$ bits that contains the last C-code of $T_j[p] = c$ with $p < i$. We can decode the final code one-by-one in $O(\log n)$ time. For $select_{T_j}(c, k)$, we check whether the numbers of C-codes exceeds k to get the final code containing the k-th c of T_j.

Let us consider $G_b(T_j) = C(2)C(1)C(4) \mid C(2)C(1)C(3) \mid C(8)C(1)$ in the example of Figure 2. We assume that these C-codes of $G_b(T_j)$ are grouped by $\log n/2$ bits. Note that the boundary of a code of $\log n/2$ bits may not be consistent with a C-code boundary, but we can find the correct position of a C-code by using G and *Pos*. For $rank_{T_j}(\mathrm{b}, 20)$, we skip the first code of $\log n/2$ bits by *DPos*, because the decoded distance sum is 7. The next code is also skipped, where the distance sum is 13. The distance sum up to the third code is 22, so we decode this final code one-by-one in $O(\log n)$ time.

Access query. Since we group the code block $G(T_j)$ by characters, the access of $T_j[p]$ requires the whole scan of $G(T_j)$. We scan each character group in the same way as the rank query. If we decode the final code of $\log n/2$ bits one-by-one, the whole scan would take $O(\sigma \log n)$ time. Instead, we use table H that returns the maximum number, k, of C-codes such that $\Sigma_{i=1}^{k} d_i \leq p'$, where p' is p minus the distances sum before the final code. If the distances sum is equal to p', then $T_j[p]$ is the current character. Otherwise, we check the next character. The total scan time is $O(\sigma + \log n) = O(\log n)$ for $\sigma \leq \log n$.

Insert/delete queries. The insertion or deletion at position p are also similar to the access query. In addition to scanning each character group, we update the final code for each c which contains the first C-code of $T_j[i] = c$ with $i \geq p$. For an insertion of c, we split the first C-code of $T_j[i] = c$ with $i = p$ into two C-codes of $T_j[p] = c$ and

$$\begin{aligned} T_j &= \texttt{abbc ccba bbca bccc cabca bbca} \\ G_{\texttt{a}}(T_j) &= C(1)C(7) \mid C(4)C(6)\mathbf{C(3)} \mid C(4) \\ G_{\texttt{b}}(T_j) &= C(2)C(1)C(4) \mid C(2)C(1)C(3) \mid \mathbf{C(6)C(3)}C(1) \\ G_{\texttt{c}}(T_j) &= C(4)C(1)C(1) \mid C(5)C(3) \mid C(1)C(1)C(1)\mathbf{C(3)}C(4) \end{aligned}$$

Figure 4. Example of an insertion to $G(T_j)$.

$T_j[i+1] = c$. We increase other final C-codes by one. The deletion is done by deleting the C-code of $T_j[i]$ with $i = p$ and decreasing other C-codes of $T_j[i]$ with $i > p$. The changes of C-codes might induce the changes of lengths of C-codes, hence we need to check the old positions of the starting C-codes to scan and update at once.

For example, let us consider an insertion of b at position 19 (Figures 2 and 4). The distance sums of $G_a(T_j)$ are 8, 20, and 24. Hence, we find the second code that has the distance sum exceeding 19 and the first C-code, $C(2)$, with distance sum $20 > 19$. We increase $C(2)$ to $C(3)$. The distance sums of $G_b(T_j)$ are 7, 13, 22. We find $C(8)$, the first C-code with distance sum $21 > 19$ and split $C(8)$ into $C(6)$ of $T_j[19] = \texttt{b}$ and $C(3)$ of $T_j[22] = \texttt{b}$. The update of $G_c(T_j)$ is similar to that of $G_a(T_j)$.

After an insertion or deletion of a character, we check whether $|G(T_j)|$ is in the range of $\log^2 n$ to $4 \log^2 n$ bits. We need to split or merge the code blocks out of the range. Since we assume $G(T_j)$ is encoded or decoded in $O(|T_j|)$ time, we spend $O(\log^2 n)$ in worst case. From the following lemma, this $O(\log^2 n)$ update time can be amortized on $\log n$ insertion or deletion queries on T_j. Therefore, we can split or merge code blocks in $O(\log n)$ amortized time.

Lemma 3.2. *If the length,* $|G(T_j)|$, *of a code block is changed by* $\log^2 n$ *bits, then there are* $\log n$ *updating queries on* T_j.

Proof. We first consider the case of inserting a character c at position i. For character c, this insertion causes to add a code of c at position i and to decrease the distance of the code of c at the next position of i. For the codes of other characters next to i, we increase the distance values by one and the code lengths by one at most. In the worst case, the total increasing length of the codes is $\sigma + O(\log \log n) < O(\log n)$. The case of deleting a character makes $O(\log n)$ bits decreasing in the code lengths. Hence, if $|G(T_j)|$ is increased by $\log^2 n$ bits, then there are at least $\log n$ insertions of characters. The decreasing of $|G(T_j)|$ is similar.

To complete our dynamic representation of texts, we will mention two issues of our structure. One is the block allocation problem. If we allocate a code block by a fixed chunk of $4 \log^2 n$ bits, we might waste quadruple space in worst case, because the encoding of T_j could take only $\log^2 n$ bits. This problem is solved by Mäkinen and Navarro's dynamic bit vector [Mäkinen and Navarro 2006] which introduces a sub-block of $\log^{3/2} n$ bits and manages a code block as $\sqrt{\log n}$ sub-blocks to obtain $o(1)$ factor in the waste space. This scheme is also applied to the rank/select structures for a $\log n$-size alphabet [Gonzalez and Navarro 2008; Lee and Park 2007]. We omit the details and refer to these versions.

The other issue is the change of $\log n$, which causes the changes of the code block size and the table sizes. We simply rebuild the total structure when n becomes $2n$ or $n/2$. The $o(n)$ entries of the tables G, Pos, $DPos$, and H can be extended or contracted by one bit in linear time. The total encoding or decoding time is also linear in the length of text, $|T|$, and the auxiliary structures are built in $O(n)$.

4. OVER-BLOCK RANK/SELECT STRUCTURES

In the previous section we showed that the rank or select on T_j can be answered by scanning $G(T_j)$. To complete the answer of rank/select queries on T, we need over-block structures which count the number of occurrences of a certain character in the previous blocks of T_j. We denote by n_i^c, the number of occurrences of character c in T_i for $1 \le i \le$ m. Given a sequence of n_i^c, we can employ two kinds of structures as the over-block rank/select structures. One is the dynamic searchable partial sum structure [Raman et al. 2001; Hon et al. 2003] and the other is the dynamic bit vector with rank/select [Mäkinen and Navarro 2006].

The searchable partial sum structures store the sequence of n_i^c and answer two queries: rank and select [Raman et al. 2001; Hon et al. 2003]. The rank query (c, j) of the partial sum structures returns the sum of n_i^c with $i < j$, which is the number of occurrences of c before T_j. The select query (c, k) returns the maximum j such that $\sum_{i=1}^{k} n_i^c \le k$. For example $T = T_1T_2T_3T_4$ and $T_1 = \mathtt{abac}$, $T_2 = \mathtt{babc}$, $T_3 = \mathtt{ccaa}$, $T_4 = \mathtt{bbca}$, the numbers of occurrences of characters are $n^{\mathrm{a}} = (2, 1, 2, 1)$, $n^{\mathrm{b}} = (1, 2, 0, 2)$ and $n^{\mathrm{c}} = (1, 1, 2, 1)$. The rank of b before T_3 is $\sum_{i=1}^{2} n_i^{\mathrm{b}} = 3$ and the select of the 4-th b is the maximum j such that $\sum_{i=1}^{j} n_i^{\mathrm{b}} \le 4$, which is 3. The original partial sum problem has only one sequence, but here we have a sequence for each character.

Our key observation is that we can simply build independent σ partial sum structures, one for each character. Because $|T_j| \le 4\log^2 n$, the n_i^c values can be represented by a simple binary form of $O(\log\log n)$ bits. Recall that the total number of blocks, m, is $O(\frac{n\log\log n}{\log^2 n})$ for $\sigma \le \log n$. The partial sum structures take a space linear in the sequence for each character, and therefore the total size is $\sigma m \cdot O(\log\log n)$ bits. For $\sigma \le \log n$, the total size is $O(\frac{n\log\log^2 n}{\log n}) = o(n)$. The partial sum structures is constructed in $O(\sigma \mathrm{m}) = o(n)$ time, too.

The $\log n$-size alphabet enables us to amortize the cost of block split/merge on the insertions or deletions on T_j. An insertion or deletion of a character c needs $O(1)$ update queries only on the sequence n^c, and a block split or merge triggers total $O(\sigma) = O(\log n)$ update queries, $O(1)$ queries for each sequence n^c. From Lemma 3.2, the split or merge of T_j occurs only if there are $\log n$ insertions or deletions in T_j, so we have amortized $O(1)$ update queries for one split or merge.

In fact, the searchable partial sum problem is related to the rank/select on bit vectors [Raman et al. 2001; Hon et al. 2003; Mäkinen and Navarro 2008]. In [Lee and Park 2007], all sequences of n^c are represented by a bit vector B and the over-block rank/select queries are processed by the binary rank/select on B. The n_i^c values of each sequence is unary-coded by a single 1 and following n_i^c 0s. For the above example, the sequence $n^{\mathrm{a}} = (2, 1, 2, 1)$ is represented by $B^{\mathrm{a}} = 1001010010$. The sequences n^{b} and n^{c} are represented by $B^{\mathrm{b}} = 101001100$ and $B^{\mathrm{c}} = 101010010$. B is the concatenation of B_{a}, B_{b}, and B_{c}. The size of B is $n + \sigma m = O(n)$ bits, but the compressed bit vector B by Mäkinen and Navarro [Mäkinen and Navarro 2006; 2008]

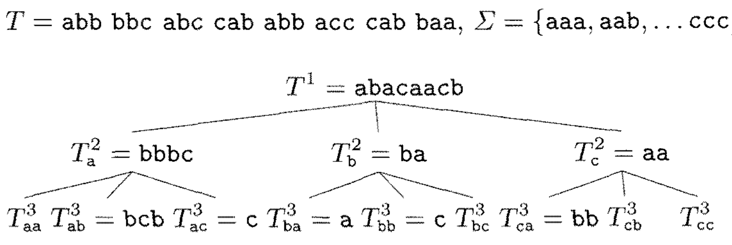

$T = \texttt{abb bbc abc cab abb acc cab baa},\ \Sigma = \{\texttt{aaa}, \texttt{aab}, \ldots \texttt{ccc}\}$
$T^1 = \texttt{abacaacb}$
$T^2_{\texttt{a}} = \texttt{bbbc}$
$T^2_{\texttt{b}} = \texttt{ba}$
$T^2_{\texttt{c}} = \texttt{aa}$
$T^3_{\texttt{aa}}$
$T^3_{\texttt{ab}} = \texttt{bcb}$
$T^3_{\texttt{ac}} = \texttt{c}$
$T^3_{\texttt{ba}} = \texttt{a}$
$T^3_{\texttt{bb}} = \texttt{c}$
$T^3_{\texttt{bc}}$
$T^3_{\texttt{ca}} = \texttt{bb}$
$T^3_{\texttt{cb}}$
$T^3_{\texttt{cc}}$

Figure 5. Example of a wavelet tree.

takes $o(n)$ bits like the partial sum structures and provides worst-case $O(\log n)$ time queries.

5. EXTENSIONS FOR A LARGE ALPHABET AND APPLICATIONS

In this section we extend our structure for a large alphabet with $\sigma = O(n^e)$ and $e < 1$, and then apply the structure to the Burrows-Wheeler Transform (BWT) of texts. The extension for a large alphabet employs k-ary wavelet trees [Ferragina et al. 2007] and obtains $O((1 + \frac{\log \sigma}{\log\log n}) \log n)$ time queries in $nH_0 + o(n \log \sigma)$ bits. The application to the BWT of texts immediately supports a dynamic compressed index for a collection of texts by Chan et al. [Chan et al. 2004]. For an nH_k compression of texts, some results [Mäkinen and Navarro 2008; Ferragina et al. 2007] showed that the gap encoding with binary wavelet trees compresses the BWT of texts in $nH_k + o(n \log \sigma)$ bits. Based on these results, our extension by k-ary wavelet trees can also achieve $nH_k + o(n \log \sigma)$ bits compression with the BWT.

5.1 Wavelet Tree Extension

Given character c in a σ-size alphabet Σ, we regard c as $l = 1 + \frac{\log \sigma}{\log\log n}$ digits of a $\log n$-size alphabet Σ', i.e., $c = c_1c_2 \ldots c_l$ and $c_j \in \Sigma'$ with $|\Sigma'| = \log n$. Let T_j be the concatenation of the j-th digit of $T[i]$ for all i. The k-ary wavelet tree stores sequence T_j at the j-th level, grouped by the first j–1 digits. Let T^j_s denote a subsequence of T^j such that the j-th digit of $T[i]$ belongs to T^j_s iff $T[i]$ has the same prefix s of j–1 digits. The root of the tree contains T^1 and each of its children contains T^2_c for $c \in \Sigma'$. If a node of the j-th level contains T^j_s, then its children contain T^{j+1}_{sc} for all $c \in \Sigma'$. The leaves of the tree contain T^l grouped by its l–1 prefix digits of $T[i]$. See Figure 5.

We maintain the k-ary wavelet tree implicitly. For each j-th level, we concatenate T^j_s by the lexicographic order of s and encode this concatenation to the code blocks of $\log^2 n$ to $4 \log^2 n$ bits. The over-block rank/select structure is built for each j-th level. We build an $O(n + \sigma)$ bits vector F_j for marking the lengths of T^j_s. By using F_j, we find $G(T^j_s)$ and branch to $G(T^{j+1}_{sc})$ from $G(T^j_s)$ at the j-th level. The total size of F^j is $O((1 + \frac{\log \sigma}{\log\log n})n)$ bits.

From Ferragina et al.'s result [Ferragina et al. 2007], we can show that our extension by the k-ary wavelet tree obtains total $nH_0(T) + o(n \log \sigma)$ bits for $\sigma = O(n^e)$ with $e < 1$. By Lemma 3.1, the partition by code blocks does not introduce extra space so that each T^j_s is encoded in $(1 + o(1))|T^j_s|H_0(T^j_s) + O(|T^j_s|)$ bits except the overhead of $O(\log n \log\log n)$ bits for the starting characters of T^j_s. There are $O(\sigma)$

nodes for all T^j_s, so the total overhead is $O(\sigma \log n \log\log n)$ and it becomes $o(n)$ for $\sigma = O(n_e)$. The sum of $|T^j_s| H_0(T^j_s)$ for all T^j_s is total nH_0 bits [Ferragina et al. 2007]. The additional dynamic structures take total $o(n \log \sigma)$ bits.

For $\log \sigma$-bits character $c = c_1c_2 \ldots c_l$, the rank of c is processed by the rank of c_j on each T_j. The rank of c_1 on the first level gives the next query position for the rank on c_2, i.e., the number of characters with the first digit c_1 and the second digit c_2. The rank query is processed from the root to a leaf, and the select query is bottom up [Ferragina et al. 2007]. The access and update of $T[i]$ use the steps of the rank [Lee and Park 2007].

5.2 Application to BWT

The BWT of T, T^{bwt}, is a permutation of T made from the preceding characters of the sorted suffixes of T. In the BWT of T, there are groups of the characters that share the same context Σ^k, and therefore there is a partition of $T^{bwt} = T^{bwt}_1 T^{bwt}_2 \ldots T^{bwt}_t$ with $t \leq \sigma^k$ such that $nH_k(T) = \Sigma^k_{i=1} |T^{bwt}_i| H_0(T^{bwt}_i)$ [Manzini 2001]. In other words, an H_0 compressor which keeps the local entropy $H_0(T^{bwt}_i)$ is an H_k compressor of T.

Our representation for a $\log n$-size alphabet can be an H_k compressor. In fact, the gap encoding methods are widely used for static nH_k compressions of full text indices [Grossi et al. 2004; Mäkinen and Navarro 2008; Sadakane 2003]. We can also obtain a dynamic structure of $nH_k(T) + o(n \log \sigma) + \sigma^{k+1} \log\log n$ bits. For any partition of T^{bwt}, $|G(T^{bwt}_i)| = (1 + o(1)) |T^{bwt}_i| H_0(T^{bwt}_i) + O(|T^{bwt}_i|)$ by Lemma 3.1. The encodings of the starting characters of T^{bwt}_i have $\sigma \log\log n$ bits overhead, and the total sum becomes $\sigma^{k+1} \log\log n$. For $k \leq (\alpha \log_\sigma n) - 1$ and $0 < \alpha < 1$, the size of our structure can be $nH_k(T) + o(n \log \sigma)$ bits.

For our extension by the k-ary wavelet tree, we follow Mäkinen and Navarro's result [Mäkinen and Navarro 2008] which shows that the binary wavelet tree with Raman et al.'s encoding [Raman et al. 2002] preserves the local entropy of T^{bwt}_i. Because we employ the k-ary wavelet tree with the gap encoding, there are some differences in the details, but the final result is the same.

We want to show that given any partition of $T = T_1T_2 \ldots T_t$, the k-ary wavelet tree with the gap encoding compresses T_j in space of $|T_j| H_0(T_j) + o(|T_j| \log \sigma) + O(\sigma \log n \log\log n)$ bits. Like the binary wavelet tree case, the decomposition of T_j can be regarded as an independent k-ary wavelet tree of T_j plus overhead. The k-ary wavelet tree enables T_j to be encoded in $|T_j| H_0(T_j) + o(|T_j| \log \sigma)$ bits. By comparing the implicit tree with the decomposition of T_j, the overhead is the encoding of the starting characters at each node, which is $O(\log n \log\log n)$. The number of the nodes in the wavelet tree of T_j is $O(\sigma)$, so the total overhead is $O(\sigma \log n \log\log n)$. Hence, the total size of the encoding of any partition of T^{bwt} is bounded by $nH_k(T) + o(n \log \sigma) + \sigma^{k+1} \log n \log\log n$ bits. For $k \leq (\alpha \log_\sigma n) - 1$ and $0 < \alpha < 1$, this can be $nH_k(T) + o(n \log \sigma)$ bits.

6. CONCLUSION

We have presented a dynamic and compressed representation of texts, which provides retrieving queries of rank/select/access and updating queries of insert/delete. This is an improvement upon our previous result [Lee and Park 2007] by compressing a text in its empirical entropy bound. Comparing with [González and Navarro 2008], our representation uses rather simple techniques to obtain a compressed space and fast rank/select time.

ACKNOWLEDGMENTS

This study was supported by Grants FPR08-A1-021 of the 21C Frontier Functional Proteomics Project from the Korean Ministry of Education, Science & Technology.

REFERENCES

CHAN, H.-L., W.-K. HON, AND T.-W. LAM. 2004. Compressed index for a dynamic collection of texts. In *Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching.* 445–456.

ELIAS, P. 1975. Universal codeword sets and representation of the integers. *IEEE Transactions on Information Theory* 21(2):194–203.

FERRAGINA, P. AND G. MANZINI. 2005. Indexing compressed text. *Journal of ACM* 52(4):552–581.

FERRAGINA, P., G. MANZINI, V. MÄKINEN, AND G. NAVARRO. 2007. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms 3*, 2.

GONZÁLEZ, R. AND G. NAVARRO. 2008. Improved dynamic rank-select entropy-bound structures. In *Proceedings of the 8th Latin American Symposium on Theoretical Informatics*. To Appear.

GROSSI, R., A. GUPTA, AND J. S. VITTER. 2003. High-order entropy-compresssed text indexes. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms.* 841–850.

GROSSI, R., A. GUPTA, AND J. S. VITTER. 2004. When indexing equals compression: experiments with compressing suffix arrays and applications. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms.* 636–645.

GROSSI, R. AND J. S. VITTER. 2005. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing* 35(2):378–407.

GUPTA, A., W.-K. HON, R. SHAH, AND J. S. VITTER. 2007. A framework for dynamizing succinct data structures. In *Proceedings of the 34th International Colloquium on Automata, Languages and Programming.* 521–532.

HON, W.-K., K. SADAKANE, AND W.-K. SUNG. 2003. Succinct data structures for searchable partial sums. In *Proceedings of the 14th Annual Symposium on Algorithms and Computation.* 505–516.

LEE, S. AND K. PARK. 2007. Dynamic rank-select structures with applications to run-length encoded texts. In *Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching.* 95–106.

MÄKINEN, V. AND G. NAVARRO. 2006. Dynamic entropy-compressed sequences and full-text indexes. In *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching.* 306–317.

MÄKINEN, V. AND G. NAVARRO. 2007. Rank and select revisited and extended. *Theoretical Computer Science* 387(3):332–347.

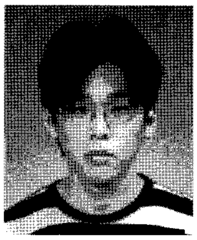
MÄKINEN, V. AND G. NAVARRO. 2008. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms.* To Appear.

MANZINI, G. 2001. An analysis of the burrows-wheeler transform. *Journal of ACM* 48(3):407–430.

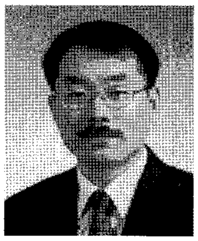
RAMAN, R., V. RAMAN, AND S. S. RAO. 2001. Succinct dynamic data structures. In *Proceedings of the 7th International Workshop on Algorithms and Data Structures.* 426–437.

RAMAN, R., V. RAMAN, AND S. S. RAO. 2002. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms.* 233–242.

SADAKANE, K. 2003. New text indexing functionalites of the compressed suffix arrays. *Journal of Algorithms* 48(2):294–313.



Sunho Lee 2009. ~ present. BK21 postdoctoral fellow, Hanyang University
2009. Ph. D. in Computer Science and Engineering, Seoul National University
2002. B. S. in Computer Science and Engineering, Seoul National University
Research interests include algorithms and data structures for string processing, and applications to bioinformatics.



Kunsoo Park 1993. 8 ~ present. Professor, Seoul National University
2005. 1. ~ 2005. 2. Visiting professor, University of Marne-la-Vallee
1995. 7. ~ 1995. 8. Visiting research fellow, Curtin University
1991. 11. ~ 1993. 8. Lecturer, King's College, University of London
1991. Ph. D. in Computer Science, Columbia University
1985. M. S. in Computer Engineering, Seoul National University
1983. B. S. in Computer Engineering, Seoul National University
Research interests are design and analysis of algorithms, especially for cryptography and bioinformatics.