Compressed Data Structures for Dynamic Sequences

J. Ian Munro and Yakov Nekrich

David R. Cheriton School of Computer Science, University of Waterloo

Abstract. We consider the problem of storing a dynamic string S over an alphabet $\Sigma = \{1, \ldots, \sigma\}$ in compressed form. Our representation supports insertions and deletions of symbols and answers three fundamental queries: $\mathrm{access}(i, S)$ returns the i-th symbol in S, $\mathrm{rank}_a(i, S)$ counts how many times a symbol a occurs among the first i positions in S, and $\mathrm{select}_a(i, S)$ finds the position where a symbol a occurs for the i-th time. We present the first fully-dynamic data structure for arbitrarily large alphabets that achieves optimal query times for all three operations and supports updates with worst-case time guarantees. Ours is also the first fully-dynamic data structure that needs only $nH_k + o(n \log \sigma)$ bits, where H_k is the k-th order entropy and n is the string length. Moreover our representation supports extraction of a substring $S[i..i + \ell]$ in optimal $O(\log n / \log \log n + \ell / \log_\sigma n)$ time.

1 Introduction

In this paper we consider the problem of storing a sequence S of length n over an alphabet $\Sigma = \{1, \ldots, \sigma\}$ so that the following operations are supported:
- $\mathrm{access}(i, S)$ returns the i-th symbol, $S[i]$, in S
- $\mathrm{rank}_a(i, S)$ counts how many times a occurs among the first i symbols in S, $\mathrm{rank}_a(i, S) = |\{ j \mid S[j] = a \text{ and } 1 \le j \le i \}|$
-$\mathrm{select}_a(i, S)$ finds the position in S where a occurs for the i-th time, $\mathrm{select}_a(i, S) = j$ where j is such that $S[j] = a$ and $\mathrm{rank}_a(j, S) = i$.

This problem, also known as the rank-select problem, is one of the most fundamental problems in compressed data structures. There are many data structures that store a string in compressed form and support three above defined operations efficiently. There are static data structures that use $nH_0 + o(n \log \sigma)$ bits or even $nH_k + o(n \log \sigma)$ bits for any $k \le \alpha \log_\sigma n - 1$ and a positive constant $\alpha < 1$[1]. Efficient static rank-select data structures are described in [11,10,8,18,19,2,14,26,4]. We refer to [4] for most recent results and a discussion of previous static solutions.

[1] Henceforth $H_0(S) = \sum_{a \in \Sigma} \frac{n_a}{n} \log \frac{n}{n_a}$, where n_a is the number of times a occurs in S, is the 0-th order entropy and $H_k(S)$ for $k \ge 0$ is the k-th order empirical entropy. $H_k(S)$ can be defined as $H_k(S) = \sum_{A \in \Sigma^k} |S_A| H_0(S_A)$, where S_A is the subsequence of S generated by symbols that follow the k-tuple A; $H_k(S)$ is the lower bound on the average space usage of any statistical compression method that encodes each symbol using the context of k previous symbols [22].

N. Bansal and I. Finocchi (Eds.): ESA 2015, LNCS 9294, pp. 891–902, 2015.
DOI: 10.1007/978-3-662-48350-3_74

In many situations we must work with dynamic sequences. We must be able to insert a new symbol at an arbitrary position i in the sequence or delete an arbitrary symbol $S[i]$. The design of dynamic solutions, that support insertions and deletions of symbols, is an important problem. Fully-dynamic data structures for rank-select problem were considered in [15,7,5,20,6,13,21,16]. Recently Navarro and Nekrich [24,25] obtained a fully-dynamic solution with $O(\log n/\log\log n)$ times for rank, access, and select operations. By the lower bound of Fredman and Saks [9], these query times are optimal. The data structure described in [24] uses $nH_0(S) + o(n \log \sigma)$ bits and supports updates in $O(\log n/\log\log n)$ amortized time. It is also possible to support updates in $O(\log n)$ worst-case time, but then the time for answering a rank query grows to $O(\log n)$ [25]. All previously known fully-dynamic data structures need at least $nH_0(S) + o(n \log \sigma)$ bits. Two only exceptions are data structures of Jansson et al. [17] and Grossi et al. [12] that keep S in $nH_k(S) + o(n \log \sigma)$ bits, but do not support rank and select queries. A more restrictive dynamic scenario was considered by Grossi et al. [12] and Jansson et al. [17]: an update operation *replaces* a symbol $S[i]$ with another symbol so that the total length of S does not change, but insertions of new symbols or deletions of symbols of S are not supported. Their data structures need $nH_k(S) + o(n \log \sigma)$ bits and answer access queries in $O(1)$ time; the data structure of Grossi et al. [12] also supports rank and select queries in $O(\log n/\log\log n)$ time.

In this paper we describe the first fully-dynamic data structure that keeps the input sequence in $nH_k(S) + o(n \log \sigma)$ bits; our representation supports rank, select, and access queries in optimal $O(\log n/\log\log n)$ time. Symbol insertions and deletions at any position in S are supported in $O(\log n/\log\log n)$ worst-case time. We list our and previous results for fully-dynamic sequences in Table 1. Our representation of dynamic sequences also supports the operation of extracting a substring. Previous dynamic data structures require $O(\ell)$ calls of access operation in order to extract the substring of length ℓ. Thus the previous best fully-dynamic representation, described in [24] needs $O(\ell(\log n/\log\log n))$ time to extract a substring $S[i..i + \ell - 1]$ of S. Data structures described in [12] and [17] support substring extraction in $O(\log n/\log\log n + \ell/\log_\sigma n)$ time but they either do not support rank and select queries or they support only updates that replace a symbol with another symbol. Our dynamic data structure can extract a substring in optimal $O(\log n/\log\log n + \ell/\log_\sigma n)$ time without any restrictions on updates or queries.

In Section 2 we describe a data structure that uses $O(\log n)$ bits per symbol and supports rank, select, and access in optimal $O(\log n/\log\log n)$ time. This data structure essentially maintains a linked list L containing all symbols of S; using some auxiliary data structures on L, we can answer rank, select, and access queries on S. In Section 3 we show how the space usage can be reduced to $O(\log \sigma)$ bits per symbol. A compressed data structure that needs $H_0(S)$ bits per symbol is presented in Section 4. The approach of Section 4 is based on dividing S into a number of subsequences. We store a fully-dynamic data structure for only one such subsequence of appropriately small size. Updates on

Table 1. Previous and New Results for Fully-Dynamic Sequences. The rightmost column indicates whether updates are amortized (A) or worst-case (W). We use notation $\lambda = \log n / \log\log n$ in this table.

Ref.	Space	Rank	Select	Access	Insert/ Delete	
[14]	$nH_0(S) + o(n\log\sigma)$	$O((1 + \log\sigma/\log\log n)\lambda)$			$O((1 + \log\sigma/\log\log n)\lambda)$	W
[26]	$nH_0(S) + o(n\log\sigma)$	$O((\log\sigma/\log\log n)\lambda)$			$O((\log\sigma/\log\log n)\lambda)$	W
[24]	$nH_0(S) + o(n\log\sigma)$	$O(\lambda)$	$O(\lambda)$	$O(\lambda)$	$O(\lambda)$	A
[24]	$nH_0(S) + o(n\log\sigma)$	$O(\log n)$	$O(\lambda)$	$O(\lambda)$	$O(\log n)$	W
[17]	$nH_k + o(n\log\sigma)$	-	-	$O(\lambda)$	$O(\lambda)$	W
[12]	$nH_k + o(n\log\sigma)$	-	-	$O(\lambda)$	$O(\lambda)$	W
New	$nH_k + o(n\log\sigma)$	$O(\lambda)$	$O(\lambda)$	$O(\lambda)$	$O(\lambda)$	W

other subsequences are supported by periodic re-building. In Section 5 we show that the space usage can be reduced to $nH_k(S) + o(n\log\sigma)$.

2 $O(n\log n)$-Bit Data Structure

We start by describing a data structure that uses $O(\log n)$ bits per symbol.

Lemma 1. *A dynamic string $S[1,m]$ for $m \le n$ over alphabet $\Sigma = \{1,\ldots,\sigma\}$ can be stored in a data structure that needs $O(m\log m)$ bits, and answers queries* access, rank *and* select *in time $O(\log m/\log\log n)$. Insertions and deletions of symbols are supported in $O(\log m/\log\log n)$ time. The data structure uses a universal look-up table of size $o(n^{\varepsilon})$ for an arbitrarily small $\varepsilon > 0$.*

Proof: We keep elements of S in a list L. Each entry of L contains a symbol $a \in \Sigma$. For every $a \in \Sigma$, we also maintain the list L_a. Entries of L_a correspond to those entries of L that contain the symbol a. We maintain data structures $D(L)$ and $D(L_a)$ that enable us to find the number of entries in L (or in some list L_a) that precede an entry $e \in L$ (resp. $e \in L_a$); we can also find the i-th entry e in L_a or L using $D(L.)$. We will prove in Lemma 4 that $D(L)$ needs $O(m\log m)$ bits and supports queries and updates on L in $O(\log m/\log\log n)$ time.

We can answer a query $\text{select}_a(i,S)$ by finding the i-th entry e_i in L_a, following the pointer from e_i to the corresponding entry $e' \in L$, and counting the number v of entries preceding e' in L. Clearly[2], $\text{select}_a(i,S) = v$. To answer a query $\text{rank}_a(i,S)$, we first find the i-th entry e in L. Then we find the last entry e_a that precedes e and contains a. Such queries can be answered in $O((\log\log\sigma)^2\log\log m)$ time as will be shown in the full version of this paper [23]. If e'_a is the entry that corresponds to e_a in L_a, then $\text{rank}_a(i,S) = v$, where v is the number of entries that precede e'_a in L_a. □

[2] To simplify the description, we assume that a list entry precedes itself.

3 $O(n \log \sigma)$-Bit Data Structure

Lemma 2. *A dynamic string $S[1,n]$ over alphabet $\Sigma = \{1, \ldots, \sigma\}$ can be stored in a data structure using $O(n \log \sigma)$ bits, and supporting queries* access, rank *and* select *in time $O(\log n / \log\log n)$. Insertions and deletions of symbols are supported in $O(\log n / \log\log n)$ time.*

Proof: If $\sigma = \log^{O(1)} n$, then the data structures described in [26] and [14] provide desired query and update times. The case $\sigma = \log^{\Omega(1)} n$ is considered below.

We show how the problem on a sequence of size n can be reduced to the same problem on a sequence of size $O(\sigma \log n)$. The sequence S is divided into chunks. We can maintain the size n_i of each chunk C_i, so that $n_i = O(\sigma \log n)$ and the total number of chunks is bounded by $O(n/\sigma)$. We will show how to maintain chunks in the full version of this paper [23]. For each $a \in \Sigma$, we keep a global bit sequence B_a. $B_a = 1^{d_1}01^{d_2}0 \ldots 1^{d_i}0 \ldots$ where d_i is the number of times a occurs in the chunk C_i. We also keep a bit sequence $B_t = 1^{n_1}01^{n_2}0 \ldots 1^{n_i}0 \ldots$. We can compute $\text{rank}_a(i, S) = v_1 + v_2$ where $v_1 = \text{rank}_1(\text{select}_0(j_1, B_a), B_a)$, $j_1 = \text{rank}_0(\text{select}_1(i, B_t), B_t)$, $v_2 = \text{rank}_a(i_1, C_{i_2})$, $i_2 = j_1 + 1$ and $i_1 = i - \text{rank}_1(\text{select}_0(j_1, B_t), B_t)$. To answer a query $\text{select}_a(i, S)$, we first find the index i_2 of the chunk C_{i_2} that contains the i-th occurrence of i, $i_2 = \text{rank}_0(\text{select}_1(i, B_a), B_a) + 1$. Then we find $v_a = \text{select}_a(C_{i_2}, i - i_1)$ for $i_1 = \text{rank}_1(\text{select}_0(i_2 - 1, B_a), B_a)$; v_a identifies the position of the $(i - i_1)$-th occurrence of a in the chunk C_{i_2}, where i_1 denotes the number of a's in the first $i_2 - 1$ chunks. Finally we compute $\text{select}_a(i, S) = v_a + s_p$ where $s_p = \text{rank}_1(\text{select}_0(i_2 - 1, B_t), B_t)$ is the total number of symbols in the first $i_2 - 1$ chunks. We can support queries and updates on B_t and on each B_a in $O(\log n / \log\log n)$ time [26]. By Lemma 1, queries and updates on C_i are supported in $O(\log \sigma / \log\log n)$ time. Hence, the query and update times of our data structure are $O(\log n / \log\log n)$.

B_t can be kept in $O((n/\sigma)\log \sigma)$ bits [26]. The array B_a uses $O(n_a \log \frac{n}{n_a})$ bits, where n_a is the number of times a occurs in S. Hence, all B_a and B_t use $O((n/\sigma)\log \sigma + \sum_a n_a \log \frac{n}{n_a}) = O(n \log \sigma)$ bits. By Lemma 1, we can also keep the data structure for each chunk in $O(\log \sigma + \log\log n) = O(\log \sigma)$ bits per symbol. $\square$

4 Compressed Data Structure

In this Section we describe a data structure that uses $H_0(S)$ bits per symbol. We start by considering the case when the alphabet size is not too large, $\sigma \leq n/\log^3 n$. The sequence S is split into subsequences S_0, S_1, $\ldots$ S_r for $r = O(\log n/(\log\log n))$. The subsequence S_0 is stored in $O(\log \sigma)$ bits per element as described in Lemma 2. Subsequences $S_1, \ldots S_r$ are substrings of $S \setminus S_0$. $S_1, \ldots S_r$ are stored in compressed static data structures. New elements are always inserted into the subsequence S_0. Deletions from S_i, $i \geq 1$, are implemented as lazy deletions: an element in S_i is marked as deleted. We guarantee that the number of elements that are marked as deleted is bounded by $O(n/r)$. If a subsequence S_i contains many elements marked as deleted, it is re-built: we create a new instance of S_i that does not contain deleted symbols. If a symbol sequence S_0 contains too many elements, we insert the elements of S_0 into S_i and re-build S_i for $i \geq 1$. Processes of constructing a new subsequence and re-building a subsequence with too many obsolete elements are run in the background.

The bit sequence M identifies elements in S that are marked as deleted: $M[j] = 0$ if and only if $S[j]$ is marked as deleted. The bit sequence R distinguishes between the elements of S_0 and elements of S_i, $i \geq 1$: $R[j] = 0$ if the j-th element of S is kept in S_0 and $R[j] = 1$ otherwise.

We further need auxiliary data structures for answering select queries. We start by defining an auxiliary subsequence $\tilde{S}$ that contains copies of elements already stored in other subsequences. Consider a subsequence $\overline{S}$ obtained by merging subsequences S_1, ..., S_r (in other words, $\overline{S}$ is obtained from S by removing elements of S_0). Let S'_a be the subsequence obtained by selecting (roughly) every r-th occurrence of a symbol a in $\overline{S}$. The subsequence S' is obtained by merging subsequences S'_a for all $a \in \Sigma$. Finally $\tilde{S}$ is obtained by merging S' and S_0. We support queries $\mathrm{select}'_a(i, \tilde{S})$ on $\tilde{S}$, defined as follows: $\mathrm{select}'_a(i, \tilde{S}) = j$ such that (i) a copy of $S[j]$ is stored in $\tilde{S}$ and (ii) if $\mathrm{select}_a(i, S) = j_1$, then $j \leq j_1$ and copies of elements $S[j+1]$, $S[j+2]$, ..., $S[j_1]$ are not stored in $\tilde{S}$. That is, $\mathrm{select}'_a(i, \tilde{S})$ returns the largest index j, such that $S[j]$ precedes $S[\mathrm{select}_a(i, S)]$ and $S[j]$ is also stored in $\tilde{S}$. The data structure for $\tilde{S}$ delivers approximate answers for select queries; we will show later how the answer to a query $\mathrm{select}_a(i, S)$ can be found quickly if the answer to $\mathrm{select}'_a(i, \tilde{S})$ is known. Queries $\mathrm{select}'(i, \tilde{S})$ can be implemented using standard operations on a bit sequence of size $O((n/r) \log\log n)$ bits; for completeness, we provide a description in the full version of this paper [23]. We remark that $\overline{S}$ and S' are introduced to define $\tilde{S}$; these two subsequences are not stored in our data structure. The bit sequence $\tilde{E}$ indicates what symbols of S are also stored in $\tilde{S}$: $\tilde{E}[i] = 1$ if a copy of $S[i]$ is stored in $\tilde{S}$ and $\tilde{E}[i] = 0$ otherwise. The bit sequence $\tilde{B}$ indicates what symbols in $\tilde{S}$ are actually from S_0: $\tilde{B}[i] = 0$ iff $\tilde{S}[i]$ is stored in the subsequence S_0. Besides, we keep bit sequences D_a for each $a \in \Sigma$. Bits of D_a correspond to occurrences of a in S. If the l-th occurrence of a in S is marked as deleted, then $D_a[l] = 0$. All other bits in D_a are set to 1.

We provide the list of subsequences in Table 2. Each subsequence is augmented with a data structure that supports rank and select queries. For simplicity we will not distinguish between a subsequence and a data structure on its elements. If a subsequence supports updates, then either (i) this is a subsequence over a small alphabet or (ii) this subsequence contains a small number of elements. In case (i), the subsequence is over an alphabet of constant size; by [26,14] queries on such subsequences are answered in $O(\log n / \log\log n)$ time. In case (ii) the subsequence contains $O(n/r)$ elements; data structures on such subsequences are implemented as in Lemma 2. All auxiliary subsequences, except for $\tilde{S}$, are of type (i). Subsequence S_0 and an auxiliary subsequence $\tilde{S}$ are of type (ii). Subsequences S_i for $i \geq 1$ are static, i.e. they are stored in data structures that do not support

Table 2. Auxiliary subsequences for answering rank and select queries. A subsequence is dynamic if both insertions and deletions are supported. If a subsequence is static, then updates are not supported. Static subsequences are re-built when they contain too many obsolete elements.

Name	Purpose	Alph. Size	Dynamic/ Static
S_0	Subsequence of S	-	Dynamic
S_i, $1 \le i \le r$	Subsequence of S	-	Static
M	Positions of symbols in S_i, $i \ge 1$, that are marked as deleted	const	Dynamic
R	Positions of symbols from S_0 in S	const	Dynamic
$\tilde{S}$	Delivers an approximate answer to select queries	-	Dynamic
S'_a, $a \in \Sigma$	Auxiliary sequences for $\tilde{S}$	-	Dynamic
$\tilde{E}$	Positions of symbols from $\tilde{S}$ in S	const	Dynamic
$\tilde{B}$	Positions of symbols from S_0 in $\tilde{S}$	const	Dynamic
D_a	Positions of symbols marked as deleted among all a's	const	Dynamic

updates. We re-build these subsequences when they contain too many obsolete elements. Thus dynamic subsequences support rank, select, access, and updates in $O(\log n/\log\log n)$ time. It is known that we can implement all basic operations on a static sequence in $O(\log n/\log\log n)$ time[3]. Our data structures on static subsequences are based on the approach of Barbay et al. [3]; however, our data structure can be constructed faster when the alphabet size is small and supports a substring extraction operation. A full description will be given in the full version of this paper [23]. We will show below that queries on S are answered by $O(1)$ queries on dynamic subsequences and $O(1)$ queries on static subsequences.

We also maintain arrays $Size[]$ and $Count_a[]$ for every $a \in \Sigma$. For any $1 \le i \le r$, $Size[i]$ is the number of symbols in S_i and $Count_a[i]$ specifies how many times a occurs in S_i. We keep a data structure that computes the sum of the first $i \le r$ entries in $Size[i]$ and find the largest j such that $\sum_{t=1}^{j} Size[t] \le q$ for any integer q. The same kinds of queries are also supported on $Count_a[]$. Arrays $Size[]$ and $Count_a[]$ use $O(\sigma \cdot r \cdot \log n) = O(n/\log n)$ bits.

Queries. To answer a query $\mathrm{rank}_a(i, S)$, we start by computing $i' = \mathrm{select}_1(i, M)$; i' is the position of the i-th element that is not marked as deleted. Then we find $i_0 = \mathrm{rank}_0(i', R)$ and $i_1 = \mathrm{rank}_1(i', R)$. By definition of R, i_0 is the number of elements of $S[1..i]$ that are stored in the subsequence S_0. The number of a's in $S_0[1..i_0]$ is computed as $c_1 = \mathrm{rank}_a(i_0, S_0)$. The number of a's in $S_1, \ldots, S_r$ before the position i' is found as follows. We identify the index t, such that $\sum_{j=1}^{t} Size[j] < i_1 \le \sum_{j=1}^{t+1} Size[j]$. Then we compute how many times a occurred in $S_1, \ldots, S_t$, $c_{2,1} = \sum_{j=1}^{t} Count_a[j]$, and in the relevant prefix of S_{t+1}, $c_{2,2} = \mathrm{rank}_a(i_1 - \sum_{j=1}^{t} Size[j], S_{t+1})$. Let $c_2 = \mathrm{rank}_1(c_{2,1} + c_{2,2}, D_a)$. Thus c_2 is the number of symbols 'a' that are not marked as deleted among the first $c_{2,1} + c_{2,2}$ occurrences of a in $S \setminus S_0$. Hence $\mathrm{rank}_a(i, S) = c_1 + c_2$.

[3] Static data structures also achieve significantly faster query times, but this is not necessary for our implementation.

To answer a query $\mathrm{select}_a(i, S)$, we first obtain an approximate answer by asking a query $\mathrm{select}'_a(i, \tilde{S})$. Let $i' = \mathrm{select}_1(i, D_a)$ be the rank of the i-th symbol a that is not marked as deleted. Let $l_0 = \mathrm{select}'_a(i', \tilde{S})$. We find $l_1 = \mathrm{rank}_1(l_0, \tilde{E})$ and $l_2 = \mathrm{select}_a(\mathrm{rank}_a(l_1, \tilde{S}) + 1, \tilde{S})$. Let $first = \mathrm{select}_1(l_1, \tilde{E})$ and $last = \mathrm{select}_1(l_2, \tilde{E})$ be the positions of $\tilde{S}[l_1]$ and $\tilde{S}[l_2]$ in S. By definition of select', $\mathrm{rank}_a(first, S) \le i$ and $\mathrm{rank}_a(last, S) > i$. If $\mathrm{rank}_a(first, S) = i$, then obviously $\mathrm{select}_a(i, S) = first$. Otherwise the answer to $\mathrm{select}_a(i, S)$ is an integer between $first$ and $last$. By definition of $\tilde{S}$, the substring $S[first]$, $S[first+1]$, $\ldots$, $S[last]$ contains at most r occurrences of a. All these occurrences are stored in subsequences S_j for $j \ge 1$. We compute $i_0 = \mathrm{rank}_a(\mathrm{rank}_0(first, R), S_0)$ and $i_1 = i' - i_0$. We find the index t such that $\sum_{j=1}^{t-1} Count_a[j] < i_1 \le \sum_{j=1}^{t} Count_a[j]$. Then $v_1 = \mathrm{select}_a(i_1 - \sum_{j=1}^{t-1} Count_a[j], S_t)$ is the position of $S[\mathrm{select}_a(i, S)]$ in S_t. We find its index in S by computing $v_2 = v_1 + \sum_{j=1}^{t-1} Size[j]$ and $v_3 = \mathrm{select}_1(v_2, R)$. Finally $\mathrm{select}_a(i, S) = \mathrm{rank}_1(v_3, M)$.

Answering an access query is straightforward. We determine whether $S[i]$ is stored in S_0 or in some S_j for $j \ge 1$ using R. Let $i' = \mathrm{select}_1(i, M)$. If $R[i'] = 0$ and $S[i]$ is stored in S_0, then $S[i] = S_0[\mathrm{rank}_0(i', R)]$. If $R[i'] = 1$, we compute $i_1 = \mathrm{rank}_1(i', R)$ and find the index j such that $\sum_{t=1}^{j-1} Size[t] < i_1 \le \sum_{t=1}^{j} Size[t]$. The answer to $\mathrm{access}(i, S)$ is $S[i] = S_j[i_2]$ for $i_2 = i_1 - \sum_{t=1}^{j-1} Size[t]$.

Space Usage. The redundancy of our data structure can be estimated as follows. The space needed to keep the symbols that are marked as deleted in subsequences S_j is bounded by $O((n/r)\log\sigma)$. S_0 also takes $O((n/r)\log\sigma)$ bits. The bit sequences R and M need $O((n/r)\log r) = o(n)$ bits; $\tilde{B}$, $\tilde{E}$ also use $O((n/r)\log r)$ bits. Each bit sequence D_a can be maintained in $O(n'_a \log(n_a/n'_a))$ bits where n_a is the total number of symbols a in S and n'_a is the number of symbols a that are marked as deleted. All D_a take $O(\sum n'_a \log \frac{n_a}{n'_a})$; the last expression can be bounded by $O((n/r)(\log r + \log\sigma))$. The subsequence $\tilde{S}$ can be stored in $O((n/r)\log\sigma)$ bits. Thus all auxiliary subsequences use $O((n/r)(\log\sigma + \log r)) = o(n\log\sigma)$ bits. Data structures for subsequences S_i, $r \ge i \ge 1$, use $\sum_{i=1}^{r}(n_i H_k(S_i) + o(n_i \log\sigma)) = nH_k(S \setminus S_0) + o(n\log\sigma)$ bits for any $k = o(\log_\sigma n)$, where n_i is the number of symbols in S_i. Since $H_k(S) \le H_0(S)$ for $k \ge 0$, all subsequences S_i are stored in $nH_0(S) + o(n\log\sigma)$ bits.

Updates. When a new symbol is inserted, we insert it into the subsequence S_0 and update the sequence R. The data structure for $\tilde{S}$ is also updated accordingly. We also insert a 1-bit at the appropriate position of bit sequences M and D_a where a is the inserted symbol. Deletions from S_0 are symmetric. When an element is deleted from S_i, $i \ge 1$, we replace the 1-bit corresponding to this element in M with a 0-bit. We also change the appropriate bit in D_a to 0, where a is the symbol that was deleted from S_i.

We must guarantee that the number of elements in S_0 is bounded by $O(n/r)$; the number of elements marked as deleted must be also bounded by $O(n/r)$. Hence we must re-build the data structure when the number of symbols in S_0 or the number of deleted symbols is too big. Since we aim for updates with worst-case bounds, the cost of re-building is distributed among $O(n/r)$ updates. We run two processes in the background. The first background process moves elements of S_0 into subsequences S_i. The second process purges sequences S_1, ..., S_r and removes all symbols marked as deleted from these sequences. Details are given in the full version of this paper.

We assumed in the description of updates that $\log n$ is fixed. In the general case we need additional background processes that increase or decrease sizes of subsequences when n becomes too large or too small. These processes are organized in a standard way. Thus we obtain the following result

Lemma 3. *A dynamic string $S[1,n]$ over alphabet $\Sigma = \{1,\ldots,\sigma\}$ for $\sigma < n/\log^3 n$ can be stored in a data structure using $nH_0 + o(n\log\sigma) + O(n\log\log n)$ bits, and supporting queries* access, rank *and* select *in time $O(\log n/\log\log n)$. Insertions and deletions of symbols are supported in $O(\log n/\log\log n)$ time.*

In the full version of this paper [23] we show that the space usage of the above described data structure can be reduced to $nH_k + o(n\log\sigma)$ bits. We also show how the result of Lemma 3 can be extended to the case when $\sigma \geq n/\log^3 n$. The full version also contains the description of the static data structure and presents the procedure for extracting a substring $S[i..i+\ell]$ of S in $O(\log n/\log\log n + \ell)$ time.

4.1 Compressed Data Structure for $\sigma > n/\log^3 n$

If the alphabet size σ is almost linear, we cannot afford storing the arrays $Count_a[]$. Instead, we keep a bit sequence $BCount_a$ for each alphabet symbol a. Let $s_{a,i}$ denote the number of a's occurrences in the subsequence S_i and $s_a = \sum_{i=1}^{r} s_{a,i}$. Then $BCount_a = 1^{s_{a,1}}01^{s_{a,2}}0\ldots1^{s_{a,r}}$. If $s_a < r\log^2 n$,we can keep $BCount_a$ in $O(s_a \log\frac{r+s_a}{s_a}) = O(s_a\log\log n)$ bits. If $s_a > r\log^2 n$, we can keep $BCount_a$ in $O(r\log\frac{r+s_a}{s_a}) = O((s_a/\log^2 n)\log n) = O(s_a/\log n)$ bits. Using $BCount_a$, we can find for any q the subsequence S_j, such that $Count_a[j] < q \leq Count_a[j+1]$ in $O(\log n/\log\log n)$ time.

We also keep an effective alphabet[4] for each S_j. We keep a bit vector $Map_j[]$ of size σ, such that $Map_j[a] = 1$ if and only if a occurs in S_j. Using $Map_j[]$, we can map a symbol $a \in [1,n]$ to a symbol $map_j(a) = \text{rank}_1(a, Map_j)$ so that $map_j(a) \in [1, |S_j|]$ for any a that occurs in S_j. Let $\Sigma_j = \{ map_j(a) \mid a \text{ occurs in } S_j \}$. For every $map_j(a)$ we can find the corresponding symbol a using a select query on Map_j. We keep a static data structure for each sequence S_j over Σ_j. Queries and updates are supported in the same way as in Lemma 3. Combining the result of this subsection and Lemma 3, we obtain the data structure for an arbitrary alphabet size.

Theorem 1. *A dynamic string $S[1,n]$ over alphabet $\Sigma = \{1,\ldots,\sigma\}$ can be stored in a data structure using $nH_0 + o(n\log\sigma)$ bits, and supporting queries*

[4] An alphabet for S_j is effective if it contains only symbols that actually occurred in S_j.

access, rank *and* select *in time* $O(\log n/\log\log n)$. *Insertions and deletions of symbols are supported in* $O(\log n/\log\log n)$ *time.*

5 Compressed Data Structure II

By slightly modifying the data structure of Theorem 1 we can reduce the space usage to essentially $H_k(S)$ bit per symbol for any $k = o(\log_\sigma n)$ simultaneously. First, we observe that any sub-sequence S_i for $i \geq 1$ is kept in a data structures that consumes $H_k(S_i) + o(|S_i| \log \sigma)$ bits of space. Thus all S_i use $\sum_{i=1}^{r}(n_i H_k(S_i) + o(n_i \log \sigma)) = nH_k(S \backslash S_0) + o(n \log \sigma)$ bits. It can be shown that $nH_k(S \backslash S_0) + o(n_i \log \sigma)) = nH_k(S \backslash S_0) + O(n\frac{\log n}{r}) + o(n \log \sigma)$ bits; for completeness, we prove this bound in the full version [23]. Since $r = O(\log n/\log\log n)$, the data structure of Theorem 1 uses $nH_k + o(n \log \sigma) + O(n \log\log n)$ bits.

In order to get rid of the $O(n \log\log n)$ additive term, we use a different static data structure; our static data structure is described in the full version. As before, the data structure for a sequence S_i uses $|S_i|H_k + o(|S_i| \log \sigma)$ bits. But we also show in the full version that our static data structure can be constructed in $O(|S_i|/\log^{1/6} n)$ time if the alphabet size σ is sufficiently small, $\sigma \leq 2^{\log^{1/3} n}$. The space usage $nH_k(S) + o(n \log \sigma)$ can be achieved by appropriate change of the parameter r. If $\sigma > 2^{\log^{1/3} n}$, we use the data structure of Theorem 1. As explained above, the space usage is $nH_k + o(n \log \sigma) + O(n \log\log n) = nH_k + o(n \log \sigma)$. If $\sigma \leq 2^{\log^{1/3} n}$ we also use the data structure of Theorem 1, but we set $r = O(\log n \log\log n)$. The data structure needs $nH_k(S) + O(n/\log\log n) + o(n \log \sigma) = nH_k(S) + o(n \log \sigma)$ bits. Since we can re-build a static data structure for a sequence S_i in $O(|S_i| \log^{1/6} n)$ time, background processes incur an additional cost of $O(\log n/\log\log n)$. Hence the cost of updates does not increase.

6 Substring Extraction

Our representation of compressed sequences also enables us to retrieve a substring $S[i..i+\ell-1]$ of S. We can retrieve a substring of S by extracting a substring of S_0 and a substring of some S_i for $i \geq 1$ and merging the result. A detailed description is provided in the full version of this paper [23]. Our result can be summed up as follows.

Theorem 2. *A dynamic string* $S[1, n]$ *over alphabet* $\Sigma = \{1, \ldots, \sigma\}$ *can be stored in a data structure using* $nH_k + o(n \log \sigma)$ *bits, and supporting queries* access, rank *and* select *in time* $O(\log n/\log\log n)$. *Insertions and deletions of symbols are supported in* $O(\log n/\log\log n)$ *time. A substring of* S *can be extracted in* $O(\log n/\log\log n + \ell/\log_\sigma n)$ *time, where* ℓ *denotes the length of the substring.*

References

1. Arge, L., Vitter, J.S.: Optimal external memory interval management. SIAM J. Comput. 32(6), 1488–1508 (2003)
2. Barbay, J., Gagie, T., Navarro, G., Nekrich, Y.: Alphabet partitioning for compressed rank/select and applications. In: Cheong, O., Chwa, K.-Y., Park, K. (eds.) ISAAC 2010, Part II. LNCS, vol. 6507, pp. 315–326. Springer, Heidelberg (2010)
3. Barbay, J., He, M., Munro, J.I., Rao, S.S.: Succinct indexes for strings, binary relations and multi-labeled trees. ACM Transactions on Algorithms 7(4), article 52 (2011)
4. Belazzougui, D., Navarro, G.: New lower and upper bounds for representing sequences. In: Epstein, L., Ferragina, P. (eds.) ESA 2012. LNCS, vol. 7501, pp. 181–192. Springer, Heidelberg (2012)
5. Blandford, D., Blelloch, G.: Compact representations of ordered sets. In: Proc. 15th SODA, pp. 11–19 (2004)
6. Chan, H., Hon, W.-K., Lam, T.-H., Sadakane, K.: Compressed indexes for dynamic text collections. ACM Transactions on Algorithms 3(2), article 21 (2007)
7. Chan, H.-L., Hon, W.-K., Lam, T.-W.: Compressed index for a dynamic collection of texts. In: Sahinalp, S.C., Muthukrishnan, S.M., Dogrusoz, U. (eds.) CPM 2004. LNCS, vol. 3109, pp. 445–456. Springer, Heidelberg (2004)
8. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Transactions on Algorithms 3(2), article 20 (2007)
9. Fredman, M., Saks, M.: The cell probe complexity of dynamic data structures. In: Proc. 21st STOC, pp. 345–354 (1989)
10. Golynski, A., Munro, J.I., Rao, S.S.: Rank/select operations on large alphabets: a tool for text indexing. In: Proc. 17th SODA, pp. 368–373 (2006)
11. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proc. 14th SODA, pp. 841–850 (2003)
12. Grossi, R., Raman, R., Satti, S.R., Venturini, R.: Dynamic compressed strings with random access. In: Fomin, F.V., Freivalds, R., Kwiatkowska, M., Peleg, D. (eds.) ICALP 2013, Part I. LNCS, vol. 7965, pp. 504–515. Springer, Heidelberg (2013)
13. Gupta, A., Hon, W.-K., Shah, R., Vitter, J.S.: A framework for dynamizing succinct data structures. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 521–532. Springer, Heidelberg (2007)
14. He, M., Munro, J.I.: Succinct representations of dynamic strings. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 334–346. Springer, Heidelberg (2010)
15. Hon, W.-K., Sadakane, K., Sung, W.-K.: Succinct data structures for searchable partial sums. In: Ibaraki, T., Katoh, N., Ono, H. (eds.) ISAAC 2003. LNCS, vol. 2906, pp. 505–516. Springer, Heidelberg (2003)
16. Hon, W.-K., Sadakane, K., Sung, W.-K.: Succinct data structures for searchable partial sums with optimal worst-case performance. Theoretical Computer Science 412(39), 5176–5186 (2011)
17. Jansson, J., Sadakane, K., Sung, W.-K.: CRAM: Compressed random access memory. In: Czumaj, A., Mehlhorn, K., Pitts, A., Wattenhofer, R. (eds.) ICALP 2012, Part I. LNCS, vol. 7391, pp. 510–521. Springer, Heidelberg (2012)
18. Lee, S., Park, K.: Dynamic rank-select structures with applications to run-length encoded texts. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 95–106. Springer, Heidelberg (2007)

19. Lee, S., Park, K.: Dynamic rank/select structures with applications to run-length encoded texts. Theoretical Computer Science 410(43), 4402–4413 (2009)
20. Mäkinen, V., Navarro, G.: Dynamic entropy-compressed sequences and full-text indexes. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 307–318. Springer, Heidelberg (2006)
21. Mäkinen, V., Navarro, G.: Dynamic entropy-compressed sequences and full-text indexes. ACM Transactions on Algorithms 4(3), article 32 (2008)
22. Manzini, G.: An analysis of the burrows-wheeler transform. J. ACM 48(3), 407–430 (2001)
23. Munro, J.I., Nekrich, Y.: Compressed data structures for dynamic sequences. ArXiv e-prints 1507.06866 (2015)
24. Navarro, G., Nekrich, Y.: Optimal dynamic sequence representations. In: Proc. 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2013), pp. 865–876 (2013)
25. Navarro, G., Nekrich, Y.: Optimal dynamic sequence representations (full version). submitted for publication (2013)
26. Navarro, G., Sadakane, K.: Fully functional static and dynamic succinct trees. ACM Transactions on Algorithms 10(3), 16 (2014)
27. Patrascu, M., Demaine, E.D.: Tight bounds for the partial-sums problem. In: Proc. 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2004), pp. 20–29 (2004)

A.1 Prefix Sum Queries on a List

In this section we describe a data structure on a list L that is used in the proof of Lemma 1 in Section 2.

Lemma 4. *We can keep a dynamic list L in an $O(m \log m)$-bit data structure $D(L)$, where m is the number of entries in L. $D(L)$ can find the i-th entry in L for $1 \leq i \leq m$ in $O(\log m / \log\log n)$ time. $D(L)$ can also compute the number of entries before a given element $e \in L$ in $O(\log m / \log\log n)$ time. Insertions and deletions are also supported in $O(\log m / \log\log n)$ time.*

Proof: $D(L)$ is implemented as a balanced tree with node degree $\Theta(\log^{\varepsilon} n)$. In every internal node we keep a data structure *Pref*(u); *Pref*(u) contains the total number $n(u_i)$ of elements stored below every child u_i of u. *Pref*(u) supports prefix sum queries (i.e., computes $\sum_{i=1}^{t} n(u_i)$ for any t) and finds the largest j, such that $\sum_{i=1}^{j} n(u_i) \leq q$ for any integer q. We implement *Pref*(u) as in Lemma 2.2 in [27] so that both types of queries are supported in $O(1)$ time. *Pref*(u) uses linear space (in the number of its elements) and can be updated in $O(1)$ time. *Pref*(u) needs a look-up table of size $o(n^{\varepsilon})$. To find the i-th entry in a list, we traverse the root-to-leaf path; in each visited node u we find the child that contains the i-th entry using *Pref*(u). To find the number of entries preceding a given entry e in a list, we traverse the leaf-to-root path π that starts in the leaf containing e. In each visited node u we answer a query to *Pref*(u): if the j-th child u_j of u is on π, then we compute $s(u) = \sum_{i=1}^{j-1} n(u_i)$ using *Pref*(u). The total number of entries to the left of e is the sum of $s(u)$ for all nodes u on π. Since we spend $O(1)$ time in each visited node, both types of queries are answered in $O(1)$ time. An update operation leads to $O(\log m / \log\log n)$ updates of data structures *Pref*(u). The tree can be re-balanced using the weight-balanced B-tree [1], so that its height is always bounded by $O(\log m / \log\log n)$. $\square$