Rank/Select on Dynamic Compressed Sequences and Applications [⋆]

Rodrigo González Gonzalo Navarro [∗]

Deptartment of Computer Science, University of Chile.
Av. Blanco Encalada 2120, 3rd Floor, Santiago, Chile.

Abstract

Operations *rank* and *select* over a sequence of symbols have many applications to the design of succinct and compressed data structures managing text collections, structured text, binary relations, trees, graphs, and so on. We are interested in the case where the collections can be updated via insertions and deletions of symbols. Two current solutions stand out as the best in the tradeoff of space versus time (when considering all the operations). One solution, by Mäkinen and Navarro, achieves compressed space (i.e., $nH_0 + o(n \log \sigma)$ bits) and $O(\log n \log \sigma)$ worst-case time for all the operations, where n is the sequence length, σ is the alphabet size, and H_0 is the zero-order entropy of the sequence. The other solution, by Lee and Park, achieves $O(\log n(1 + \frac{\log \sigma}{\log \log n}))$ amortized time and uncompressed space, i.e. $n \log_2 \sigma + O(n) + o(n \log \sigma)$ bits. In this paper we show that the best of both worlds can be achieved. We combine the solutions to obtain $nH_0 + o(n \log \sigma)$ bits of space and $O(\log n(1 + \frac{\log \sigma}{\log \log n}))$ worst-case time for all the operations. Apart from the best current solution to the problem, we obtain several byproducts of independent interest applicable to partial sums, text indexes, suffix arrays, the Burrows-Wheeler transform, and others.

Key words: Compressed data structures, empirical entropy, text indexing.

[⋆] Supported in part by Millennium Institute for Cell Dynamics and Biotechnology, Grant ICM P05-001-F, Mideplan, Chile; and Fondecyt Grant 1-080019, Chile. An early version of this paper appeared in *Proc. LATIN'08*.

[∗] Corresponding author.
Email addresses: `rgonzale@dcc.uchile.cl` (Rodrigo González), `gnavarro@dcc.uchile.cl` (Gonzalo Navarro).

1 Introduction and Related Work

Compressed data structures aims at representing classical data structures such as sequences, trees, graphs, etc., in little space while keeping the functionality of the structure. That is, compressed data structures should operate without the need to decompress them. This is a very active area of research stimulated by today's steep memory hierarchies and large available data sizes. See for example a recent survey [26].

One of the most useful structures are the bit vectors with *rank* and *select* operations: $rank(B, i)$ gives the number of 1-bits in $B[1, i]$ and $select(B, i)$ gives the position of the i-th 1 in B. Both can be solved in constant time using $o(n)$ bits on top of $B[1, n]$ [25]. Moreover, B can be represented in compressed form, so that the constant times are retained and the space becomes $nH_0(B) + o(n)$ [29]. Here $H_0(B)$ is the zero-order entropy of B, $H_0(B) = \frac{n_0}{n} \log \frac{n}{n_0} + \frac{n_1}{n} \log \frac{n}{n_1}$, where B has n_0 0s and n_1 1s (logarithms are in base 2 by default in this paper). From this compressed representation one can easily retrieve $B[i] = rank(B, i) - rank(B, i-1)$ in constant time, so the compressed representation *replaces* B and in addition gives rank/select functionality on it.

Rank and select queries generalize to sequences $T[1, n]$ over an alphabet Σ of size σ, where one aims at a (hopefully compressed) representation efficiently supporting the following operations:

- $access(T, i)$ returns the symbol $T[i]$.
- $rank_c(T, i)$ returns the number of times symbol c appears in the prefix $T[1, i]$.
- $select_c(T, i)$ returns the position of the i-th c in T.

Improvements in *rank/select* operations on sequences have a great impact on many other succinct data structures, especially on those aimed at text indexing [26,12,8,21], but also labeled trees, structured texts, binary relations, graphs, and others [1,2,15].

The first structure providing support for *rank/select* on a sequence of symbols was the *wavelet tree* [14,10]. Wavelet trees are perfectly balanced static trees of height $\log \sigma$. They answer the three queries in $O(\log \sigma)$ time, by working $O(1)$ per tree level. They store a bitmap of length n per level, which is preprocessed for constant-time binary *rank/select* queries. Their total space requirement is $n \log \sigma + o(n \log \sigma)$, where the extra sublinear term is the space needed by the binary *rank/select* structures. By representing those bitmaps in compressed form [29] the $O(\log \sigma)$ *rank/select* times are retained and the space becomes $nH_0(T) + o(n \log \sigma)$, where $H_0(T)$ is the zero-order empirical entropy of T (that is, $\sum_{c \in \Sigma} \frac{n_c}{n} \log \frac{n}{n_c}$, where c occurs n_c times in T). Since the wavelet tree gives $access(T, i)$ to any symbol $T[i]$, it can be used to *replace* T.

A stronger version of wavelet trees are *multiary wavelet trees* [8], which achieve the same space but improve the query times to $O(1 + \frac{\log \sigma}{\log \log n})$. The trick is to make the tree ρ-ary for some $\rho = O(\log^\alpha n)$ and constant $0 < \alpha < 1$, so that its height is reduced. Now the tree does not store a bitmap per level, but rather a sequence over an alphabet of size ρ. They show how to do *rank*/*select* on those sequences in constant time for such a small ρ.

Chan et al. [6] considered dynamic capabilities for the sequences, by including insert/delete operations. The result is the problem we address in this paper, which we now define formally

Definition 1. The *Dynamic Sequence with Indels* problem consists in maintaining a sequence $T = t_1 t_2 \dots t_n$ of symbols over an alphabet Σ of size σ, supporting the queries $access(T, i)$, $rank_c(T, i)$, and $select_c(T, i)$ as defined above, as well as the operations:

- $insert_c(T, i)$ inserts symbol c between $T[i-1]$ and $T[i]$.
- $delete(T, i)$ deletes $T[i]$ from T.

Chan et al. presented a structure for binary sequences taking $O(n)$ bits of space and performing all the operations in $O(\log n)$ time. Blanford and Blelloch [3] improved the space to $O(nH_0)$, and finally Mäkinen and Navarro [20,21] achieved $nH_0(B)+o(n)$ bits of space, still solving all the operations in $O(\log n)$ time. This is achieved with a binary tree that stores $\Theta(\log^2 n)$ bits at the leaves, and at internal nodes stores summary *rank*/*select* information on the subtrees.

The solution is easily extended to handle sequences. A wavelet tree using dynamic bitmaps yields a dynamic sequence representation that takes $nH_0(T) + o(n \log \sigma)$ bits and solves all the operations in time $O(\log n \log \sigma)$ [21].

Recently, Lee and Park [19] manage to improve the time complexities of this solution. They show that the $O(\log n)$ time complexities can be achieved for alphabets of size up to $\sigma = O(\log n)$, yet only in an amortized way. They combine this tool with a multiary wavelet tree to achieve $O(\log n(1 + \frac{\log \sigma}{\log \log n}))$ time.

The key to the success of Lee and Park is a clever detachment of two roles of tree leaves that are entangled in Mäkinen and Navarro's solution: In the latter, the leaves are the memory allocation unit (that is, whole leaves are allocated or freed), and also the information summarization unit (that is, the tree maintains information up to leaf granularity, and the rest has to be collected by sequentially scanning a leaf). In Lee and Park's solution leaves are the information summarization unit, but handle an internal linked list with smaller memory allocation units. This permits moving symbols to accommodate the space upon insertions/deletions within a leaf, without having to update summarization information for the data moved. This was the main bottleneck that prevented the use of larger alphabets in $O(\log n)$ time in Mäkinen and Navarro's method.

Despite these improvements compared to Mäkinen and Navarro's, the work by Lee and Park has several weaknesses: (1) it is not compressed, but rather takes $n \log \sigma + O(n) + o(n \log \sigma)$ bits of space; (2) in addition to not compressing T, the extra space includes an $O(n)$ term, as shown; (3) times are amortized, not worst-case.

In this paper we show that it is possible to obtain the best from both worlds. We combine the works [21,19] to obtain a structure that (1) takes $nH_0(T) + o(n \log \sigma)$ bits of space, and (2) performs all the operations in $O(\log n(1 + \frac{\log \sigma}{\log \log n}))$ worst-case time. (This is achieved even for the case where $\lceil \log n \rceil$ changes and so does the length of the structure pointers in order to maintain the promised space bounds.) The result becomes *the most efficient dynamic representation of sequences*, both in time and space, and its benefits have immediate applications to other succinct data structures such as compressed text indexes, as we show at the end.

The combination of both methods is by no means simple. Some parts are not hard to merge, such as the role detachment for leaves [19] with the compressed representation of sequences [8] and multi-ary wavelet trees, plus the memory management techniques to support changes of $\lceil \log n \rceil$ within the same worst-case time bounds and no extra space [21]. However, others require new algorithmic ideas. Lee and Park spend $O(n)$ extra bits in bitmaps that maintain leaf-granularity information on *rank*/*select*. We show that this can be replaced by dynamic partial sums, which use sublinear space. However, we need σ partial sums and cannot afford to update them individually upon a leaf insertion/deletion. Hence we create a new structure where a collection of σ sequences are maintained in synchronization, and this can be of independent interest. The second problem was that leaf splitting/merging in Lee and Park's work triggered too many updates to summarization data, which could not be handled in $O(\log n)$ worst-case time, only in $O(\log n)$ amortized time. To get rid of this problem we redefined the leaf fill ratio invariants, preferring a weaker condition that still ensures that leaves are sufficiently full and can be maintained within the $O(\log n)$-worst-case-time bound. This can also be of independent interest.

Our result is not only interesting by itself, but also derives into the best current algorithm to maintain a dynamic collection of texts that can be searched for patterns, and to build indexes for static text collections within compressed space. In addition, our results permit building suffix arrays [23] in competitive time, improving in particular the best algorithm to build it within $O(n \log \sigma)$ bits of space when the alphabet is not too large. Finally, we derive the best current algorithm to compute the Burrows-Wheeler Transform [4] within $n \log \sigma + O(n)$ bits of space.

We remind that there is a static sequence representation [12] that requires $n \log \sigma + n\, o(\log \sigma)$ bits and answers the queries in $O(\log \log \sigma)$ time. There has been work on dynamizing this structure [15], where they achieve the same space plus $o(n)$ bits, the query times are increased by $O(\frac{1}{\epsilon} \log \log n)$, and the update times are $O(\frac{1}{\epsilon} n^{\epsilon})$ amortized, for any constant $0 < \epsilon < 1$. In fact the method can be used to dynamize any other scheme (such as the wavelet-tree-based ones [8]), at the same extra cost. This is extremely relevant when query times are more important than update times. In this paper we focus on achieving the best time *for all the operations*. In particular, this is crucial when using the scheme to achieve good construction times within compressed space.

The paper proceeds as follows. In Section 2 we describe a solution to handle a collection of several synchronized partial sums. This is used in Section 3 to design a dynamic *rank/select* solution for small alphabets ($O(\log n)$) with no compression. In Section 4 we introduce compression, first for even smaller alphabets ($o(\log n / \log \log n)$), and then generalizing for arbitrary alphabets via multi-ary wavelet trees. We explore some consequences and future work directions in the Discussion section.

As for the model of computation, our results (and all the mentioned ones) assume a RAM model with word size $w = \Omega(\log n)$, so that operations on $O(\log n)$ contiguous bits can be carried out in constant time. For the dynamic structures, we always allocate $\omega(\log n)$-bit chunks of the same size (or a finite set of sizes), which can be handled in constant time and asymptotically no extra space [30].

2 Collection of Searchable Partial Sums with Indels

In this section we generalize the well-known partial sums problem to handle a collection of somehow "synchronized" sequences. Apart from having independent interest, this will be an essential tool for the main development in the paper.

Definition 2. The *Searchable Partial Sums with Indels (SPSI)* problem [17] consists in maintaining a sequence S of nonnegative integers $s_1, \ldots, s_n$, each one of $k = O(\log n)$ bits, supporting the following queries and operations:

- $sum(S, i)$ is $\sum_{\ell=1}^{i} s_\ell$.
- $search(S, y)$ is the smallest i' such that $sum(S, i') \geq y$.

- *update*(S, i, x) updates s_i to $s_i + x$ (x can be negative as long as the result is not).
- *insert*(S, i, x) inserts a new integer x between s_{i-1} and s_i.
- *delete*(S, i) deletes s_i from the sequence.

It is possible to solve the SPSI problem using $kn + o(kn)$ bits of space and $O(\log n)$ time per operation [21]. We now define our extension of this problem.

Definition 3. The *Collection of Searchable Partial Sums with Indels (CSPSI)* problem consists in maintaining a collection of σ sequences $C = \{S^1, \ldots, S^\sigma\}$ of nonnegative integers $\{s_i^j\}_{1 \le j \le \sigma, 1 \le i \le n}$, each one of $k = O(\log n)$ bits. The following operations must be supported:

- *sum*(C, j, i) is $\sum_{\ell=1}^{i} s_\ell^j$;
- *search*(C, j, y) is the smallest i' such that *sum*$(C, j, i') \ge y$;
- *update*(C, j, i, x) updates s_i^j to $s_i^j + x$;
- *insert*(C, i) inserts 0 between s_{i-1}^j and s_i^j for all $1 \le j \le \sigma$.;
- *delete*(C, i) deletes s_i^j from the sequence S^j for all $1 \le j \le \sigma$; To perform *delete*(C, i) it must hold $s_i^j = 0$ for all $1 \le j \le \sigma$.

Note the limitations about inserting/deleting only zeros, and at the same place in all sequences. In the sequel we show how to solve the CSPSI problem in $O(\sigma + \log n)$ time, using $O(\sigma k n)$ bits of space.

Data structure. We construct a red-black tree over C, where each leaf contains a non-empty *superblock*, whose size goes from $\frac{1}{2}\log^2 n$ to $2\log^2 n$ bits[1]. The leftmost leaf contains $s_1^1 \cdots s_{b_1}^1 s_1^2 \cdots s_{b_1}^2 \cdots\ s_1^\sigma \cdots s_{b_1}^\sigma$, the second leftmost leaf contains $s_{b_1+1}^1 \cdots s_{b_2}^1 s_{b_1+1}^2 \cdots s_{b_2}^2 \cdots s_{b_1+1}^\sigma \cdots s_{b_2}^\sigma$, and so on. The size of the leftmost leaf is $\sigma k b_1$ bits, the size of the second leftmost leaf is $\sigma k(b_2 - b_1)$ bits, and so on. The size of the leaves is variable and bounded, so $b_1, b_2, \ldots$ are such that $\frac{1}{2}\log^2 n \le \sigma k b_1, \sigma k(b_2 - b_1), \ldots \le 2\log^2 n$.[2] Each internal node v stores counters $\{r^j(v)\}_{1 \le j \le \sigma}$ and $p(v)$, where $r^j(v)$ is the sum of the integers in the left subtree for sequence S^j and $p(v)$ is the number of positions stored in the left subtree (for any sequence).

Each superblock is further divided into *blocks* of $\sqrt{\log n}\log n$ bits, so each superblock has between $\frac{1}{2}\sqrt{\log n}$ and $2\sqrt{\log n}$ blocks. We maintain these blocks using a linked list. Only the last block in the list could have some free space, all the other use all of their bits. To scan a leaf we proceed block by block. To directly access an arbitrary element in a leaf we must also follow the links of the blocks until we arrive at the correct block. This takes $O(\sqrt{\log n})$ steps.

[1] In most cases we ignore floors and ceilings for simplicity.
[2] If $\sigma k > 2\log^2 n$, we just store σk bits per leaf. All the algorithms in the sequel get simplified and the complexities are maintained.

Computing $sum(C, j, i)$**.** We traverse the tree to find the leaf containing the i-th position. We start with $sum \leftarrow 0$ and $v \leftarrow root$. If $p(v) \geq i$ we enter the left subtree, otherwise we enter the right subtree with $i \leftarrow i - p(v)$ and $sum \leftarrow sum + r^j(v)$. We reach the leaf that contains the i-th position in $O(\log n)$ time. Then we scan the leaf, summing up from where the sequence S^j begins, in chunks of size $\frac{1}{2}\log n$ bits using a universal precomputed table Y, until we reach position i. Table Y receives any possible sequence of dk bits, for $d = \lfloor \frac{\frac{1}{2}\log n}{k} \rfloor$, and gives the sum of the d k-bit numbers encoded. The last (at most $d-1$) integers must be added individually.[3] The sum query takes in total $O(\log n)$ time, and table Y adds only $O(\sqrt{n}\ \mathrm{polylog}(n))$ bits of space.

Block boundaries do not affect the procedure. If the sequence of dk bits we must input to Y is split between the current and next block, we read the corresponding bits from both blocks to compose the sequence before applying Y. Thus the complexities are not affected.

Computing $search(C, j, y)$**.** We enter the tree to find the smallest i' such that $sum(C, j, i') \geq y$. We start with $pos \leftarrow 0$ and $v \leftarrow root$. If $r^j(v) \geq y$ we enter the left subtree, otherwise we enter the right subtree with $y \leftarrow y - r^j(v)$ and $pos \leftarrow pos + p(v)$. We reach the leaf that contains the i'-th position in $O(\log n)$ time. Then we scan the leaf, summing up from where the sequence S^j begins, in chunks of size $\frac{1}{2}\log n$ bits using table Y, until this sum is greater than y after adding up i' integers; the answer is then $pos + i'$. (More precisely, once an application of the table exceeds y, we must reprocess the last chunk number-wise.) The $search$ query takes in total $O(\log n)$ time.

Operation $update(C, j, i, x)$**.** We proceed similarly to sum, updating $r^j(v)$ as we traverse the tree. That is, we update $r^j(v)$ to $r^j(v) + x$ each time we go left from v. When we reach the leaf we directly update s_i^j to $s_i^j + x$ in $O(\sqrt{\log n})$ time (direct access). The $update$ operation takes in total $O(\log n)$ time.

For the next operations, we note that a leaf has at most $m = \lfloor \frac{2\log^2 n}{\sigma k} \rfloor$ integers from any sequence. Then a subsequence of a given sequence has at most mk bits. So if we copy a subsequence in chunks of $\frac{1}{2}\log n$ bits, the process will take $1 + \lceil \frac{2mk}{\log n} \rceil = O(1 + \frac{\log n}{\sigma})$ time in the RAM model[4]. As we have σ sequences, we can copy a given subsequence of them all in $O(\sigma + \log n)$ time.

[3] Note that if $k > \frac{1}{2}\log n$ we can just add each number individually within the time bounds.

[4] This requires shifting bits, which in case it is not supported by the model, can be handled using small universal tables of the kind of Y.

The next operations are solved by a constant number applications of these copying operations. Again, block boundaries do not affect the complexities.

Operation *insert*(C, i)**.** We traverse the tree similarly to *sum*, updating $p(v)$ as we traverse the tree. That is, we increase $p(v)$ by 1 each time we go left from v. Then we create a new copy of the leaf arrived at (by allocating new blocks as needed), adding a 0 between s^j_{i-1} and s^j_i for all j. This is done by first copying the subsequences $\ldots s^j_{i-1}$ for all j, then adding 0 to each sequence, and finally copying the subsequences $s^j_i \ldots$ for all j. As we have just explained, this can be done in $O(\sigma + \log n)$ time.

If the new leaf uses more than $2\log^2 n$ bits, it is split into two. An overflowed leaf has $m = \lfloor \frac{2\log^2 n}{\sigma k} \rfloor + 1$ integers in each sequence. So we store in the left leaf the first $\lfloor m/2 \rfloor$ integers of each sequence and in the right leaf we store the rest. These two copies can be done again in $O(\sigma + \log n)$ time. The new leaves are made children of a new node μ. We compute each $r^j(\mu)$ by scanning and summing on the left leaf. This summing can be done in $O(\sigma + \log n)$ time using table Y. We also set $p(\mu) = \lfloor m/2 \rfloor$. Finally, we check if we need to rebalance the tree. If needed, the red-black tree is rebalanced with $O(1)$ rotations and $O(\log n)$ red-black tag updates [7, Chapter 13.3]. After a rotation we need to update $r^j(\cdot)$ and $p(\cdot)$ only for one tree node, which is easily done in $O(\sigma)$ time. The *insert* operation takes in total $O(\sigma + \log n)$ time.

Operation *delete*(C, i)**.** We traverse the tree similarly to *sum*, updating $p(v)$ while we traverse the tree. That is, we decrease $p(v)$ by 1 each time we go left from v. Then, similarly to *insert*, we make a new copy of the leaf (allocating blocks as needed), deleting s^j_i for all j. This takes $O(\sigma + \log n)$ time.

There are three possibilities after this deletion: (i) The new leaf uses more than $\frac{1}{2}\log^2 n$ bits, in which case we are done. (ii) The new leaf uses less than $\frac{1}{2}\log^2 n$ and its sibling is also a leaf, in which case we merge it with its sibling, again in $O(\sigma + \log n)$ time. Note that this merging removes the leaf's parent but does not require any recomputation of $r^j(\cdot)$ or $p(\cdot)$. (iii) The new leaf uses less than $\frac{1}{2}\log^2 n$ and its sibling is an internal node μ, in which case by the red-black tree properties we have that μ must have two leaf children[5] In this case we merge our new leaf with the closest child of μ, updating the counters of μ in $O(\sigma)$ time, and letting μ replace the parent of our original leaf.

[5] For each node, all paths from the node to descendant leaves contain the same number of black nodes and all the leaves are black. In particular, for the parent of the deleted leaf, if the sibling is an internal node then it is red and its children must be black and leaves.

In cases (*ii*) and (*iii*), the merged leaf might use more than $2\log^2 n$ bits. In this case we split it again into two halves, just as we do in *insert* (and including the recomputation of $r^j(\cdot)$ and $p(\cdot)$). The tree might have to be rebalanced as well. The *delete* operation takes in total $O(\sigma + \log n)$ time.

The breakdown of the space requirement for the structure is as follows.

- All the sequence representations add up to σkn bits of space.
- Each pointer of the linked list of blocks uses $O(\log n)$ bits and we have $O(\frac{\sigma kn}{\sqrt{\log n}\log n})$ full blocks, totalizing $O(\frac{\sigma kn}{\sqrt{\log n}})$ bits.
- The last block in each superblock is not necessarily fully used. We have at most $\lceil \frac{2\sigma kn}{\log^2 n} \rceil$ superblocks, each of which can waste an underused block of size $\sqrt{\log n}\log n$ bits, totalizing $O(\frac{\sigma kn}{\sqrt{\log n}})$ bits.
- For each internal node we have two pointers, red-black data, a counter $p(\cdot)$, and σ counters $r^j(\cdot) \leq 2^k \cdot n$, totalizing $O(\log n) + \sigma(k + \log n) = O(\sigma \log n)$ bits per node. So, the internal nodes use $O(\frac{\sigma kn}{\log^2 n}\ \sigma \log n) = O(\frac{\sigma^2 kn}{\log n})$ bits overall.

We have proved our main result in this section.

Theorem 1 *The Collection of Searchable Partial Sums with Indels problem with σ sequences of n numbers of k bits can be solved, in a RAM machine of $w = \Omega(\log n)$ bits, using $\sigma kn(1 + O(\frac{1}{\sqrt{\log n}} + \frac{\sigma}{\log n}))$ bits of space, supporting all the operations in $O(\sigma + \log n)$ worst-case time. Note that, if $\sigma = O(\log n)$ the space is $O(\sigma kn)$ and the time is $O(\log n)$.*

If we had tried to solve the CSPSI problem by just managing σ SPSI individual problems, the time complexities would have raised to $O(\sigma \log n)$.

We note that we have actually assumed that $w = \Theta(\log n)$ in our space computation (as we have used w-bit system pointers). The general case $w = \Omega(\log n)$ can be addressed using the same technique developed in previous work [21, Sec. 4.5, 4.6, and 6.4], which uses a more refined memory management with pointers of $(\log n) \pm 1$ bits, and splits the sequence into three in a way that retains the worst-case complexities.

The three subsequences are called *previous*, *current* and *next* [21, Sec. 4.5]. Let $l = \lceil \log n \rceil$ be the current pointer width in use, where n is the current length of the sequences. A prefix of all sequences is in *previous* using $l - 1$ bits, and a suffix in *next* using $l + 1$ bits. The middle part is in *current* and uses l bits. Upon insertions and deletions, some elements are moved across the three structures so as to ensure that, when n becomes a new power of 2 (i.e., $\lceil \log n \rceil$ changes), all the elements reside in *previous* (if n becomes $n/2$) or in *next* (if n becomes $2n$) and we can smoothly change l.

To carry out the queries over this split structure we must maintain, for each of the three trees, summary $p(\cdot)$ and $r^j(\cdot)$ data on the whole trees. This allows us to know on which of the trees to operate and also gives us information to translate the local result of one tree into the final answer of the structure.

3 Uncompressed Dynamic Rank-Select Structures for a Small Alphabet

We now turn our attention into the dynamic rank/select problem. We start with a simpler setting, where the alphabet is small, $\sigma = O(\log n)$, and we do not yet attempt to achieve compressed space. In the next section we build on this one to achieve our stronger result.

Data structure. We construct a red-black tree over $T[1, n]$ where each leaf contains a non-empty *superblock* of size up to $2\log^2 n$ bits. Each internal node v stores counters $r(v)$ and $p(v)$, where $r(v)$ is the number of superblocks in the left subtree and $p(v)$ is the number of symbols stored in the left subtree.

A superblock storing less than $\log^2 n$ bits will be called *sparse*. Operations *insert* and *delete* will maintain the invariant that no two consecutive sparse superblocks may exist. This ensures that every consecutive pair of superblocks holds at least $\log^2 n$ bits from T, that is, an average fill ratio of at least 1/2, and thus there are at most $1 + \frac{2n\log\sigma}{\log^2 n}$ superblocks.

For each superblock i, we maintain s_i^j, the number of occurrences of symbol j in superblock i, for $1 \le j \le \sigma$. We store all these sequences of numbers using a *Collection of Searchable Partial Sums with Indels*, C (Section 2). The length of each sequence will be at most $1 + \frac{2n\log\sigma}{\log^2 n}$ integers, we assume $\sigma = O(\log n)$, and $k = O(\log\log n)$ holds because $s_i^j \le \frac{2\log^2 n}{\log\sigma}$. So the partial sums operate in $O(\log n)$ worst-case time (Theorem 1).

Just as in Sec. 2, each superblock is further divided into *blocks* of $\sqrt{\log n}\log n$ bits, so each superblock has up to $2\sqrt{\log n}$ blocks. We maintain these blocks using a linked list. Only the last block could be not fully used, the rest use all of their bits.

The overall space usage of our structure is $n\log\sigma + O(\frac{n\log\sigma}{\sqrt{\log n}})$, as $\sigma = O(\log n)$:

- The text itself uses $n\log\sigma$ bits of space.
- The *CSPSI C* uses $O(\sigma\log\log n\frac{n\log\sigma}{\log^2 n}) = O(\frac{n\log\log n\log\sigma}{\log n})$ bits of space.

- Each pointer of the linked list of blocks uses $O(\log n)$ bits and we have full $O(\frac{n \log \sigma}{\sqrt{\log n \log n}})$ blocks, totalizing $O(\frac{n \log \sigma}{\sqrt{\log n}})$ bits.
- The last block in each superblock is not necessarily fully used. We have at most $1 + \frac{2n \log \sigma}{\log^2 n}$ superblocks, each of which can waste an underused block of size $\sqrt{\log n} \log n$ bits, totalizing $O(\frac{n \log \sigma}{\sqrt{\log n}})$ bits.
- The tree pointers and counters use $O(\frac{n \log \sigma}{\log^2 n} \cdot \log n) = O(\frac{n \log \sigma}{\log n})$ bits.

Now we show how to carry out all the queries/operations in $O(\log n)$ time. First, it is important to notice, as in Section 2, that each block can be scanned or shifted in $O(\sqrt{\log n})$ time, using tables that process chunks of $\frac{1}{2} \log n$ bits[6]. Given that there are $O(\sqrt{\log n})$ blocks in a superblock, we can scan or shift elements within a superblock in $O(\log n)$ time, even considering block boundaries.

Computing $access(T, i)$**.** We traverse the tree to find the leaf containing the i-th position. We start with $sb \leftarrow 1$ and $pos \leftarrow i$. If $p(v) \geq pos$ we enter the left subtree, otherwise we enter the right subtree with $sb \leftarrow sb + r(v)$ and $pos \leftarrow pos - p(v)$. We reach the leaf that contains the i-th position in $O(\log n)$ time. Then we directly access the *pos*-th symbol of superblock *sb*.[7] Note that, within the same $O(\log n)$ time, we can extract any $O(\log^2 n)$-bit long sequence of symbols from T (by moving to next leaves if necessary).

Computing $rank_c(T, i)$**.** We find the leaf containing the i-th position, just as for *access*. Then we scan superblock *sb* from the first block summing up the occurrences of c up to the position *pos*, using a table Z to sum the c's. Z receives a symbol c and $\lfloor \frac{1}{2} \log_\sigma n \rfloor$ symbols ($\leq \frac{1}{2} \log n$ bits), and tells how many times does c appear in the sequence (again, we can just proceed symbolwise if $\log \sigma > \frac{1}{2} \log n$). We add to this quantity $sum(C, c, sb - 1)$, the number of times that c appears before superblock *sb*. The *rank* query takes in total $O(\log n)$ time. Table Z requires $O(\sigma \sqrt{n}\ \text{polylog}(n)) = O(\sqrt{n}\ \text{polylog}(n))$ bits.

Computing $select_c(T, i)$**.** We calculate $j = search(C, c, i)$; this way we know that the i-th c belongs to superblock j and it is the i'-th appearance of c within superblock j, for $i' = i - sum(C, c, j - 1)$. Then we traverse the tree to find the leaf representing superblock j. We start with $sb \leftarrow j$ and $pos \leftarrow 0$. If $r(v) \geq sb$ we enter the left subtree, otherwise we enter the right subtree with

[6] Again, if $\log \sigma > \frac{1}{2} \log n$, we can process each symbol individually within the time bounds. This can happen even if $\sigma = O(\log n)$.

[7] Actually we do not need to know the superblock number *sb* for the *access* query, but we need it for the next ones.

$sb \leftarrow sb - r(v)$ and $pos \leftarrow pos + p(v)$. We reach the correct leaf in $O(\log n)$ time. Then we scan superblock j from the first block, searching for the position of the i'-th appearance of symbol c within superblock j, using table Z. To this position we add pos to obtain the final result. The *select* query takes in total $O(\log n)$ time.

Operation $insert_c(T, i)$**.** We obtain sb and pos just like in the *access* query, except that we start with $pos \leftarrow i - 1$, so as to insert right after position $i - 1$. Then, if superblock sb contains room for one more symbol, we insert c right after the pos-th position of sb, by shifting the symbols through the blocks as explained. If the insertion causes an overflow in the last block of sb, we simply add a new block at the end of the linked list to hold the trailing bits.

We also carry out $update(C, c, sb, 1)$ and retraverse the path from the root to sb adding 1 to $p(v)$ each time we go left from v. In this case we finish in $O(\log n)$ time.

If, instead, the superblock is full, we cannot carry out the insertion yet. We first move one symbol to the previous superblock (creating a new one if this is not possible): We first $delete(T, d)$ the first symbol c' from block sb (the global position of c' is $d = i - pos$), and this cannot cause an underflow of sb. Now, we check how many symbols does superblock $sb - 1$ have (this is easy by subtracting the pos numbers corresponding to accessing blocks $sb - 1$ and sb). If superblock $sb - 1$ can hold one more symbol, we insert the removed symbol c' at the end of superblock $sb - 1$. This is done by calling $insert_{c'}(T, d)$, a recursive invocation that now will arrive at block $sb - 1$ and will not overflow it (thus no further recursion will occur).[8]

If superblock $sb - 1$ is also full or does not exist, then we are entitled to create a sparse superblock between $sb - 1$ and sb, without breaking the invariant on sparse superblocks. We create such an empty superblock and insert symbol c' into it, using the following procedure: We retraverse the path from the root to sb, updating $r(v)$ to $r(v) + 1$ each time we go left from v. When we arrive again at leaf sb we create a new node μ with $r(\mu) = 1$ and $p(\mu) = 1$. Its left child is the new empty superblock, where the single symbol c' is inserted, and its right child is sb. We also execute $insert(C, sb)$ and $update(C, sb, c', 1)$.

After creating μ, we must check if we need to rebalance the tree. If it is needed, it can be done with $O(1)$ rotations and $O(\log n)$ red-black tag updates. After a rotation we need to update $r(\cdot)$ and $p(\cdot)$ only for one tree node. These updates can be done in constant time.

[8] We note that, if one deletes the first symbol of a block and reinserts it at the same position, it will get inserted into the previous block.

Now that we have finally made room to carry out the original insertion, we rerun $insert_c(T, i)$ and it will not overflow again. The whole *insert* operation takes $O(\log n)$ time.

Operation $delete(T, i)$**.** We obtain *sb* and *pos* just as in the *access* query, updating $p(v)$ to $p(v) - 1$ each time we go left from v. Then we delete the *pos*-th position (let c be the symbol deleted) of the *sb*-th superblock, by shifting the symbols back through the blocks. If this deletion empties the last block, we free it. In any case we call $update(C, c, sb, -1)$ on the partial sums.

There are three possibilities after this deletion: (*i*) superblock *sb* is not sparse after the deletion, in which case we are done; (*ii*) *sb* was already sparse before the deletion, in which case we have only to check that it has not become empty; (*iii*) *sb* turned to sparse due to the deletion, in which case we have to care about the invariant on sparse superblocks.

If superblock *sb* becomes empty, we retraverse the path from the root to it, updating $r(v)$ to $r(v) - 1$ each time we go left from v, in $O(\log n)$ time. When we arrive at leaf *sb* again, we remove it and invoke $delete(C, sb)$. Finally, we check if we need to rebalance the tree, in which case $O(1)$ rotations and $O(\log n)$ red-black tag updates suffice, just as for insertion. After a rotation we also need to update $r(\cdot)$ and $p(\cdot)$ only for one tree node. These updates take constant time.

If, instead, superblock *sb* turned to sparse, we make sure that neither superblocks $sb - 1$ or $sb + 1$ are also sparse. If they are not, then superblock *sb* can become sparse and hence we finish without further intervention.

If superblock $sb + 1$ is sparse, we $delete(T, d)$ its first symbol c' (at position d), and $insert_{c'}(T, d)$ at the end of superblock *sb* (as done for the insertion). This recursive call brings no problems because $sb + 1$ is already sparse, and we restore the non-sparse status of *sb*. If superblock $sb + 1$ becomes empty, we remove it just as explained for the case of superblock *sb*. The action is symmetric if $sb + 1$ is not sparse but $sb - 1$ is.[9]

The *delete* operation takes in total $O(\log n)$ time.

Theorem 2 *Given a text T of length n over a small alphabet of size $\sigma = O(\log n)$, the Dynamic Sequence with Indels problem under RAM model with word size $w = \Omega(\log n)$ can be solved using $n \log \sigma + O(\frac{n \log \sigma}{\sqrt{\log n}})$ bits of space,*

[9] For the symmetric case one needs a slightly different version of procedure *insert*, which inserts after, not before, position i.

supporting all the queries access, rank, select, insert and delete, in $O(\log n)$ worst-case time.

We note again that we have actually assumed that $w = \Theta(\log n)$ in our space computation. The general case $w = \Omega(\log n)$ can be obtained using exactly the same techniques developed previously [21, Sec. 4.5, 4.6, and 6.4], with no changes.

4 Compressed Dynamic Rank-Select Structures

We now extend our results to use a compressed sequence representation, by just changing the way we store/manage the *blocks*. The key idea is to detach the *representational* and the *physical* (i.e., compressed) sizes of the storage units at different levels.

We use the same red-black tree over $T[1,n]$, where each leaf contains a non-empty superblock *representing* up to $2\log^2 n$ bits *of the original text* T (they will actually store more or less bits depending on how compressible is the portion of T they represent). The same superblock splitting/merging policy related to sparse superblocks is used. Each internal node has the same counters and they are managed in the same way. So all the queries/operations are exactly the same up to the superblock level. Compression is encapsulated inside the superblocks.

In *physical* terms, a superblock is divided into blocks just as before, and they are still of the same *physical* size, $\sqrt{\log n}\log n$ bits. Depending on compressibility, blocks will represent more or less symbols of the original text, as their physical size is fixed.

In *logical* terms, a superblock is divided into *segments* representing $\lfloor \frac{1}{2}\log_\sigma n \rfloor$ original symbols[10] from T. We represent each segment using the (c,o)-pair encoding of Ferragina et al. [8]: The c part is of fixed width and tells how many occurrences of each alphabet symbol are there in the segment; whereas the o part is of variable width and gives the identifier of the segment among those sharing the same c component. Each c component uses at most $\sigma \log\log n$ bits; while the o components use at most $\frac{1}{2}\log n$ bits each, and overall add up to $nH_0(T) + O(n\log\sigma/\log n)$ bits [8, Sec. 3.1].

In a block of $\sqrt{\log n}\log n$ bits, we store as many bits as they fit. The universal tables (like Y) used to sequentially process the blocks in chunks of $\frac{1}{2}\log n$ bits must now be modified to process the compressed sequence of (c,o) pairs. This is complex because an insertion in a segment introduces a displacement that

[10] Or just one symbol if $\frac{1}{2}\log_\sigma n < 1$.

propagates over all the segments of the superblock, which must be completely recomputed and rewritten (and it can even cause the physical size of the whole superblock to double!). Fortunately all those tedious details have been already sorted out in previous work [21, Sec. 5.2, 6.1, and 6.2], where their "superblocks" play the role of our "blocks", and their tree rearrangements are not necessary for us because we are within a leaf now. Their "partial blocks" mechanism is also not useful for us, because we can tolerate those propagations to extend over all the blocks of our superblocks. Hence only the last block of our superblocks is not completely full.

The time achieved in there [21] is $O(1)$ per $\Theta(\log n)$ physical bits. Even in the worst case (where compression does not work at all in the superblock), the number of physical bits will be $\frac{2\log^2 n}{\frac{1}{2}\log n}(\sigma \log\log n + \frac{1}{2}\log n) = O(\log^2 n + \sigma \log n \log\log n)$, and thus the time to solve any query or carry out any update on a superblock will be $O(\log n + \sigma \log\log n)$.

Let us now consider the space usage of these new structures, focusing only on what differs from the uncompressed version:

- The text itself (as a sequence of pairs (c, o)) uses $nH_0(T) + O(\frac{\sigma n \log\log n}{\log_\sigma n})$ bits.
- The number of full blocks is $O(\frac{nH_0(T) + \frac{\sigma n \log\log n}{\log_\sigma n}}{\sqrt{\log n}\log n})$, and thus the space wasted by their pointers is $O(\frac{n \log\sigma(\sigma \log\log n + \log n)}{\sqrt{\log n}\log n})$ bits.
- The extra space in bits for the tables to operate the (c, o) encoding is $O(\sqrt{n}\ \sigma\ \text{polylog}(n))$.

It can be seen that the time and space complexities depend sharply on σ. Thus the solution is indeed of interest only for rather small $\sigma = o(\log n/\log\log n)$. For such a small alphabet we have the following theorem. Again, all the issues of varying $\lceil \log n \rceil$ and the case $w = \omega(\log n)$ are handled just as in previous work [21, Sec. 4.5, 4.6, and 6.4]

Theorem 3 *Given a text T of length n over a small alphabet of size $\sigma = O(\frac{\sqrt{\log n}}{\log\log n})$ and zero-order entropy $H_0(T)$, the Dynamic Sequence with Indels problem under RAM model with word size $w = \Omega(\log n)$ can be solved using $nH_0(T) + O(\frac{n\log\sigma}{\sqrt{\log n}})$ bits of space, supporting the queries access, rank, select, insert and delete in $O(\log n)$ worst-case time.*

To extend our results to a larger alphabet of size $\sigma = \Omega(\sqrt{\log n}/\log\log n)$, we use a generalized ρ-ary wavelet tree [8] over T, where $\rho = \Theta(\sqrt{\log n}/\log\log n)$. Essentially, this generalized wavelet tree makes a sequence with the first $\log\rho$ bits of the symbols at the first level, the next $\log\rho$ bits at the second level (where the symbols with the same first $\log\rho$ bits are grouped in the same child of the root), and so on. The tree has $O(\log_\rho \sigma) = O(\frac{\log \sigma}{\log\log n})$ levels. We store on each level a sequence over an alphabet of size ρ, which is handled using the solution of Theorem 3, for which ρ is small enough. Hence each query and operation takes $O(\log n)$ time per level, adding up $O(\log n \frac{\log \sigma}{\log\log n})$ worst-case time overall.

As shown by Ferragina et al. [8], the sum of the zero-order-entropy representations of the sequences at each level adds up to the zero-order entropy of T. In addition, the generalized ρ-ary wavelet tree handles changes in $\lceil \log n \rceil$ automatically, as this is encapsulated within each level. We thus obtain our main theorem, where we have included the case of small σ as well. We recall that, within the same time, *access* can retrieve $O(\log_\sigma n \log n)$ consecutive symbols from T.

Theorem 4 *Given a text T of length n over an alphabet of size σ and zero-order entropy $H_0(T)$, the Dynamic Sequence with Indels problem under RAM model with word size $w = \Omega(\log n)$ can be solved using $nH_0(T) + O(\frac{n \log \sigma}{\sqrt{\log n}})$ bits of space, supporting queries access, rank, select, insert and delete in $O(\log n(1 + \frac{\log \sigma}{\log\log n}))$ worst-case time.*

5 Discussion

We have shown that the best two existing solutions to the *Dynamic Sequence with Indels* problem [21,19] can be merged so as to obtain the best from both. This merging is not trivial and involves some byproducts that can be of independent interest. In particular, we have shown how to handle efficiently a synchronized collection of partial sums. We show now a couple of immediate consequences of our improved result.

Very recently [21,22] it has been shown that a wavelet tree built over the Burrows-Wheeler Transform T^{bwt} of a text T [4], and compressed using the (c, o) pair technique, achieves high-order entropy space, namely $nH_h(T) + o(n \log \sigma)$ for any $h+1 \le \alpha \log_\sigma n$ and constant $0 < \alpha < 1$, where $H_h(T)$ is the h-th order empirical entropy of T [24]. This is used by Mäkinen and Navarro [21] to obtain a dynamic text index that handles a collection $\mathcal{C}$ of texts and permits searching for patterns, extracting text snippets, and inserting/deleting texts in/from the collection. Using their definitions [21, Sec. 7] and using their same sampling step, we can state a stronger version of those theorems:

Theorem 5 *The Dynamic Text Collection problem can be solved with a data structure of size $nH_h(\mathcal{C}) + o(n \log \sigma) + O(\sigma^{h+1} \log n + m \log n + w)$ bits, simultaneously for all h. Here n is the length of the concatenation of m texts, $\mathcal{C} = 0\ T_1 0\ T_2 \cdots 0\ T_m$, and we assume that $\sigma = o(n)$ is the alphabet size and*

$w = \Omega(\log n)$ is the machine word size under the RAM model. The structure supports counting of the occurrences of a pattern P in $O(|P| \log n(1+\frac{\log \sigma}{\log\log n}))$ time, and inserting and deleting a text T in $O(|T| \log n(1 + \frac{\log \sigma}{\log\log n}))$ time. After counting, any occurrence can be located in time $O(\log^2 n(1 + \frac{\log\log n}{\log \sigma}))$. Any substring of length ℓ from any T in the collection can be displayed in time $O(\log^2 n(1 + \frac{\log\log n}{\log \sigma}) + \ell \log n(1 + \frac{\log \sigma}{\log\log n}))$. For $h \leq (\alpha \log_\sigma n) - 1$, for any constant $0 < \alpha < 1$, the space complexity simplifies to $nH_h(\mathcal{C}) + o(n \log \sigma) + O(m \log n + w)$ bits.

When the alphabet is of moderate size, that is, $\sigma = O(\text{polylog}(n))$, the times obtained above become $O(|P| \log n)$ for counting, $O(|T| \log n)$ for text insertion/deletion, $O(\log n \log_\sigma n \log\log n)$ for locating, and $O(\log n(\log_\sigma n \log\log n + \ell))$ for displaying.

Another important application that derives from this one is the compressed construction of text indexes. For example, a variant of the FM-index [8] requires h-th entropy space once built, but in order to build it we need $O(n \log n)$ bits of space. The previous theorem can be used to build the FM-index of a text by starting with an empty collection and inserting the text T of interest. Our new results make this process faster.

Theorem 6 *The Alphabet-Friendly FM-index of a text $T[1, n]$ over an alphabet of size σ can be built using $nH_h(T) + o(n \log \sigma)$ bits, simultaneously for all $h \leq (\alpha \log_\sigma n) - 1$ and any constant $0 < \alpha < 1$, in time $O(n \log n(1 + \frac{\log \sigma}{\log\log n}))$.*

We note that this is the same asymptotic space required for the final, static, FM-index [8]. This FM-index is not only relevant by itself, but also as an intermediate step to compute other important structures such as the *suffix array* [23] and the *Burrows-Wheeler Transform (BWT)* [4] of T. Both are easily derived from our dynamic FM-index. Although the final product takes in this case more space than our intermediate representation, we can output the result in order, so that we do not need to maintain the large representation in memory. Our next discussion assumes this model: we must output the suffix array or the BWT sequentially (as otherwise there is no point in building them in little space).

The BWT is the simpler problem for us. We can easily derive it sequentially from the FM-index, by obtaining one by one the symbols in $O(1 + \frac{\log \sigma}{\log\log n})$ time each, and sending them to the output. The best previous result we know of, in terms of space complexity [18], achieves $O(n \log^2 n)$ time ($O(n \log n)$ on average) using $O(n)$ bits in addition to the $n \log \sigma$ bits of the text. This is asymptotically worse than our space and time for any σ. We note that, using previous work [21], one achieves $O(n \log n \log \sigma)$ time, which may be as bad as $O(n \log^2 n)$ for large σ.

Using our result to build the suffix array is a bit more complicated. Let us focus on the case $\sigma = O(\text{polylog}(n))$, where our FM-index worst-case construction time becomes $O(n \log n)$. To obtain the suffix array sequentially we must carry out one *locate* operation for each cell, which can be made as fast as $O(\log n)$ time per cell if we spend $O(n)$ additional bits of space. Thus we can build the suffix array sequentially within $n \log \sigma + o(n \log \sigma) + O(n)$ bits (even on uncompressible texts) and in $O(n \log n)$ time. This was indeed the best known time complexity to build the suffix array until a few years ago [28]. Nowadays linear-time algorithms exist, yet all of them require $O(n \log n)$ bits of space. On the other hand, the best current result on compressed suffix array construction [16] takes $O(n \log \sigma)$ bits of space and $O(n \log n)$ time for arbitrary alphabets (note that the space is not compressed and its constant term is not 1).

Finally, let us discuss how much our results could be improved. Chan et al. [5] recently showed that the *Dynamic Sequence with Indels* problem on bits ($\sigma = 2$) can be solved in $O(\frac{\log n}{\log \log n})$ time for all operations, using $O(n)$ bits of space (this is striking because the rank/select problem was conjectured by several to have the same $\Omega(\log n)$ lower bound of partial sums [27]). Combining with multiary wavelet trees one immediately achieves $O(n \log \sigma)$ bits of space and $O(\frac{\log n \log \sigma}{(\log \log n)^2})$ time for general alphabets. This time matches the lower bound of Fredman and Saks for rank/select [11] as long as $\sigma = O(\text{polylog}(n))$, whereas it is not known whether the result would be time-optimal for larger σ. In any case, this raises the challenge of achieving that complexity within $nH_0 + o(n \log \sigma)$ bits of space.

Alternatively, one would like to improve the space to high-order entropy (not only for the *Dynamic Text Collection* problem, but for the *Dynamic Sequence with Indels* problem). This has not been achieved even if we disregard operations *rank* and *select* and is satisfied only with *access*, *insert*, and *delete*. The dynamic support for the existing nH_h-space solutions to *access* is currently null or very rudimentary [31,13,9].

Finally, one can wish to handle a stronger set of operations. In particular, our wavelet trees are markedly static in shape, and thus supporting changes in the alphabet Σ looks challenging. This would have applications in a dynamic scenario where the set of symbols is not known in advance.

References

[1] J. Barbay, A. Golynski, I. Munro, S. Rao, Adaptive searching in succinctly encoded binary relations and tree-structured documents, in: Proc. 17th CPM, 2006, pp. 24–35.

[2] J. Barbay, M. He, I. Munro, S. Rao, Succinct indexes for strings, binary relations and multi-labeled trees, in: Proc. 18th SODA, 2007, pp. 680–689.

[3] D. Blandford, G. Blelloch, Compact representations of ordered sets, in: Proc. 15th SODA, 2004, pp. 11–19.

[4] M. Burrows, D. Wheeler, A block sorting lossless data compression algorithm, Tech. Rep. 124, Digital Equipment Corporation (1994).

[5] H. Chan, W. Hon, T. Lam, K. Sadakane, Compressed indexes for dynamic text collections, ACM TALG 3 (2) (2007) 21.

[6] H.-L. Chan, W.-K. Hon, T.-W. Lam, Compressed index for a dynamic collection of texts, in: Proc. 15th Annual Symposium on Combinatorial Pattern Matching (CPM), LNCS 3109, 2004, pp. 445–456.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein., Introduction to Algorithms, vol. Second Edition, MIT Press and McGraw-Hill, 2001.

[8] P. Ferragina, G. Manzini, V. Mäkinen, G. Navarro, Compressed representations of sequences and full-text indexes, ACM TALG 3 (2) (2007) article 20.

[9] P. Ferragina, R. Venturini, A simple storage scheme for strings achieving entropy bounds, Theoretical Computer Science 372 (1) (2007) 115–121.

[10] L. Foschini, R. Grossi, A. Gupta, J. Vitter, When indexing equals compression: Experiments with compressing suffix arrays and applications, ACM TALG 2 (4) (2006) 611–639.

[11] M. Fredman, M. Saks, The cell probe complexity of dynamic data structures, in: Proc. 43th ACM STOC, 1989, pp. 345–354.

[12] A. Golynski, I. Munro, S. Rao, Rank/select operations on large alphabets: a tool for text indexing, in: Proc. 17th SODA, 2006, pp. 368–373.

[13] R. González, G. Navarro, Statistical encoding of succinct data structures, in: Proc. 17th CPM, LNCS 4009, 2006, pp. 295–306.

[14] R. Grossi, A. Gupta, J. Vitter, High-order entropy-compressed text indexes, in: Proc. 14th SODA, 2003, pp. 841–850.

[15] A. Gupta, W.-K. Hon, R. Shah, J. S. Vitter, A framework for dynamizing succinct data structures, in: Proc. 34th International Colloquium on Automata, Languages and Programming (ICALP), 2007, pp. 521–532.

[16] W.-K. Hon, T.-W. Lam, K. Sadakane, W.-K. Sung, S.-M. Yiu, A space and time efficient algorithm for constructing compressed suffix arrays, Algorithmica 48 (1).

[17] W.-K. Hon, K. Sadakane, W.-K. Sung, Succinct data structures for searchable partials sums, in: Proc. 14th ISAAC, LNCS 2906, 2003, pp. 505–516.

[18] J. Kärkkäinen, Fast BWT in small space by blockwise suffix sorting, in: Proc. DIMACS Workshop on the Burrows-Wheeler Transform: Ten Years Later, 2004.

[19] S. Lee, K. Park, Dynamic rank-select structures with applications to run-length encoded texts, in: Proc. 18th CPM, LNCS 4580, 2007, pp. 95–106.

[20] V. Mäkinen, G. Navarro, Dynamic entropy-compressed sequences and full-text indexes, in: Proc. 17th CPM, LNCS 4009, 2006, pp. 307–318, extended version to appear in *ACM TALG*.

[21] V. Mäkinen, G. Navarro, Dynamic entropy-compressed sequences and full-text indexes, ACM TALG. To appear. Also as Technical Report TR/DCC-2006-10, Dept. of Computer Science, Univ. of Chile, July 2006, `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/dynamic.ps.gz`.

[22] V. Mäkinen, G. Navarro, Implicit compression boosting with applications to self-indexing, in: Proc. 14th SPIRE, LNCS 4726, 2007, pp. 214–226.

[23] U. Manber, G. Myers, Suffix arrays: A new method for on-line string searches, SIAM Journal of Computing 22 (1993) 935–948.

[24] G. Manzini, An analysis of the Burrows-Wheeler transform, Journal of the ACM 48 (3) (2001) 407–430.

[25] I. Munro, Tables, in: Proc. 16th FSTTCS, LNCS 1180, 1996, pp. 37–42.

[26] G. Navarro, V. Mäkinen, Compressed full-text indexes, ACM Computing Surveys 39 (1) (2007) article 2.

[27] M. Patrascu, E. Demaine, Logarithmic lower bounds in the cell probe model, SIAM Journal on Computing 35 (4) (2006) 932–963.

[28] S. Puglisi, W. Smyth, A. Turpin, A taxonomy of suffix array construction algorithms, ACM Computing Surveys 39 (2) (2007) article 4.

[29] R. Raman, V. Raman, S. Rao, Succinct indexable dictionaries with applications to encoding k-ary trees and multisets, in: Proc. 13th SODA, 2002, pp. 233–242.

[30] R. Raman, S. S. Rao, Succinct dynamic dictionaries and trees, in: Proc. 30th ICALP, LNCS 2719, 2003, pp. 357–368.

[31] K. Sadakane, R. Grossi, Squeezing succinct data structures into entropy bounds, in: Proc. 17th SODA, 2006, pp. 1230–1239.