OPTIMAL DYNAMIC SEQUENCE REPRESENTATIONS *

GONZALO NAVARRO† AND YAKOV NEKRICH‡

Abstract. We describe a data structure that supports access, rank and select queries, as well as symbol insertions and deletions, on a string $S[1,n]$ over alphabet $[1..\sigma]$ in time $O(\lg n/\lg\lg n)$, which is optimal even on binary sequences and in the amortized sense. Our time is worst-case for the queries and amortized for the updates. This complexity is better than the best previous ones by a $\Theta(1+\lg\sigma/\lg\lg n)$ factor. We also design a variant where times are worst-case, yet rank and updates take $O(\lg n)$ time. Our structure uses $nH_0(S)+o(n\lg\sigma)+O(\sigma\lg n)$ bits, where $H_0(S)$ is the zero-order entropy of S. Finally, we pursue various extensions and applications of the result.

1. Introduction. String representations supporting rank and select queries are fundamental in many data structures, including full-text indexes [26, 20, 23], permutations [23, 2], inverted indexes [11, 2], graphs [18], document retrieval indexes [55], labeled trees [23, 5], XML indexes [28, 19], binary relations [5], and many more. The problem is to encode a string $S[1,n]$ over alphabet $\Sigma=[1..\sigma]$ so as to support the following queries:

$$\begin{aligned}\mathsf{rank}_a(S,i) &= \text{ number of occurrences of } a\in\Sigma \text{ in } S[1,i], \text{ for } 1\le i\le n.\\ \mathsf{select}_a(S,i) &= \text{ position in } S \text{ of the } i\text{-th occurrence of } a\in\Sigma, \text{ for } 1\le i\le \mathsf{rank}_a(S,n).\\ \mathsf{access}(S,i) &= S[i].\end{aligned}$$

There exist various *static* representations of S (i.e., S cannot change) that support these operations [26, 23, 20, 2, 7]. The most recent work [7] shows a lower bound of $\Omega(\lg\frac{\lg\sigma}{\lg w})$ time for operation rank on a RAM machine with w-bit words, using any space of the form $O(n\lg^{O(1)}n)$. It also provides a matching upper bound that in addition achieves almost constant time for select and access, using compressed space. Thus the problem for static representations is essentially closed.

However, various applications need dynamism, that is, the ability to update S via insertions and deletions of symbols. Formally:

$$\begin{aligned}\mathsf{insert}_a(S,i) &: \text{ inserts } a\in\Sigma \text{ between } S[i-1] \text{ and } S[i], \text{ for } 1\le i\le n.\\ \mathsf{delete}(S,i) &: \text{ deletes } S[i] \text{ from } S, \text{ for } 1\le i\le n.\end{aligned}$$

A lower bound for this case, in order to just support operations rank, insert and delete, even for bit vectors ($\sigma=2$) and in the amortized sense, is $\Omega(\lg n/\lg\lg n)$ [22]. On the other hand the best known upper bound [29, 49] is $O((1+\lg\sigma/\lg\lg n)\lg n/\lg\lg n)$, that is, a factor $\Theta(\lg\sigma/\lg\lg n)$ away from the lower bound for alphabets larger than polylogarithmic. Their space is $nH_0(S)+o(n\lg\sigma)$ bits, where $H_0(S)=\sum_{a\in\Sigma}(n_a/n)\lg(n/n_a)\le\lg\sigma$ is the zero-order entropy of S, n_a being the number of occurrences of a in S.

In this paper we close this gap by providing an *optimal-time* dynamic representation of sequences. Our representation takes $O(\lg n/\lg\lg n)$ time for all the operations,

*Partially funded by Millennium Nucleus Information and Coordination in Networks ICM/FIC P10-024F, Chile. An early partial version of this paper appeared in *Proc. SODA'13*.

†Department of Computer Science, University of Chile. gnavarro@dcc.uchile.cl.

‡David R. Cheriton School of Computer Science, University of Waterloo. yakov.nekrich@googlemail.com.

worst-case for the three queries and amortized for updates. We present a second variant achieving worst-case bounds for all the operations, $O(\lg n/\lg\lg n)$ for select and access and $O(\lg n)$ for rank, insert and delete. The space is also $nH_0(S)+o(n\lg\sigma)$ bits. Time $O(\lg n)$ is still faster than previous work for $\lg\sigma=\Omega((\lg\lg n)^2)$. This gets much closer to closing this problem under the dynamic scenario as well.

We then show how to handle general alphabets, such as $\Sigma=\mathbb{R}$, or $\Sigma=\Gamma^*$ for a symbol alphabet Γ, in optimal time. For example, in the comparison model for $\Sigma=\mathbb{R}$, the time is $O(\lg\sigma+\lg n/\lg\lg n)$, where σ is the number of distinct symbols that appear in S; in the case $\Sigma=\Gamma^*$ for general Γ, the time is $O(|a|+\lg\gamma+\lg n/\lg\lg n)$, where $|a|$ is the length of the involved symbol (a string) and γ the number of distinct symbols of Γ that appear in the elements of S. Previous dynamic solutions have assumed that the alphabet $[1..\sigma]$ was static. An exception for the case $\Sigma=\Gamma^*$ [27] obtains $O(|a|\lg\gamma\lg n)$ time for all the operations.

At the end, we describe several applications where our result offers improved time/space tradeoffs. These include compressed indexes for dynamic text collections, construction of the Burrows-Wheeler transform [12] and construction of static compressed text indexes within compressed space, among others.

We start with an overview of the state of the art, putting our solution in context, in Section 2. We review the wavelet tree data structure [26], which is fundamental in our solution (and in most previous ones) in Section 3. In Section 4 we describe the core of our amortized solution, deferring to Section 5 the management of deletions and its relation with a split-find data structure needed for rank and insert. Section 6 encapsulates a technical part related to the structure of blocks that handle subsequences of polylogarithmic size. Section 7 deals with the changes in $\lg n$ and how we obtain times independent of σ, and concludes with Theorem 7.1, our result on uncompressed sequences. Then Section 8 shows how to improve the data encoding to obtain compressed space in Theorem 8.1, and Section 9 shows how to obtain worst-case times, Theorem 9.1. Finally, Section 10 describes some extensions and applications of our results. We conclude in Section 11.

2. Related Work. With one exception [28], all the previous work on dynamic sequences build on the *wavelet tree* structure [26]. The wavelet tree decomposes S hierarchically. In a first level, it separates larger from smaller symbols, marking in a bit vector which symbols of S are larger and which are smaller. The two subsequences of S are recursively separated. The $\lg\sigma$ levels of bit vectors describe S, and access, rank and select operations on S are carried out via $\lg\sigma$ rank and select operations on the bit vectors (see Section 3 for more details).

In the static case, rank and select operations on bit vectors take constant time, and therefore access, rank and select on S takes $O(\lg\sigma)$ time [26]. This can be reduced to $O(1+\lg\sigma/\lg\lg n)$ by using multiary wavelet trees [20]. These separate the symbols into $\rho=\Theta(\lg^\varepsilon n)$ ranges, for any constant $0<\varepsilon<1$, and instead of bit vectors store sequences over an alphabet of size ρ. On an alphabet of that size, rank and select can still be solved in constant time, and thus the time on the wavelet tree operations is reduced to $O(\lceil\lg\sigma/\lg\rho\rceil)$.

Insertions and deletions in S can also be carried out by inserting and deleting bits from $\lg\sigma$ bit vectors. However, the operations on dynamic bit vectors are bound to be slower. Fredman and Saks [22] show that $\Omega(\lg n/\lg\lg n)$ time is necessary, even in the amortized sense, to support rank, insert and delete operations on a bit vector. By using dynamic bit vector solutions [30, 14, 8, 13, 32] on the wavelet tree levels, one immediately obtains a dynamic sequence representation, where the space and the

time of the dynamic bit vector solution is multiplied by $\lg\sigma$ (the sum of the zero-order entropies of the bit vectors adds up to $nH_0(S)$ [26]). With this combination one can obtain times as good as $O(\lg\sigma\lg n/\lg\lg n)$ (using $n\lg\sigma+o(n\lg\sigma)$ bits) [13], or spaces as good as $O(nH_0(S))$ bits (with $O(\lg\sigma\lg n)$ time) [8].[1]

Mäkinen and Navarro [38, 39] made the above combination explicit, and obtained $O(\lg\sigma\lg n)$ time for all the sequence operations coupled with the best compressed space until then, $nH_0(S)+o(n\lg\sigma)$ bits. They also obtained $O((1+\lg\sigma/\lg\lg n)\lg n)$ query time, but with an update time of $O(\lg\sigma\lg^{1+\varepsilon} n)$, for any constant $0<\varepsilon<1$. This was achieved by replacing binary with multiary wavelet trees, and obtaining $O(\lg n)$ query time for the operations on sequences over a small alphabet of size $o(\lg n)$.

Lee and Park [36, 37] pursued this path further, obtaining $O((1+\lg\sigma/\lg\lg n)\lg n)$ time for queries and update operations, yet the space was not compressed, $n\lg\sigma+o(n\lg\sigma)$ bits, and update times were amortized. Shortly after, González and Navarro [24, 25] obtained the best of both worlds, making all the times worst-case and compressing the space again to $nH_0(S)+o(n\lg\sigma)$ bits. Both solutions managed to solve all query and update operations in $O(\lg n)$ time on sequences over small alphabets of size $o(\lg n)$.

Finally, almost simultaneously, He and Munro [29] and Navarro and Sadakane [49] obtained the currently best result, $O((1+\lg\sigma/\lg\lg n)\lg n/\lg\lg n)$ time, still within the same compressed space. They did so by improving the times of the dynamic sequences on small alphabets to $O(\lg n/\lg\lg n)$, which as said is optimal even on bit vectors and in the amortized sense.

As mentioned, the solution by Gupta et al. [28] deviates from this path and is a general framework for using any static data structure and periodically rebuilding it. By using it over a given representation [23], it achieves $O(\lg\lg n)$ query time and $O(n^\varepsilon)$ amortized update time. It would probably achieve compressed space if combined with more recent static data structures [2]. This shows that query times can be significantly smaller if one allows for much higher update times. In this paper, however, we focus on achieving similar times for all the operations. Table 2.1 gives more details on previous and our new results.

Wavelet trees can also be used to model $n\times n$ grids of points, in which case $\sigma=n$. Bose et al. [10] used a wavelet-tree-like structure to solve range counting in optimal static time $O(\lg n/\lg\lg n)$, using operations slightly more complex than rank on the wavelet tree levels. It is conceivable that this can be turned into an $O((\lg n/\lg\lg n)^2)$ time algorithm using dynamic sequences on the wavelet tree levels. On the other hand, $\Omega((\lg n/\lg\lg n)^2)$ is a lower bound for dynamic range counting in two dimensions [52]. This suggests that it is unlikely to obtain better results for dynamic wavelet trees, and thus that the current path of progress on dynamic wavelet trees has reached its optimum in $O((1+\lg\sigma/\lg\lg n)\lg n/\lg\lg n)$ time.

In this paper we show that this dead-end can be broken by abandoning the implicit assumption that, to provide access, rank and select on S, we *must* provide rank and select on the bit vectors (or sequences over $[1..\rho]$). We show that all what is needed is to *track* positions of S downwards and upwards along the wavelet tree. It turns out that this tracking can be done in *constant* time per level, breaking the $\Theta(\lg n/\lg\lg n)$ per-level barrier.

As a result, we obtain the *optimal* time complexity $O(\lg n/\lg\lg n)$ for all the

[1] For simplicity, we are omitting an $O(\sigma\lg n)$ additive term present in the space complexities, which is $o(n\lg\sigma)$ as long as $\sigma=o(n)$. We do write it explicitly in the theorems.

TABLE 2.1

History of results on managing dynamic sequences $S[1,n]$ over alphabet $[1..\sigma]$, assuming $\sigma = o(n/\lg n)$ to simplify. Some results [30, 8, 14] were presented only for binary sequences and the result we give is obtained by using them in combination with wavelet trees. Column WA tells whether the update times are (W)orst-case or (A)mortized.

Source	Space (bits)	Query time	Update time	WA
[22]		$\Omega(\lg n/\lg\lg n)$ for rank + insert + delete		A
[30, 32]	$n\lg\sigma + O(n\lg\sigma(\lg\lg n)^2/\lg n)$	$O(\lg\sigma\lg n/\lg\lg n)$	$O(\lg\sigma(\lg n/\lg\lg n)^2)$	A
[14]	$O(n\lg\sigma)$	$O(\lg\sigma\lg n)$	$O(\lg\sigma\lg n)$	W
[8]	$O(nH_0(S)+\lg n)$	$O(\lg\sigma\lg n)$	$O(\lg\sigma\lg n)$	W
[38, 39]	$nH_0(S)+O(n\lg\sigma/\sqrt{\lg n})$	$O(\lg\sigma\lg n)$	$O(\lg\sigma\lg n)$	W
	$nH_0(S)+O(n\lg\sigma/\lg^{1/2-\varepsilon} n)$	$O((1+\frac{1}{\varepsilon}\lg\sigma/\lg\lg n)\lg n)$	$O(\frac{1}{\varepsilon}\lg\sigma\lg^{1+\varepsilon} n)$	W
[13]	$O(n\lg\sigma)$	$O(\lg\sigma\lg n/\lg\lg n)$	$O(\lg\sigma\lg n/\lg\lg n)$	W
[28]	$n\lg\sigma + O(n\lg\sigma/\lg\lg\sigma)$	$O(\frac{1}{\varepsilon}\lg\lg n+\lg\lg\sigma)$	$O(\frac{1}{\varepsilon}n^{\varepsilon})$	A
[36, 37]	$n\lg\sigma + O(n\lg\sigma/\sqrt{\lg n})+O(n)$	$O((1+\lg\sigma/\lg\lg n)\lg n)$	$O((1+\lg\sigma/\lg\lg n)\lg n)$	A
[24, 25]	$nH_0(S)+O(n\lg\sigma/\sqrt{\lg n})$	$O((1+\lg\sigma/\lg\lg n)\lg n)$	$O((1+\lg\sigma/\lg\lg n)\lg n)$	W
[29]	$nH_0(S)+O(n\lg\sigma/\sqrt{\lg n})$	$O((1+\lg\sigma/\lg\lg n)\lg n/\lg\lg n)$	$O((1+\lg\sigma/\lg\lg n)\lg n/\lg\lg n)$	W
[49]	$nH_0(S)+O(n\lg\sigma/(\varepsilon\lg^{1-\varepsilon} n))$	$O((1+\frac{1}{\varepsilon}\lg\sigma/\lg\lg n)\lg n/\lg\lg n)$	$O((1+\frac{1}{\varepsilon}\lg\sigma/\lg\lg n)\lg n/\lg\lg n)$	W
Ours	$nH_0(S)+O(n\lg\sigma/\lg^{1-\varepsilon} n)$	$O(\frac{1}{\varepsilon^2}\lg n/\lg\lg n)$	$O(\frac{1}{\varepsilon^2}\lg n/\lg\lg n)$	A
Ours	$nH_0(S)+O(nH_0(S)/\lg\lg n)+O(n\lg\sigma/\lg^{1-\varepsilon} n)$	$O(\frac{1}{\varepsilon^2}\lg n/\lg\lg n)$, $O(\frac{1}{\varepsilon}\lg n)$ for rank	$O(\lg n)$	W

queries (worst-case) and update operations (amortized), independently of the alphabet size. This is $\Theta(1 + \lg\sigma/\lg\lg n)$ times faster than what was believed to be the "ultimate" solution. Our space is $nH_0(S) + o(n\lg\sigma)$ bits, similar to previous solutions. We develop, alternatively, a data structure achieving worst-case time for all the operations, yet this raises to $O(\lg n)$ for rank, insert and delete.

Among the many applications of this result, it is worth mentioning that any dynamic sequence representation supporting rank and insert in $O(t(n))$ amortized time can be used to compute the Burrows-Wheeler transform (BWT) [12] of a sequence $S[1,n]$ in worst-case time $O(n\,t(n))$. Thus our results allow us to build the BWT in $O(n\lg n/\lg\lg n)$ time and compressed space. The best existing space-time tradeoffs are by Okanohara and Sadakane [51], who achieve optimal $O(n)$ time within $O(n\lg\sigma\lg\lg_\sigma n)$ bits, Hon et al. [31], who achieve $O(n\lg\lg\sigma)$ time with $O(n\lg\sigma)$ bits, and Kärkkäinen [34], who obtains $O(n\lg n + nv)$ time and $O(n\lg n/\sqrt{v})$ extra bits for a parameter v. Using less space allows us to improve BWT-based compressors (like Bzip2) by allowing them to cut the sequence into larger blocks, given a fixed amount of main memory for the compressor. Several other results will be mentioned in Section 10.

3. Basic Notation and Wavelet Trees. Let S be a string over alphabet $\Sigma = [1..\sigma]$. In this paper we will use a model for strings that, although it would be more complex than necessary for typical purposes, is adequate for our descriptions. A string S will be a *set of distinct elements* $S = \{s_1, s_2, \ldots, s_n\}$. Each *element* $s \in S$ will have two components, $s.\mathrm{key} \in \mathbb{N}$ and $s.\mathrm{chr} \in \Sigma$. We will write $S[i]$ to denote the element $s \in S$ with the i-th smallest $s.\mathrm{key}$ value, thus $S[i].\mathrm{chr}$ is what is usually regarded as the i-th character (or symbol) of S. We denote $S[i].\mathrm{str} = S[1].\mathrm{chr} \circ S[2].\mathrm{chr} \circ \ldots \circ S[|S|].\mathrm{chr}$, what is usually regarded as the string itself (an element of Σ^*) where $\circ$ is the concatenation operator. Sometimes we refer to S as a *sequence* of its elements $s \in S$, by taking them in increasing $s.\mathrm{key}$ order. Note that a subset of S corresponds to what is usually called a subsequence. When clear from context, we will write S instead of $S.\mathrm{str}$. Note that we will not represent any string S as a set of elements, but rather will use this conceptual model to describe our algorithms and data structures. Thus a string S can be physically represented using $|S|\lg\sigma$ bits by writing the symbols of $S.\mathrm{str}$.

We use this model also for binary strings B over alphabet $\Sigma = \{0,1\}$. To avoid confusion, elements of binary strings will be called *binary elements*, or *belements* for short. Thus, if b is a belement, $b.\mathrm{chr} \in \{0,1\}$ is a bit. We will reserve the term "bit vector" to describe physical sequences of bits. Thus $B.\mathrm{str}$ can be represented as a bit vector, using $|B|$ bits of space.

Now we describe the wavelet tree data structure [26]. We associate each $a \in \Sigma$ to a leaf v_a of a balanced binary tree $\mathcal{T}$. Each node if $\mathcal{T}$ is associated to a subset $\Sigma_v \subseteq \Sigma$, where $\Sigma_v = \{a \in \Sigma,\ v_a \text{ descends from } v\}$. In particular, $\Sigma_{v_a} = \{a\}$ for any $a \in \Sigma$, and $\Sigma_{v_r} = \Sigma$ for the root v_r of $\mathcal{T}$. The essential idea of the wavelet tree is the representation of a string S by binary strings stored in the nodes of $\mathcal{T}$. We associate a string $S(v) \subseteq S$ with every node v of $\mathcal{T}$, $S(v) = \{s \in S, s.\mathrm{chr} \in \Sigma_v\}$. The wavelet tree does not store strings $S(v)$ explicitly, but just binary strings $B(v)$ at internal nodes v. We set $B(v)[i].\mathrm{chr} = t$ if $S(v)[i] \in S(v_t)$, where v_t is the t-th child of v (the left child corresponds to $t = 0$ and the right to $t = 1$). This data structure (i.e., $\mathcal{T}$ and the bit vectors $B(v).\mathrm{str}$) is called the *wavelet tree* of S. Note that no bit vectors are stored for the leaf nodes v_a, as the conceptual strings $S(v_a)$ do not need to be represented. Since $\mathcal{T}$ has $O(\sigma)$ nodes and $\lceil\lg\sigma\rceil$ levels, and the bit vectors at each level add up to

length n, the wavelet tree requires $n\lceil \lg \sigma \rceil + O(\sigma \lg n)$ bits of space. If the bit vectors $B(v)$.str are compressed to $|B(v)|H_0(B(v)) + o(|B(v)|)$ bits, the total size adds up to $nH_0(S) + o(n \lg \sigma) + O(\sigma \lg n)$ bits [26] (we write $H_0(S)$ for $H_0(S.\text{str})$). There are various surveys on wavelet trees [48, 40, 47].

For any element $S[i]$ and every internal node v such that $S[i].\text{chr} \in \Sigma_v$, there is exactly one element $S(v)[j] = S[i]$. Then the belement $b_v = B(v)[j]$ indicates in which child of v is the leaf $v_{S(v)[j].\text{chr}} = v_{S[i].\text{chr}}$ stored. We will say that such b_v *encodes* $S[i]$ in $B(v)$, and will write $b_v.\text{enc} = S[i] = S(v)[j]$. We will also say that belement $b_v \in B(v)$ *corresponds* to a belement $b_u \in B(u)$ if $b_v.\text{enc} = b_u.\text{enc}$ in two nodes v and u on a path of $\mathcal{T}$. Identifying the belements that encode the same element plays a crucial role in wavelet trees. Other, more complex, operations rely on the ability to navigate in the tree and keep track of belements that encode the same element.

The wavelet tree encodes S, in the sense that it allows us to extract any $S[i].\text{chr}$. To implement $\mathsf{access}(S, i)$ we traverse a path from the root v_r to the leaf $v_{S[i].\text{chr}}$. In each visited node we read the belement b_v that encodes $S[i]$ and proceed to the corresponding belement in the $b_v.\text{chr}$-th child of v. Upon arriving to a leaf v_a we answer $\mathsf{access}(S, i) = a$.

The wavelet tree also implements operations rank and select. To compute $\mathsf{select}_a(S, i)$, we start at the (conceptual) element $S(v_a)[i]$ and identify the corresponding belement $b_v = B(v)[j]$ in the parent v of v_a, that is, $b_v.\text{enc} = B(v)[j].\text{enc} = S(v)[j] = S(v_a)[i]$. We continue this process towards the root until reaching a belement $B(v_r)[j]$ such that $B(v_r)[j].\text{enc} = S(v_r)[j] = S[j] = S(v_a)[i]$. Then the answer is $\mathsf{select}_a(S, i) = j$. Finally, to compute $\mathsf{rank}_a(S, i)$, we traverse the wavelet tree from element $S(v_r)[i]$ and $b_v = B(v_r)[i]$ to some element in the leaf v_a. At each node v in the path, we identify the child v_t of v such that $a \in \Sigma_{v_t}$, and then find the belement b' with $b'.\text{chr} = t$ and highest $b'.\text{key} \leq b_v.\text{key}$. Then we move to the belement $b_t = B(v_t)[j]$ corresponding to b' in v_t. Upon arriving at element $S(v_a)[j]$, the answer is $\mathsf{rank}(S, i) = j$.

The standard method used in wavelet trees for identifying corresponding belements is to maintain $\mathsf{rank}/\mathsf{select}$ data structures on the bit vectors $B(v)$.str. Let $B(v)[i].\text{chr} = t$, then we can find the *offset* j of the corresponding belement $B(v_t)[j]$ in the child v_t of v as $j = \mathsf{rank}_t(B(v).\text{str}, i)$. Conversely, we can find the offset j of the belement $B(v)[j]$ corresponding to $B(v_t)[i]$ as $j = \mathsf{select}_t(B(v).\text{str}, i)$. Finally, the more complicated process of finding the b' needed for $\mathsf{rank}_a(S, i)$ is easily solved using rank on $B(v)$.str: If $b_v = B(v)[i]$ and v_t is the t-th child of v, then without the need to find b' we know that its corresponding belement in v_t is $B(v_t)[j]$, for $j = \mathsf{rank}_t(B(v).\text{str}, i)$. This approach leads to $O(\lg \sigma)$ query times in the static case because $\mathsf{rank}/\mathsf{select}$ queries on a bit vector $B(v)$.str can be answered in constant time and $|B(v)|+o(|B(v)|)$ bits of space [46, 17], and even using $|B(v)|H_0(B(v))+o(|B(v)|)$ bits [53]. However, we need $\Omega(\lg n/\lg\lg n)$ time to support $\mathsf{rank}/\mathsf{select}$ and updates on a bit vector [22], which multiplies the operation times in the dynamic case.

An improvement (for both static and dynamic wavelet trees) can be achieved by increasing the fan-out of the wavelet tree to $\rho = \Theta(\lg^\varepsilon n)$ for a constant $0 < \varepsilon < 1$: the strings $B(v)$ are not anymore binary but all the definitions and procedures remain verbatim. This enables us to reduce the height of the wavelet trees and the query time by a $\Theta(\lg\lg n)$ factor, because the $\mathsf{rank}/\mathsf{select}$ times over alphabet $[1..\rho]$ are still constant in the static case [20] and $O(\lg n/\lg\lg n)$ in the dynamic case [29, 49]. However, it seems that further improvements that are based on dynamic $\mathsf{rank}/\mathsf{select}$ queries in every node are not possible.

In this paper we use a different approach to identifying the corresponding ele-

ments. We partition sequences $B(v)$ into blocks, which are stored in compact list structures $L(v)$. Pointers from selected elements in $L(v)$ to the structure $L(v_t)$ in children nodes v_t (and vice versa) enable us to navigate between nodes of the wavelet tree in constant time. We extend the idea to multiary wavelet trees.

Our ideas are related to the general concept of fractional cascading [15, 16], and in particular to its dynamic variant [43]. Similar techniques have also been used recently in some geometric data structures [9, 50]. However, applying them on compressed data structures where the bit budget is severely limited is much more challenging.

4. Basic Structure. We start by describing the main components of our modified wavelet tree. Then, we show how our structure supports $\mathsf{access}(S, i)$ and $\mathsf{select}_a(S, i)$. In the third part of this section we describe additional structures that enable us to answer $\mathsf{rank}_a(S, i)$. Finally, we show how to support updates. Along this and the next sections we will obtain time $O((\lg \sigma + \lg n)/\lg\lg n)$ for all the operations; in Section 7 we will obtain times fully independent of σ.

In the rest of the paper, we will use the term "elements" both for $S(v)[e]$ and $B(v)[e]$, as no confusion will arise.

4.1. Structure. We assume that the wavelet tree $\mathcal{T}$ has node degree $\rho = \Theta(\lg^\varepsilon n)$. We divide the sequences $B(v)$ into $g(v)$ *blocks* $G_1(v), G_2(v), \ldots, G_{g(v)}(v)$, and store those blocks in a doubly-linked list $L(v)$. Each block $G_j(v)$ contains $|G_j(v)| = \Theta(\lg^3 n/\lg\rho)$ consecutive elements from $B(v)$, except the last, which can be smaller. For each $G_j(v)$ we maintain a data structure $R_j(v)$ that supports access, rank and select queries on $G_j(v).\mathrm{str}$. Since a block contains a poly-logarithmic number of elements over an alphabet of size ρ, we can answer those queries in $O(1)$ time (this will be described later, in Section 6, because the details are rather technical).

The *location* of an element $B(v)[e]$ consists of two parts: (1) a unique identifier of the block $G_j(v)$ that contains the offset e, and (2) the *local index* of e within $G_j(v)$. Such a pair gives constant-time access to the element. A *pointer* to an element will indicate, precisely, its location.

We maintain pointers between selected corresponding elements in $L(v)$ and the lists of its parent and children.

- If an element $B(v)[e]$ is stored in a block $G_j(v)$ and $B(v)[e].\mathrm{chr} = t \neq B(v)[e'].\mathrm{chr}$ for all $e' < e$ in $G_j(v)$ (i.e., e is the first offset where symbol t occurs in $G_j(v).\mathrm{str}$), then we store a pointer from $B(v)[e]$ to the corresponding element $B(v_t)[e_t]$ in $L(v_t)$. Pointers are bidirectional, that is, we also store a pointer from $B(v_t)[e_t]$ to $B(v)[e]$.
- In addition, if e is the first offset in its block $G_j(v)$ (i.e., the location of $B(v)[e]$ is $(G_j(v), 1)$) and $B(u)[e']$ corresponds to $B(v)[e]$ in the parent u of v, then we store a pointer from $B(v)[e]$ to $B(u)[e']$ and, by bidirectionality, from $B(u)[e']$ to $B(v)[e]$.

All these pointers will be called *inter-node pointers*. We describe how they are implemented later in this section. Figure 4.1 shows an example (disregard for now the dashed arrows).

It is easy to see that the number of inter-node pointers from $L(v)$ to $L(v_t)$, for any fixed t, is $O(g(v))$. Hence, the total number of pointers that point downwards from a node v is $O(g(v)\rho)$. Additionally, there are $O(g(v))$ upward pointers to the parent of v. Thus, the total number of inter-node pointers in the wavelet tree equals $O(\sum_{v\in\mathcal{T}} g(v)\rho) = O(n\lg\sigma/\lg^{3-\varepsilon} n + \sigma\lg^\varepsilon n)$, where the term $\sigma\lg^\varepsilon n$ accounts for the v nodes that have just one block, $G_1(v)$. Since the children v_t of those nodes must also have just one block, $G_1(v_t)$, we avoid storing their pointers, as we know that all point

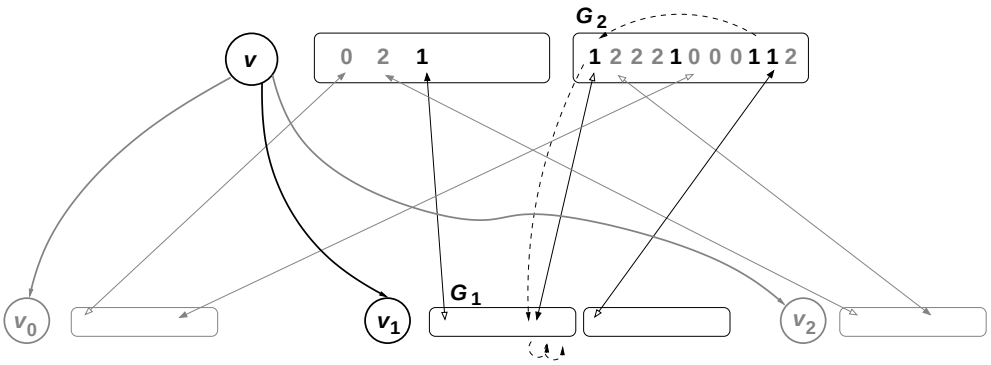

FIG. 4.1. *An example of our data structure with $\rho = 3$. Circles represent wavelet tree nodes. The parent node v has children v_0, v_1 and v_2. We emphasize the relation between v and v_2; the rest are grayed. Each node shows (part of) its list L, focusing on the relation between $G_2(v)$ and its children. Each block G_j is a rectangle. Bidirectional pointers are shown with solid arrows, with filled arrowheads marking their original direction. Thus one pointer leaves $G_2(v)$ from the first occurrence of each t towards its corresponding element in $L(v_t)$. Also, the first element of each block in $L(v_t)$ points upward. The dashed arrows illustrate the process of tracking downwards $G_2(v)[9]$, the third 1. The first step finds the closest previous 1 that has a downward pointer. It is $G_2(v)[1]$. The second step is to follow the pointer towards $G_1(v_1)$. The third step is to advance by 2 in $G_1(v_1)$, because* $\mathsf{rank}_1(G_2(v).\mathrm{str}, 9) - \mathsf{rank}_1(G_2(v).\mathrm{str}, 1) = 2$.

to the same block $G_1(v_t)$, and their index inside $G_1(v_t)$ can be found with constant-time rank/select operations inside $G_1(v)$. Similarly, if the parent u of v also has only one block, $G_1(u)$, we avoid storing explicit pointers between $G_1(v)$ and $G_1(u)$, as they can be computed in constant time with rank/select inside $G_1(u)$. If, instead, u has more than one block, then we explicitly represent the pointers between $L(v)$ and $L(u)$ and charge their space to u. This yields the cleaner expression $O(n \lg \sigma / \lg^{3-\varepsilon} n)$ for the number of pointers.

The pointers leaving from a block $G_j(v)$ are stored in a data structure $F_j(v)$. Using $F_j(v)$ we can find, for any offset e in $G_j(v)$ and any $0 \le t < \rho$, the last offset $e' \le e$ in $G_j(v)$ such that there is a pointer from $B(v)[e']$ to an element $B(v_t)[e'_t]$ in $L(v_t)$. We describe in Section 6 how $F_j(v)$ implements the queries and updates in constant time.

A *dynamic partial-sums data structure* maintains m nonnegative integers $x_1, \ldots, x_m$, and supports two queries: $sum(j) = \sum_{i=1}^{j} x_i$ and $search(v) = \max\{j, sum(j) \le v\}$, as well as updates to elements x_j by $\pm O(\lg m)$, and inserting and deleting elements $x_j = 0$. Furthermore, we can associate satellite data y_j to each x_j, so that $search(v)$ can return y_j instead of simply j. The following lemma is useful.

LEMMA 4.1 ([49, Lem. 1], adapted). *A dynamic partial-sums data structure over m elements with satellite data supports operations sum, search, updates by $O(\lg m)$, and insertions/deletions of zero values in $O(\lg m / \lg\lg m)$ worst-case time per operation, using $O(m \lg m)$ bits of space.*

In addition to the pointers, we store, for the root node v_r, a dynamic searchable partial-sums data structure $K(v_r)$ on the blocks of $L(v_r)$: the values x_j are the sizes $|G_j(v_r)|$, the satellite data y_j is a reference to block $G_j(v_r)$, and $\lg m = \Theta(\lg n)$. Using $K(v_r)$, query $search(e)$ returns the block $G_j(v_r)$ that contains the element $S(v_r)[e]$, and query $sum(j-1)$ returns the sizes of all the blocks that precede $G_j(v_r)$. The same data structures $K(v_a)$ are also stored in the leaves v_a of $\mathcal{T}$. Since $g(v_r) = O(n \lg \rho / \lg^3 n)$, and also $\sum_{a \in \Sigma} g(v_a) = O(n \lg \rho / \lg^3 n)$, we store $O(n \lg\lg n / \lg^3 n)$

TABLE 4.1

Structures inside any node v of the wavelet tree $\mathcal{T}$, or only in the root node v_r and the leaves v_a. The third column gives the extra space in bits, on top of the data, for the whole structure.

Structure	Meaning	Extra space in bits
$L(v)$	List of blocks storing $B(v)$	$O(n \lg \sigma / \lg^2 n + \sigma \lg n)$
$G_j(v)$	j-th block of list $L(v)$	$O(n \lg \sigma (\lg \lg n)^2 / \lg n + \sigma \lg n)$
$R_j(v)$	Supports rank/select/access inside $G_j(v)$	$O(n \lg \sigma \lg \lg n / \lg^{1-\varepsilon} n)$
$F_j(v)$	Pointers leaving from $G_j(v)$	$O(n \lg \sigma \lg \lg n / \lg^{1-\varepsilon} n)$
$H_j(v)$	Pointers arriving at $G_j(v)$	$O(n \lg \sigma / \lg^2 n)$
$P_t(v)$	Predecessor in $L(v)$ containing symbol t	$O(n \lg \sigma / \lg^{2-\varepsilon} n)$
$K(v)$	Partial sums on block lengths for v_r and v_a	$O(n \lg \lg n / \lg^2 n)$
$D_j(v)$	Deleted elements in $G_j(v)$, for v_r and v_a	$O(n (\lg \lg n)^2 / \lg n)$
DEL	Global list of deleted elements in $\overline{S}$	$O(n / \lg n + n \lg \sigma / \lg^2 n)$

elements in the partial sums $K(v_r)$ and $K(v_a)$, for an overall size of $O(n \lg \lg n / \lg^2 n)$ bits.

We recall that we do not store a sequence $B(v_a)$ in a leaf node v_a, only in internal nodes. Nevertheless, we divide the (implicit) sequence $B(v_a)$ into blocks and store their sizes in $K(v_a)$; we maintain $K(v_a)$ only if $L(v_a)$ consists of more than one block. Moreover we store inter-node pointers from the parent of v_a to v_a and vice versa. Pointers in a leaf are maintained using the same rules of any other node.

For future reference, we provide the list of secondary data structures in Table 4.1. They will be described in detail along the next sections.

4.2. Access and Select Queries. Assume the location $(G_j(v), i_v)$ of an element $B(v)[e]$ in $L(v)$ is known, and $B(v)[e].\text{chr} = t$. Then, the location of the corresponding element $B(v_t)[e_t]$ in $L(v_t)$ is computed as follows. Using $F_j(v)$, we find the local index i'_v of the largest offset $e' \leq e$ in $G_j(v)$ such that there is a pointer from $B(v)[e']$ to some $B(v_t)[e'_t]$ in $L(v_t)$. Due to our construction, such e' must exist (it may be e itself), since the first such e' in each block has a pointer. Let $(G_\ell(v_t), i'_t)$ be the location of $B(v_t)[e'_t]$ in $L(v_t)$. Due to our rules to define pointers, $B(v_t)[e_t]$ also belongs to $G_\ell(v_t)$, since if it belonged to another block $G_m(v_t)$, the upward pointer from the first offset of $G_m(v_t)$ would point between e' and e, and since pointers are bidirectional, this would contradict the definition of e'. Furthermore, let $r_v = \mathsf{rank}_t(G_j(v).\text{str}, i_v)$ and $r'_v = \mathsf{rank}_t(G_j(v).\text{str}, i'_v)$. Then the local index of e_t is $i'_t + (r_v - r'_v)$. Thus we can find the location of $B(v_t)[e_t]$ in $O(1)$ time if the location of $B(v)[e]$ is known. The dashed arrows in Figure 4.1 show an example.

Analogously, assume we know the location $(G_j(v_t), i_t)$ of $B(v_t)[e_t]$ and want to find the location of the corresponding element $B(v)[e]$ in its parent node v. Using $F_j(v_t)$ we find the last offset $e'_t \leq e_t$ in $G_j(v_t)$ such that there is an upward pointer from $B(v_t)[e'_t]$. Offset e'_t exists by construction (it can be the upward pointer from the first index in $G_j(v_t)$ or the reverse of some pointer from $L(v)$ to $G_j(v_t)$). Let $B(v_t)[e'_t]$ point to $B(v)[e']$, with location $(G_\ell(v), i'_v)$. Then, by our construction, $B(v)[e]$ is also in $G_\ell(v)$, since if it belonged to a different block $G_m(v)$, then there would be a pointer from the first occurrence of t in $G_m(v).\text{str}$ pointing between e'_t and e_t, and its bidirectional version would contradict the definition of e'_t. Furthermore, let i'_t be the local index of e'_t in $G_j(v_t)$. Then the local index of $B(v)[e]$ is $i_v = \mathsf{select}_t(G_\ell(v).\text{str}, \mathsf{rank}_t(G_\ell(v).\text{str}, i'_v) + (i_t - i'_t))$.

To solve $\mathsf{access}(S, i)$, we visit the nodes $v_0 = v_r, v_1, \ldots, v_h = v_a$, where $h = \lg_\rho \sigma$ is the height of $\mathcal{T}$, v_k is the t_k-th child of v_{k-1} and $B(v_{k-1})[e_{k-1}].\mathrm{chr} = t_k$ encodes $S[i]$. We do not compute the offsets $e_1, \ldots, e_h$, but just their locations. The location of $B(v_r)[e_0 = i]$ is found in $O(\lg n / \lg\lg n)$ time using the partial-sums structure $K(v_r)$. If the location of $B(v_{k-1})[e_{k-1}]$ is known, we can find that of $B(v_k)[e_k]$ in $O(1)$ time, as explained. When a leaf node $v_h = v_a$ is reached, we have $S[i] = a$.

To solve $\mathsf{select}_a(S, i)$, we set $e_h = i$ and identify the location of $B(v_a)[e_h]$ in the list $L(v_a)$ of the leaf v_a, using structure $K(v_a)$. Then we traverse the path $v_h = v_a, v_{h-1}, \ldots, v_0 = v_r$ where v_{k-1} is the parent of v_k, until the root node is reached. In every node v_k, we find the location of $B(v_{k-1})[e_{k-1}]$ in $L(v_{k-1})$ that corresponds to $B(v_k)[e_k]$, as explained above. Finally, we compute the number of elements that precede e_0 in $L(v_r)$ using structure $K(v_r)$.

Thus access and select require $O(\lg_\rho \sigma + \lg n / \lg\lg n) = O((\lg \sigma + \lg n)/\lg\lg n)$ worst-case time.

4.3. Rank Queries. We need some additional data structures for the efficient support of rank queries. In every node v such that $L(v)$ consists of more than one block, we store a data structure $P(v)$. Using $P(v)$ we can find, for any $0 \le t < \rho$ and for any block $G_j(v)$, the last block $G_\ell(v)$ such that $\ell \le j$ and $G_\ell(v)$ contains an element $B(v)[e]$ with $B(v)[e].\mathrm{chr} = t$. $P(v)$ consists of ρ predecessor data structures $P_t(v)$ for $0 \le t < \rho$. We describe in Section 5 a way to support these predecessor queries in constant time in our scenario.

Let the location of $B(v)[e]$ be $(G_j(v), i)$. Structure $P(v)$ enables us to find the last offset $e' \le e$ such that $B(v)[e'].\mathrm{chr} = t$. First, we use $R_j(v)$ to compute $r = \mathsf{rank}_t(G_j(v).\mathrm{str}, i)$. If $r > 0$, then e' belongs to the same block $G_j(v)$, and its local index is $\mathsf{select}_t(G_j(v).\mathrm{str}, r)$. Otherwise, we use $P_t(v)$ to find the last block $G_\ell(v)$ that precedes $G_j(v)$ and $G_\ell(v).\mathrm{str}$ contains an occurrence of t. Then we find the local index of the last such occurrence in $G_\ell(v).\mathrm{str}$ using $R_\ell(v)$.

Now we are ready to describe the procedure to answer $\mathsf{rank}_a(S, i)$. The symbol a is represented as a concatenation of symbols $t_0 \circ t_1 \circ \ldots \circ t_h$, where each $t_k \in [1..\rho]$. We traverse the path from the root $v_r = v_0$ to the leaf $v_a = v_h$. We find the location of $e_0 = i$ in v_r using the data structure $K(v_r)$. In each node v_k, $0 \le k < h$, we identify the location of the last element $B(v_k)[e'_k]$ such that $e'_k \le e_k$ and $B(v_k)[e'_k].\mathrm{chr} = t_k$, using $P_{t_k}(v_k)$ as explained. From the location of $B(v_k)[e'_k]$ we find the location of the corresponding element $B(v_{k+1})[e_{k+1}]$ in $L(v_{k+1})$, just as done for access.

When our procedure reaches the leaf node $v_h = v_a$, the (virtual) element $B(v_h)[e_h]$ encodes the last occurrence of a in $S[1, i]$. Note that we know the location $(G_\ell(v_h), i_h)$ of $B(v_h)[e_h]$, not the offset e_h itself. Then we find the number r of elements in all the blocks that precede $G_\ell(v_h)$ using $K(v_a)$. Finally, $\mathsf{rank}_a(S, i) = r + i_h$.

Therefore, the total time for rank is also $O(\lg_\rho \sigma + \lg n / \lg\lg n) = O((\lg \sigma + \lg n)/\lg\lg n)$.

4.4. Insertions. Now we describe how inter-node pointers are implemented. We say that an element is *pointed* if there is a pointer to it. We cannot directly store the local index of a pointed element in the pointer: when a new element is inserted into a block, the indexes of all the elements that follow it are incremented by 1. Since a block can contain $\Theta(\lg^3 n / \lg \rho)$ pointed elements, we would have to update that many pointers after each insertion and deletion.

Therefore we resort to the following two-level scheme. Each pointed element in a block is assigned a unique identifier. When a new element is inserted, we assign it the identifier $max\_id + 1$, where max_id is the maximum identifier value used so far. We

also maintain a data structure $H_j(v)$ for each block $G_j(v)$, which enables us to find the local index of a pointed element given its identifier in $G_j(v)$. The implementation of $H_j(v)$ is based on standard word RAM techniques and a table that contains the identifiers of the pointed elements; the details are given in Section 6.

Now we describe how to insert a new symbol a at position i in S. Let $e_0, e_1, \ldots, e_h$ be the offsets of the elements that will encode $a = t_0 \circ \ldots \circ t_h$ in $v_r = v_0, v_1, \ldots, v_h = v_a$. We can find the location of $B(v_r)[e_0 = i]$ in $L(v_r)$ in $O(\lg n / \lg\lg n)$ time using $K(v_r)$, and then insert a new element b_{v_r} with $b_{v_r}.\mathrm{chr} = t_0$ at that location, so that now $B(v_r)[e_0] = b_{v_r}$. Now, given the location of $B(v_k)[e_k]$ in $L(v_k)$, where $B(v_k)[e_k].\mathrm{chr} = t_k$ has just been inserted, we find the location of $B(v_k)[e'_k]$, for the largest offset $e'_k < e_k$ such that $B(v_k)[e'_k].\mathrm{chr} = t_k$, in the same way as for rank queries.[2] From the location of $B(v_k)[e'_k]$, we find the location of the corresponding element, $B(v_{k+1})[e''_{k+1}]$, in $L(v_{k+1})$. The symbol t_{k+1} must then be inserted into $L(v_{k+1})$ immediately after $B(v_{k+1})[e''_{k+1}]$, that is, at offset $e_{k+1} = e''_{k+1} + 1$.

The insertion of a new element b_{v_k} into a block $G_j(v_k)$ is handled by structure $R_j(v_k)$ and the memory manager of the block. We must also update structures $F_j(v_k)$ and $H_j(v_k)$ to keep the correct alignments, and possibly to create and destroy a constant number inter-node pointers to maintain our invariants. Also, since pointers are bidirectional, a constant number of inter-node pointers in the parent and children of node v_k may be updated. All those changes can be done in $O(1)$ time; see Section 6 for the details. Insertions may also require updating structures $P_t(v_k)$, which takes $O(1)$ amortized time, see Section 5. Finally, if v_k is the root node or a leaf, we also update $K(v_k)$. This update is only by ± 1, so we recall it requires just $O(\lg n / \lg\lg n)$ time.

If $|G_j(v_k)|$ exceeds $2\lg^3 n / \lg \rho$, we split $G_j(v_k)$ evenly into two blocks, $G_{j_1}(v_k)$ and $G_{j_2}(v_k)$. Then, we rebuild the data structures R, F and H for the two new blocks. Note that there are inter-node pointers to $G_j(v_k)$ that now could become dangling pointers, but all those can be known from $F_j(v_k)$, since pointers are bidirectional, and updated to point to the right locations in $G_{j_1}(v_k)$ or $G_{j_2}(v_k)$. Finally, if v_k is the root or a leaf, then $K(v_k)$ is updated.

The total cost of splitting a block is dominated by that of building the new data structures R, F and H. These are easily built in $O(\lg^3 n / \lg \rho)$ time. Since we split a block $G_j(v)$ at most once per sequence of $\Theta(\lg^3 n / \lg \rho)$ insertions in $G_j(v)$, the amortized cost incurred by splitting a block is $O(1)$. Therefore the total amortized cost of an insertion in $L(v)$ is $O(1)$. The insertion of a new symbol leads to $O(\lg_\rho \sigma)$ insertions into lists $L(v_k)$.

To update $K(v_k)$, upon an overflow in $G_j(v_k)$, we add a new element before or after x_j. Lemma 4.1 does not support updates with large values. Inserting a new value $x_{j+1} = 0$, then increasing it up to $x_{j+1} = |G_{j_2}(v_k)|$, and then decreasing the value of $x_j = |G_j(v_k)|$ to make it $x_j = |G_{j_1}(v_k)|$, can be done in $O(\lg^3 n/(\lg \rho \lg\lg n))$ time by adding/subtracting $O(\lg n)$ units at a time. Each such increment/decrement and insertion takes $O(\lg n / \lg\lg n)$ time, and we carry it out $O(|G_j(v_k)| / \lg n) = O(\lg^2 n / \lg \rho)$ times. Still, this total cost amortizes to $o(1)$ per operation.

Hence, the total amortized cost of an insertion is $O(\lg_\rho \sigma + \lg n / \lg\lg n) = O((\lg \sigma + \lg n) / \lg\lg n)$.

[2] We cannot descend using e_k itself, as for access, because the inter-node pointer invariants have not yet been restored for the new elements.

Next we describe how deletions are handled, and we also describe the data structure $P(v)$.

5. Lazy Deletions and Data Structure $P(v)$. We do not process deletions immediately, but in lazy form: we do not maintain exactly S but a supersequence $\overline{S}$ of it. When $S[i]$ is deleted from S, we retain it in $\overline{S}$ but mark $S[i]$ as deleted. When the number of elements marked as deleted exceeds a certain threshold, we expunge them all from the data structure. We define $\overline{B}(v)$ and the list $\overline{L}(v)$ for the sequence $\overline{S}$ in the same way as $B(v)$ and $L(v)$ are defined for S.

We will make use of *split-find* data structures. This structure maintains a sequence of objects, some of which are marked. Operation *split*(x) marks object x. Operation *find*(x) gives the rightmost marked object that is not after x in the sequence. A new object can be inserted immediately before or after an object x in the sequence. A split-find data structure [33] can implement operation *find* in constant time, and operation *split* and insertions in amortized constant time.

Since elements of $\overline{L}(v)$ are never removed, we can implement $P(v)$ using split-find data structures. For each t, $0 \le t < \rho$, $P_t(v)$ will be a split-find data structure containing one object per block $G_j(v)$ in $L(v)$. The marked objects in $P_t(v)$ are the blocks $G_j(v)$ that contain an occurrence of t, so *find*($G_j(v)$) gives the block $G_\ell(v)$ for the maximum $\ell \le j$ such that $G_\ell(v)$.str contains a t, in constant time as desired.

The insertion of a symbol t in $G_j(v)$ may induce a new split in $P_t(v)$, if $G_j(v)$ was not already marked. Furthermore, overflows in $G_j(v)$, which convert it into two blocks $G_{j_1}(v)$ and $G_{j_2}(v)$, induce insertions in $P_t(v)$. Note that an overflow in $G_j(v)$ triggers ρ insertions in the $P_t(v)$ structures, but the resulting $O(\rho)$ time amortizes to $o(1)$ because overflows occur every $\Theta(\lg^3 n/\lg\rho)$ insertions.

Structures $P_t(v)$ do not support "unsplitting" nor removals. The replacement of $G_j(v)$ by $G_{j_1}(v)$ and $G_{j_2}(v)$ is implemented by leaving in $P_t(v)$ the object corresponding to $G_j(v)$ and inserting one corresponding to either $G_{j_1}(v)$ or $G_{j_2}(v)$. If $G_j(v)$.str contained t, then at least one of $G_{j_1}(v)$.str and $G_{j_2}(v)$.str must contain t, and the other can be inserted as a new element (possibly followed by a split, if it also contains t).

We will maintain partial-sums structures $\overline{K}(v)$, storing the number of non-deleted elements in each block of $\overline{L}(v)$, for the root node $v = v_r$ and each leaf node, $v = v_a$. Moreover, we maintain a new data structure $D_j(v)$ for every block $G_j(v)$, where v is either the root or a leaf node. $D_j(v)$ returns, in constant time, the number of non-deleted elements in $G_j(v)[1,i]$, for any local index i, as well the local index in $G_j(v)$ of the i-th non-deleted element. The implementation of $D_j(v)$ is described in Section 6. We use $\overline{K}(v)$ and $D_j(v)$ to find the offset $\overline{e}$ in $\overline{L}(v)$ of the e-th non-deleted element, and to count the number of non-deleted elements that occur up to offset $\overline{e}$ in $\overline{L}(v)$.

We also store a global list *DEL* that contains, in any order, all the elements marked as deleted that have not yet been expunged from the wavelet tree. For any element $\overline{S}[i]$ in *DEL* we store a pointer to $\overline{B}(v_r)[i]$ in $\overline{L}(v_r)$. These are implemented in the same way as inter-node pointers.

5.1. Queries. Queries are answered very similarly to Section 4. The main idea is that we can essentially ignore deleted elements except at the root and at the leaves.

$\mathsf{access}(S,i)$**:** We do exactly as in Section 3, except that e_0 refers to the i-th non-deleted element in $\overline{L}(v_r)$, and is found using $\overline{K}(v_r)$ and $D_j(v_r)$.

$\mathsf{select}_a(S,i)$**:** We find the location of the i-th non-deleted element in $\overline{L}(v_h)$, where $v_h = v_a$, and the offset e_h is found using $\overline{K}(v_a)$ and some $D_j(v_a)$. Then we

move up in the tree exactly as in Section 4. When the root node $v_0 = v_r$ is reached, we count the number of non-deleted elements that precede offset e_0 using $\overline{K}(v_r)$ and some $D_j(v_r)$.

$\mathsf{rank}_a(S, i)$**:** We find the offset e_0 of the i-th non-deleted element in $\overline{L}(v_r)$. Let v_k, t_k be defined as in Section 4. In every node v_k, we find the last offset $e'_k \leq e_k$ such that $\overline{B}(v_k)[e'_k].\mathrm{chr} = t_k$. Note that this element may be a deleted one, but it still drives us to the correct location in $\overline{L}(v_{k+1})$. We proceed exactly as in Section 4 until we arrive at a leaf $v_h = v_a$. Finally, we count the number of non-deleted elements preceding offset e_h using $\overline{K}(v_a)$ and some $D_j(v_a)$.

5.2. Updates. Insertions are performed just as in Section 4, except that we also update the data structure $D_j(v_k)$ when an element $\overline{B}(v_k)[e_k]$ that encodes the inserted symbol a is added to a block $G_j(v_k)$. When $S[i]$ is deleted, we append it to the list *DEL*. Then we visit each block $G_j(v_k)$ containing the element $\overline{B}(v_k)[e_k]$ that encodes $S[i]$ and update the data structures $D_j(v_k)$. Finally, $\overline{K}(v_r)$ and $\overline{K}(v_a)$ are also updated. This takes in total $O(\lg_\rho \sigma + \lg n/\lg\lg n) = O((\lg\sigma + \lg n)/\lg\lg n)$ time.

When the number of symbols in the list *DEL* reaches $n/\lg^2 n$, we perform a procedure to effectively delete all of its elements. Therefore *DEL* never requires more than $O(n/\lg n)$ bits, and the space overhead due to storing deleted symbols is $O(n\lg\sigma/\lg^2 n)$ bits.

Let $\overline{B}(v_k)[e_k]$, $0 \leq k \leq h$, be the elements that encode a symbol $\overline{S}[i] \in DEL$. The method for tracking the elements $\overline{B}(v_k)[e_k]$, removing them from their blocks $G_j(v_k)$, and updating the block structures, is symmetric to the insertion procedure described in Section 4. In this case we do not need the predecessor queries to track the elements to delete, as the procedure is similar to that for accessing $S[i]$. When the size of a block $G_j(v_k)$ falls below $(\lg^3 n)/(2\lg\rho)$ and it is not the last block of $\overline{L}(v_k)$, we merge it with $G_{j+1}(v_k)$, and then split the result if its size exceeds $2\lg^3 n/\lg\rho$. This retains $O(1)$ amortized time per deletion in any node v_k, including the updates to $\overline{K}(v_k)$ structures, and adds up to $O((\lg\sigma+\lg n)/\lg\lg n)$ amortized time per deleted symbol.

Once all the pointers in *DEL* are processed, we rebuild from scratch the structures $P(v)$ for all nodes v. The total size of all the $P(v)$ structures is $O(\rho n\lg\sigma/\lg^3 n)$. Since a data structure for incremental split-find is constructed in linear time [33], all the $P(v)$ structures are rebuilt in $O(n\lg\sigma/\lg^{3-\varepsilon} n)$ time. Hence the amortized time to rebuild the $P(v)$s is $O(\lg\sigma/\lg^{1-\varepsilon} n)$, which does not affect the amortized time $O((\lg\sigma+\lg n)/\lg\lg n)$ to carry out the effective deletions.

6. Data Structures for Handling Blocks. We describe the way the data is stored in blocks $G_j(v)$, as well as the way the various structures inside blocks operate. All the data structures are based on the same idea: We maintain a tree with node degree $\lg^\delta n$ and leaves that contain $O(\lg n)$ bits. Since elements within a block can be addressed with $O(\lg\lg n)$ bits, each internal node and each leaf fits into $O(1)$ machine words. As a result, we can support searching and basic operations in each node in constant time.

6.1. Data Organization. The block data is physically stored as a sequence of *miniblocks* of $\lg_\rho n$ to $2\lg_\rho n$ symbols, using $\Theta(\lg n)$ bits. Thus there are $O(|G_j(v)|/\lg_\rho n) = O(\lg^2 n)$ miniblocks in a block. These miniblocks will be the leaves of a B-tree T with internal nodes of arity τ to 2τ, for $\tau = \Theta(\lg^\delta n)$ and some constant $0 < \delta < 1$. The height of this tree is constant, $O(1/\delta)$. Each node of T stores $\Theta(\tau)$ counters telling the number of symbols stored at the leaves that descend from each child. This requires

just $O(\tau \lg\lg n) = o(\lg n)$ bits. To access any position of $G_j(v)$, we descend in T, using the counters to determine the correct child. When we arrive at a leaf, we know the local offset of the desired symbol within the leaf, and can access it directly. Since the counters fit in less than a machine word, a small global precomputed table gives the correct child in constant time, for any possible sequence of up to 2τ counters and any possible desired offset. The table has $2^{2\tau \lg(2 \lg_\rho n)} \cdot 2 \lg_\rho n$ entries, which is $o(n^\alpha)$ for any constant $\alpha > 0$. Therefore, we have $O(1)$ time access to any symbol.

Upon updates in $G_j(v)$, we arrive at the correct leaf of its B-tree T, insert or delete the symbol (in constant time because the leaf contains $O(\lg n)$ bits), and update the counters in the path from the root (in constant time as they have $o(\lg n)$ bits, with the help of another global precomputed table). Splits/merges upon overflows/underflows of leaves are handled as usual, and can be solved in $O(1/\delta)$ constant-time operations (again, with the help of global precomputed tables to update the counters upon splits and merges).

The space overhead of the internal nodes of T is $O(|G_j(v)| \lg \rho \lg\lg n/\lg n)$ bits, as there are $O((|G_j(v)|/\lg_\rho n)/\tau)$ internal nodes and each one uses $O(\tau \lg\lg n)$ bits for its counters.

We consider now the space used by the data itself, that is, the string $G_j(v)$.str. In order not to waste space, the miniblock leaves are stored using a simple memory management structure.

LEMMA 6.1 ([45]). *A memory area storing s bits in total, handling chunks of sizes up to b bits, can be managed so that chunks can be allocated, freed, and accessed in $O(b/\lg n)$ worst-case time, within a total space of $s + O((s/b)\lg s + b^2 + b \lg s)$ bits. This space forms a contiguous area that grows or shrinks by multiples of some fixed value in $\Theta(b)$.*

For our case, where $s = |G_j(v)| \lg \rho = O(\lg^3 n)$ and $b = \Theta(\lg n)$ is a memory area of polylogarithmic size, the lemma allows us to allocate, free, and access miniblocks in constant time, while using pointers of $O(\lg\lg n)$ bits and wasting $O(|G_j(v)| \lg \rho \lg\lg n/\lg^2 n + \lg^2 n + \lg n \lg\lg n)$ bits. Added up over all the wavelet tree, the first term adds up to $O(n \lg\sigma \lg\lg n/\lg^2 n)$ bits, whereas the rest adds up to $O(n \lg\sigma/\lg n + \sigma \lg^2 n)$ bits. In order to reduce the last term to $O(\sigma \lg n)$, the first blocks $G_1(v)$ of all the σ wavelet tree lists $L(v)$ are grouped into sets of size $\Theta(\lg n)$ and their memory areas are managed together. Their combined address spaces are of size $O(\lg^4 n)$, which does not affect the result. This grouping is fixed and does not change upon updates.

The allocation structure of Lemma 6.1 uses a memory area of fixed-size cells that grows or shrinks at the end as miniblocks are created or destroyed. Such structure is called an *extendible array (EA)* [54], and the problem is how to handle a *collection* of EAs. In our case, we must handle a set of $O(n \lg\sigma/\lg^3 n + \sigma/\lg n)$ EAs. A collection of EAs must support accessing any cell of any EA, letting any EA grow or shrink by one cell, and create and destroy EAs. The following lemma, which assumes words of $\lg n$ bits, is useful.

LEMMA 6.2 ([54, Lem. 1], simplified). *A collection of a EAs of total size s bits can be represented using $s + O(a \lg n + \sqrt{sa \lg n})$ bits of space, so that the operations of creation of an empty EA and access take constant worst-case time, whereas grow/shrink take constant amortized time. An EA of s' bits can be destroyed in time $O(s'/\lg n)$.*

In our case $a = O(n \lg\sigma/\lg^3 n + \sigma/\lg n)$ and $s = O(n \lg\sigma)$, so the space overhead posed by the EAs is $O(n \lg\sigma/\lg^2 n + \sigma + n \lg\sigma/\lg n + \sqrt{n\sigma \lg\sigma}) = O(n \lg\sigma/\lg n + \sigma \lg n)$.

When we store the miniblocks in compressed form, in Section 8, they could use as little as $O(\lg^{\varepsilon} n \lg\lg n)$ bits, and thus we could store up to $\Theta(\lg^{1-\varepsilon} n/\lg\lg n)$ miniblocks in a single leaf of T. This can still can be handled in constant time using (more complicated) global precomputed tables [39], and the counters and pointers of $O(\lg\lg n)$ bits are still large enough.

Summing up, the total space overhead of the structure is $O(n \lg\sigma (\lg\lg n)^2/\lg n + \sigma \lg n)$ bits.

6.2. Structure $R_j(v)$. To support rank and select we enrich T with further information per node. We store ρ counters with the number of occurrences of each symbol in the subtree of each child. The node size becomes $O(\tau\rho \lg\lg n) = O(\lg^{\varepsilon+\delta} n \lg\lg n) = o(\lg n)$ as long as $\varepsilon + \delta < 1$. This adds up to $O(|G_j(v)|\rho \lg\rho \lg\lg n/\lg n)$ bits because the leaves of T handle $\Theta(\tau)$ miniblocks. Added over the whole wavelet tree, this is $O(n \lg\sigma \lg\lg n/\lg^{1-\varepsilon} n)$ bits.

With this information on the nodes we can easily solve rank and select in constant time, by descending on T and determining the correct child (and accumulating data about the leftward children) in $O(1)$ time using global precomputed tables. Nodes can also be updated in constant time even upon splits and merges, since all the counters can be recomputed in $O(1)$ time with the help, again, of global precomputed tables.

6.3. Structure $F_j(v)$. This structure stores all the inter-node pointers leaving from block $G_j(v)$, to its parent and to any of the ρ children of node v.

The structure is a tree T_f very similar in spirit to T. The pointers are stored at the leaves of T_f, in increasing order of their source index inside $G_j(v)$. The pointers stored are inter-node, and thus require $\Theta(\lg n)$ bits. Thus we store a constant number of pointers per leaf of T_f. For each pointer we store the local index in $G_j(v)$ holding the pointer, and the target location, formed by a system-wide pointer to a block $G_\ell(u)$ plus an identifier of the local index within $G_\ell(u)$ (see Section 6.4). The internal nodes x, of arity $\Theta(\tau)$, maintain information on the number of indexes of $G_j(v)$ covered by each child of x, and the number of pointers of each kind stored in the subtree of each child of x ($1 + \rho$ counters, for the parent of v and for the t-th wavelet tree child of v, for each $0 \leq t < rho$). This requires $O(\tau\rho \lg\lg n) = o(\lg n)$ bits, as before. To find the last local index up to i holding a pointer of a certain kind, we traverse T_f from the root looking for index i. At each node x, it might be that the child y where we have to enter holds pointers of that kind, or not. If it does, then we first enter into child y. If we return with an answer, we recursively return it. If we return with no answer, or there are no pointers of the desired kind below y, we enter into the last sibling to the left of y that holds a pointer of the desired kind, and switch to a different mode where we simply go down the tree looking for the rightmost child with a pointer of the desired kind. It is not hard to see that this procedure visits $O(1/\delta)$ nodes, and thus it is constant-time because all the computations inside nodes can be done in $O(1)$ time with global precomputed tables. When we arrive at the leaf, we scan for the desired pointer in constant time.

The tree T_f must be updated when a symbol t is inserted before any other occurrence of t in $G_j(v)$, when a symbol is inserted at the first position of $G_j(v)$ and, similarly, when symbols are deleted from $G_j(v)$. The needed queries are easily answered with tree T (enriched for rank/select queries). Moreover, due to the bidirectionality, we must also update T_f when pointers to $G_j(v)$ are created from the parent or a child of v, or when they are deleted. Those updates work just like on the tree T. T_f is also updated upon insertions and deletions of symbols, even if they do not change pointers, to maintain the positions up to date. In this case we traverse T_f looking for

the position of the update, change the local indexes stored at the leaf, and update the subtree sizes stored at the internal nodes of T_f. The total space overhead is as of $R_j(v)$.

6.4. Structure $H_j(v)$. This structure manages the inter-node pointers that point inside $G_j(v)$. As explained in Section 4.4, we give a handle to the outside nodes, that does not change over time, and $H_j(v)$ translates handles to local indexes in $G_j(v)$.

We store a tree T_h that is just like T_f, where the incoming pointers are stored. T_h is simpler, however, because at each node we only need to store the number of indexes covered by the subtree of each child. It must also be possible to traverse T_h from a leaf to the root.

In addition, we manage a table Tbl so that $Tbl[id]$ points to the leaf of T_h where the pointer corresponding to handle id is stored. Tbl is also managed as a tree similar to T_f, with pointers sorted by id, where a constant number of identifiers id are stored at the leaves together with their pointers to the leaves of T_h (note that there are $O(\lg^3 n/\lg\rho)$ identifiers at most, so we need $O(\lg\lg n)$ bits for the identifiers and their pointers to T_h). Each internal node in Tbl maintains the maximum identifier, and the number of identifiers, stored at its leaves. Thus one can, in constant time, find the pointer to T_h corresponding to a given id, and also find the smallest unused identifer when a fresh one is needed (by looking for the first leaf of Tbl where the maximum identifier is larger than the number of identifiers up to that leaf).

At the leaves of T_h we store, for each pointer, a backpointer to the corresponding leaf of Tbl and the local index in $G_j(v)$. Given a handle id, we find using Tbl the corresponding place in the leaf of T_h, and move upwards up to the root of T_h, adding to the leaf index the number of indexes covered by the leftward children of each node. At the end we obtain the local index of id.

When pointers to $G_j(v)$ are created or destroyed, we insert or remove pointers in T_h. Insertion requires traversing T_h top-down to find the leaf covering the desired index, and then creating a fresh entry $Tbl[id]$ pointing to it (so that id will be the external handle associated to the inserted pointer). Deletion of id requires going to the leaf of T_h given by $Tbl[id]$, removing the pointer from the leaf, and freeing id from Tbl. Splits and merges of leaves of T_h require moving a constant number of pointers, updating their pointers from Tbl (which are found using the backpointers), and updating the counters towards the root of T_h. Similarly, the splits and merges in Tbl require updating the backpointers from T_h, which are found using the pointers from Tbl to T_h. We must also update T_h upon symbol insertions and deletions in $G_j(v)$, to maintain the indexes up to date.

Tbl and T_h may contain up to $\Theta(\lg^3 n/\lg\rho)$ pointers of $O(\lg\lg n)$ bits, which can be significant for some blocks. However, across the whole structure there can be only $O(\rho n\lg\sigma/\lg^3 n)$ pointers, adding up to $s = O(\rho n\lg\sigma\lg\lg n/\lg^3 n)$ bits, spread across $a = O(n\lg\sigma/\lg^3 n)$ structures Tbl and T_h. Using again Lemma 6.2, a collection of EAs poses an overhead of $O(n\lg\sigma/\lg^2 n)$ bits. This includes the space overhead of the values stored at internal nodes.

6.5. Structure $D_j(v)$ and the Final Result. Structure $D_j(v)$ is implemented as a tree T_d analogous to T, storing at each node the number of elements and the number of non-deleted elements below each child. It takes $O(|G_j(v)|\lg\rho\lg\lg n/\lg n)$ bits. Since these are stored only for the root v_r and the leaves v_a of $\mathcal{T}$, its space adds up to $O(n\lg\rho\lg\lg n/\lg n) = O(n(\lg\lg n)^2/\lg n)$ bits.

Let us now consider the total space for all the data structures. While the raw data adds up to $n \lg \sigma$ bits, the extra space is dominated by structures $R_j(v)$ and $F_j(v)$, adding up to $O(n \lg \sigma \lg \lg n / \lg^{1-\delta-\varepsilon} n)$ bits, plus $O(\sigma \lg n)$ for the memory management overhead. We can use, say, $\delta = \varepsilon$ and then have $O(1/\varepsilon)$ time and $O(n \lg \sigma \lg \lg n / \lg^{1-\varepsilon} n + \sigma \lg n)$ bits for any $0 < \varepsilon < 1$ (renaming 2ε as ε). By further increasing ε infinitesimally, we can write the space overhead as $O(n \lg \sigma / \lg^{1-\varepsilon} n + \sigma \lg n)$ bits for any $0 < \varepsilon < 1$.

7. Changes in $\lg n$ and Alphabet Independence. Note that our structures depend on the value of $\lg n$, so they should be rebuilt when $\lceil \lg n \rceil$ changes. We use $w = \lceil \lg n \rceil$ as a fixed value and rebuild the structure from scratch when n reaches another power of two (more precisely, we use words of $w = \lceil \lg n \rceil$ bits until $\lceil \lg n \rceil$ increases by 1 or decreases by 2, and only then update w and rebuild). These reconstructions do not affect the amortized complexities, and the slightly larger words waste an $O(1/\lg n)$ extra space factor in the redundancy.

We take advantage of using a fixed w value to get rid of the alphabet dependence. If $\lg \sigma \leq w$, our time complexities are the optimal $O(\lg n / \lg \lg n)$. However, if σ is larger, this means that not all the alphabet symbols can appear in the current sequence (which contains at most $n \leq 2^w < \sigma$ distinct symbols). Therefore, in this case we create the wavelet tree for an alphabet of size $s = 2^w$, not σ (this wavelet tree is created when w changes). We also set up a *mapping* array $SN[1, \sigma]$ that will tell to which value in $[1..s]$ is a symbol mapped, and a *reverse mapping* $NS[1, s]$ that tells to which original symbol in $[1..\sigma]$ does a mapped symbol correspond. Both SN and NS are initialized in constant time [42, Section III.8.1] and require $O(\sigma \lg n + n \lg \sigma)$ bits of space. Since this is used only when $\sigma > n$, the space is $O(\sigma \lg n)$.

Upon operations $\mathsf{rank}_a(S, i)$ and $\mathsf{select}_a(S, j)$, the symbol a is mapped using SN (the answer is obvious if a does not appear in SN) in constant time. The answer of operation $\mathsf{access}(S, i)$ is mapped using NS in constant time as well. Upon insertion of a, we also map a using SN. If not present in SN, we find a free slot $NS[i]$ (we maintain a list of free slots) and assign $NS[i] = a$ and $SN[a] = i$. When the last occurrence of a symbol a is deleted (we maintain global counters to determine this in constant time) we return its slot to the free list and uninitialize its entry in SN. In this way, when $\lg \sigma > \lg n$, we can support all the operations in time $O(\lg s / \lg \lg s) = O(\lg n / \lg \lg n)$.

We are ready to state a first version of our result, not yet compressing the sequence. In Section 6 it was shown that the time for the operations is $O(1/\varepsilon)$. Since the height of the wavelet tree is $\lg_\rho \min(\sigma, s) = O(\frac{1}{\varepsilon} \lg n / \lg \lg n)$, then we have $O(\frac{1}{\varepsilon^2} \lg n / \lg \lg n)$ time for all the operations. Considering the total space overhead obtained in Section 6, which also dominates previous ones like the space for the structures $L(v)$, $K(v)$, $P(v)$ and *DEL* (see Table 4.1), we obtain the following result. (Note that when $\sigma > n$ we use an alphabet of size $O(n)$, but then still we need the SN mapping, that takes $O(\sigma \lg n)$ bits.)

THEOREM 7.1. *A dynamic string $S[1, n]$ over alphabet $[1..\sigma]$ can be stored in a structure using $n \lg \sigma + O(n \lg \sigma / \lg^{1-\varepsilon} n + \sigma \lg n)$ bits, for any $0 < \varepsilon < 1$, and supporting queries* access, rank *and* select *in time $O(\frac{1}{\varepsilon^2} \lg n / \lg \lg n)$. Insertions and deletions of symbols are supported in $O(\frac{1}{\varepsilon^2} \lg n / \lg \lg n)$ amortized time.*

8. Compressed Space. Now we compress the space of the data structure to zero-order entropy ($nH_0(S)$ plus redundancy). We show how a different encoding of the bits within the blocks reduces the $n \lg \sigma$ to $nH_0(S)$ in the space without affecting the time complexities.

Raman et al. [53] describe an encoding for a bit vector $B[1,n]$ that occupies $nH_0(B) + O(n \lg\lg n/\lg n)$ bits of space. It consists of cutting the bit vector into chunks of length $b = (\lg n)/2$ and encoding each chunk i as a pair (c_i, o_i): c_i is the *class*, which indicates how many 1s are there in the chunk, and o_i is the *offset*, which is the index of this particular chunk within its class. The c_i components add up to $O(n \lg\lg n/\lg n)$ bits, whereas the o_i components add up to $nH_0(B)$. Navarro and Sadakane [49, Sec. 8] describe a technique to maintain a dynamic bit vector in this format. They allow the chunk length b to vary, so they encode triples (b_i, c_i, o_i) maintaining the invariant that $b_i + b_{i+1} > b$ for any i. They show that this retains the same space, and that each update affects $O(1)$ chunks.

We extend this encoding to handle an alphabet $[1..\rho]$ [20], so that $b = (\lg_\rho n)/2$ symbols, and each chunk is encoded as a tuple $(b_i, c_i^1, \ldots, c_i^\rho, o_i)$ where c_i^t counts the occurrences of t in the block. The classes $(b_i, c_i^1, \ldots, c_i^\rho)$ use $O(\rho n \lg\lg n/\lg n)$ bits, and the offsets still add up to $nH_0(B)$. Blocks are encoded/decoded in $O(1)$ time, as the class takes $O(\rho \lg\lg n) = o(\lg n)$ bits and the block encoding requires at most $O(\lg n)$ bits. At the end of Section 6.1 we show that the plain representation of the data can be changed by a compressed one without affecting the operations inside blocks, thanks to the fact that a tuple $(b_i, c_i^1, \ldots, c_i^\rho, o_i)$ uses at least $\Theta(\lg^\varepsilon n \lg\lg n)$ bits, so the number of miniblocks that fit in a single leaf of T is still polylogarithmic, and then the counters in internal nodes of T still require $O(\lg\lg n)$ bits.

The sum of the local entropies of the chunks, across the whole $L(v)$, adds up to $nH_0(B_v)$, and these add up to $nH_0(S)$ [26]. The redundancy over the entropy is $O(\rho \lg\lg n)$ bits per miniblock, adding up to $O(nH_0(S) \lg\lg n/\lg^{1-\varepsilon} n)$ bits. The fact that we store $\overline{S}$ instead of S, with up to $O(n/\lg^2 n)$ spurious symbols, can increase $nH_0(S)$ up to $nH_0(\overline{S}) \leq nH_0(S) + O(n/\lg n)$ bits. Adjusting ε infinitesimally, all the space overhead on top of $nH_0(S)$ is dominated by the $O(n \lg\sigma/\lg^{1-\varepsilon} n + \sigma \lg n)$ bits of space overhead obtained in Section 6.[3] Thus we get the following result, for any desired $0 < \varepsilon < 1$.

THEOREM 8.1. *A dynamic string $S[1,n]$ over alphabet $[1..\sigma]$ and with zero-order empirical entropy $H_0(S)$ can be stored in a structure using $nH_0(S) + O(n \lg\sigma/\lg^{1-\varepsilon} n + \sigma \lg n)$ bits, for any $0 < \varepsilon < 1$, and supporting queries* access, rank *and* select *in time $O(\frac{1}{\varepsilon^2} \lg n/\lg\lg n)$. Insertions and deletions of symbols are supported in $O(\frac{1}{\varepsilon^2} \lg n/\lg\lg n)$ amortized time.*

9. Worst-Case Complexities. While in previous sections we have obtained optimal time and compressed space, the time for the update operations is amortized. In this section we derive worst-case time complexities, at the price of losing the time optimality, which will now become logarithmic for some operations. Along the rest of the section we remove the various sources of amortization in our solution.

9.1. Block Splits and Merges. Our amortized solution splits overflowing blocks and rebuilds the two new blocks from scratch (Section 4.4). Similarly, it merges underflowing blocks (as a part of the cleaning of the global *DEL* list in Section 5.2). This gives good amortized times but in the worst case the cost is $\Omega(\lg^3 n/\lg\lg n)$.

We use a technique [25] that avoids global rebuildings. A block is called *dense* if it contains at least $\lg^3 n$ bits, and *sparse* otherwise. While sparse blocks of any size (larger than zero) are allowed, we maintain the invariant that no two consecutive

[3] Actually, some of this space overhead is also reduced to a function of $H_0(S)$ instead of $\lg\sigma$, but there are anyway other additive terms that are insensitive to the compression, which make this path less attractive to pursue.

sparse blocks may exist. This retains the fact that there are $O(n \lg \sigma / \lg^3 n + \sigma)$ blocks in the data structure. The maximum size of a block will be $2 \lg^3 n$ bits. When a block overflows due to an insertion, we move its last element to the beginning of the next block. If the next block would also overflow, then we are entitled to create a new sparse block between both dense blocks, containing only that element. Analogously, when a deletion converts a dense block into sparse (i.e., it falls below length $\lg^3 n$ bits), we check if both neighbors are dense. If they are, the current block can become sparse too. If, instead, there is a sparse neighbor, we move its first/last element into the current block to avoid it becoming sparse. If this makes that sparse neighbor block become of size zero, we remove it.

Therefore, we only create and destroy empty blocks, and move a constant number of elements to neighboring blocks. This can be done in constant worst-case time. It also simplifies the operations on the partial-sum data structures $K(v)$, since now only updates by ± 1 and insertions/deletions of elements with value zero are necessary, and these are carried out in $O(\lg n / \lg \lg n)$ worst case time (Lemma 4.1). Recall that $\lg n$ is fixed in each instance of our data structure, so the definition of sparse and dense is static.

9.2. Split-Find Data Structure and Lazy Deletions. The split-find data structure [33] we used in Section 5 to implement the P_t structures has constant amortized insertion time. We replace it by another one [44, Thm 4.1] achieving $O(\lg \lg n)$ worst-case time. Their structure handles a list of colored elements (list nodes), where each element can have $O(1)$ colors (each color is a positive integer bounded by $O(\lg^\varepsilon n)$ for a constant $0 < \varepsilon < 1$). We will only use list nodes with 0 or 1 color. The operations of interest to us are: creating a new list node without colors, assigning or removing a color to/from a list node, and finding the last list node preceding a given node and having some given color. Node deletions are not supported. The number of list nodes must be smaller than a certain upper bound n', and the operations cost $O(\lg \lg n')$. In our case, since $\lg n$ is fixed, we can use $n' = 2^w = O(n)$ as the upper bound.

We use ρ colors, one per symbol in the sequences. Each time we create a block, we add a new uncolored node to the list, with a bidirectional pointer to the block. Each time we insert a symbol $t \in [1..\rho]$ for the first time in a block, we add a new node colored t to the list, right after the uncolored element that represents the block, and also set a bidirectional pointer between this node and the block.

We cannot use the lazy deletions mechanism of Section 5, as it gives only good amortized complexity. We carry out the deletions immediately in the blocks, as said in Section 9.1. Each time the last occurrence of a symbol $t \in [1..\rho]$ is deleted from a block, we remove the color from the corresponding list node (if the symbol reappears later, we reuse the same node and color it, instead of creating a new one).

Therefore, finding the last block where a symbol t appears, as needed by the rank query and for insertions, corresponds to finding the last list node colored t and preceding the uncolored node that represents the current block.

Since list nodes cannot be deleted, when a block disappears its (uncolored) list nodes are left without an associated block. This does not alter the result of queries, but there is the risk of maintaining too many useless nodes. We permanently run an incremental list "copying" process, traversing the current list of blocks and inserting the corresponding nodes into a new list. This new list is also updated, together with the current list, on operations concerning the blocks already copied. When the new list is ready it becomes the current list and the previous list is incrementally deleted. In $O(n\rho \lg \rho / \lg^3 n)$ steps we have copied the current list; by this time the number of

useless nodes is at most $O(n\rho \lg \rho / \lg^3 n)$ and just poses $O(n \lg\lg n / \lg^{2-\varepsilon} n)$ bits of space overhead.

Note that blocks must manage the sets of up to ρ pointers to their colored nodes. This is easily handled in constant time with the same techniques used for structure $F_j(v)$ in Section 6.

Since the colored list data structure requires $O(\lg\lg n)$ time, operations rank and insert take worst-case time $O(\frac{1}{\varepsilon}\lg n)$, whereas access, select and delete still stay in $O(\frac{1}{\varepsilon^2}\lg n/\lg\lg n)$.

9.3. Changes in $\lg n$. As an alternative to reconstructing the whole structure when n doubles or halves, Mäkinen and Navarro [39] describe a way to handle this problem without affecting the space nor the time complexities, in a worst-case scenario. The sequence is cut into a prefix, a middle part, and a suffix. The middle part uses a fixed value $\lceil \lg n \rceil$, the prefix uses $\lceil \lg n \rceil - 1$ and the suffix uses $\lceil \lg n \rceil + 1$. Insertions and deletions trigger slight expansions and contractions in this separation, so that when n doubles all the sequence is in the suffix part, and when n halves all the sequence is in the prefix part, and we smoothly move to a new value of $\lg n$. This means that the value of $\lceil \lg n \rceil$ is fixed for any instance of our data structure. Operations access, rank and select, as well as insert and delete, are easily adapted to handle this split string.

Actually, to have sufficient time to build global precomputed tables of size $O(n^\alpha)$ for $0 < \alpha < 1$, the solution [39] maintains the sequence split into five, not three, parts. This gives also sufficient time to build any global precomputed table we need to handle block operations in constant time, as well as to build the wavelet tree structures of the new partitions.

9.4. Memory Management Inside Blocks. The extendible arrays (EAs) of Lemma 6.2 (Section 6) have amortized times to grow and shrink. Converting those to worst-case time requires a constant space overhead factor. While this is acceptable for the EAs of structures Tbl and T_h in Section 6, they raise the overall space to $O(nH_0(S))$ bits if used to maintain the main data. Instead, we get rid completely of the EA mechanism to maintain the data, and use a single large memory area for all the miniblocks of Section 6, using Lemma 6.1.

The problem of using a single memory area is that the pointers to the miniblocks require $\Theta(\lg n)$ bits, which is excessive because miniblocks are also of $\Theta(\lg n)$ bits. Instead, we use slightly larger miniblocks, of $\Theta(\lg n \lg\lg n)$ bits. This makes the overhead due to pointers to miniblocks $O(|G_j(v)|/\lg\lg n)$, adding up to additional $O(nH_0(S)/\lg\lg n + n\lg\sigma/\lg^{1-\varepsilon} n) = o(n\lg\sigma)$ bits.

The price of using larger miniblocks is that now the operations on blocks are not anymore constant time because they need to traverse a miniblock, which takes time $O(\lg\lg n)$. We can still retain constant time for the query operations, by considering *logical* miniblocks of $\Theta(\lg n)$ bits, which are stored in *physical* areas of $\Theta(\lg\lg n)$ miniblocks. However, update operations like insert and delete must shift all the data in the miniblock area and possibly relocate it in the memory manager, plus updating pointers to all the logical miniblocks displaced or relocated. This costs $O(\lg\lg n)$ time per insertion and deletion. This completes our result.

THEOREM 9.1. *A dynamic string $S[1,n]$ over alphabet $[1..\sigma]$ can be stored in a structure using $nH_0(S) + O(nH_0(S)/\lg\lg n + n\lg\sigma/\lg^{1-\varepsilon} n + \sigma\lg n) = nH_0(S) + o(n\lg\sigma) + O(\sigma\lg n)$ bits, for any constant $0 < \varepsilon < 1$, and supporting queries* access *and* select *in worst-case time $O(\frac{1}{\varepsilon^2}\lg n/\lg\lg n)$, and query* rank*, insertions and deletions in worst-case time $O(\frac{1}{\varepsilon}\lg n)$.*

10. Extensions and Applications. We first describe two direct applications of our result in important open problems. Then we extend our results to handling general alphabets, and describe various applications of this extension.

10.1. Dynamic Sequence Collections. A landmark application of dynamic sequences, stressed out in several papers along time [14, 38, 13, 38, 36, 39, 24, 37, 25, 29, 49], is to maintain a collection $\mathcal{C}$ of texts, where one can carry out indexed pattern matching, as well as inserting and deleting texts from the collection. Plugging in our new representation we can significantly improve the time and space of previous work, with an amortized and with a worst-case update time, respectively.

THEOREM 10.1. *There exists a data structure for handling a collection $\mathcal{C}$ of texts over an alphabet $[1..\sigma]$ within size $nH_h(\mathcal{C}) + o(n \lg \sigma) + O(\sigma^{h+1} \lg n + m \lg n)$ bits, simultaneously for all h. Here n is the length of the concatenation of m texts, $\mathcal{C} = T_1 \circ T_2 \cdots \circ T_m$, and we assume that the alphabet size is $\sigma = o(n)$. The structure supports counting of the occurrences of a pattern P in $O(|P| \lg n / \lg \lg n)$ time. After counting, any occurrence can be located in time $O(\lg_\sigma n \lg n)$. Any substring of length ℓ from any T in the collection can be displayed in time $O((\ell / \lg \lg n + \lg_\sigma n) \lg n)$. Inserting or deleting a text T takes $O(\lg n + |T| \lg n / \lg \lg n)$ amortized time. For $0 \leq h \leq (\alpha \lg_\sigma n) - 1$, for any constant $0 < \alpha < 1$, the space simplifies to $nH_h(\mathcal{C}) + o(n \lg \sigma) + O(m \lg n)$ bits.*

THEOREM 10.2. *There exists a data structure for handling a collection $\mathcal{C}$ of texts over an alphabet $[1..\sigma]$ within size $nH_h(\mathcal{C}) + o(n \lg \sigma) + O(\sigma^{h+1} \lg n + m \lg n)$ bits, simultaneously for all h. Here n is the length of the concatenation of m texts, $\mathcal{C} = T_1 \circ T_2 \cdots \circ T_m$, and we assume that the alphabet size is $\sigma = o(n)$. The structure supports counting of the occurrences of a pattern P in $O(|P| \lg n)$ time. After counting, any occurrence can be located in time $O(\lg_\sigma n \lg n \lg \lg n)$. Any substring of length ℓ from any T in the collection can be displayed in time $O((\ell + \lg_\sigma n \lg \lg n) \lg n)$. Inserting or deleting a text T takes $O(|T| \lg n)$ time. For $0 \leq h \leq (\alpha \lg_\sigma n) - 1$, for any constant $0 < \alpha < 1$, the space simplifies to $nH_h(\mathcal{C}) + o(n \lg \sigma) + O(m \lg n)$ bits.*

The theorems refer to $H_h(\mathcal{C})$, the h-th order empirical entropy of sequence $\mathcal{C}$ [41]. This is a lower bound to any semistatic statistical compressor that encodes each symbol as a function of the h preceding symbols in the sequence, and it holds $H_h(\mathcal{C}) \leq H_{h-1}(\mathcal{C}) \leq H_0(\mathcal{C}) \leq \lg \sigma$ for any $h > 0$. To offer search capabilities, the Burrows-Wheeler Transform (BWT) [12] of $\mathcal{C}$, $\mathcal{C}^{bwt}$, is represented, not $\mathcal{C}$; then access and rank operations on $\mathcal{C}^{bwt}$ are used to support pattern searches and text extractions. Kärkkäinen and Puglisi [35] showed that, if $\mathcal{C}^{bwt}$ is split into superblocks of size $\Theta(\sigma \lg^2 n)$, and a zero-order compressed representation is used for each superblock, the total number of bits is $nH_h(\mathcal{C}) + o(n)$.

We use their partitioning, and Theorems 8.1 or 9.1 to represent each superblock. For Theorem 10.1, the superblock sizes are easily maintained upon insertions and deletions of symbols, by splitting and merging superblocks and rebuilding the structures involved, without affecting the amortized time per operation. They [35] also need to manage a table storing the rank of each symbol up to the beginning of each superblock. This is arranged, in the dynamic scenario, with σ partial sum data structures containing $O(n/(\sigma \lg^2 n))$ elements each, plus another one storing the superblock lengths. This adds $O(n / \lg n)$ bits and $O(\lg n / \lg \lg n)$ time per operation (Lemma 4.1). Upon block splits and merges, we use the same techniques used for K structures described in Section 4.4.

For Theorem 10.2 we use the smooth block size management algorithm described in Section 9.1 for the superblocks, which guarantees worst-case times and the same

space redundancy. Then partial-sum data structures are used without problems.

Finally, the locating and displaying overheads are obtained by marking one element out of $\lg_\sigma n \lg\lg n$, so that the space overhead of $o(n \lg \sigma)$ is maintained. Other simpler data structures used in previous work [39], such as mappings from document identifiers to their position in $\mathcal{C}^{bwt}$ and the samplings of the suffix array, can easily be replaced by $O(\lg n / \lg\lg n)$ time partial-sums data structures and simpler structures to maintain dictionaries of values.

10.2. Burrows-Wheeler Transform. Another application of dynamic sequences is to build the BWT of a text T, T^{bwt}, within compressed space, by starting from an empty sequence and inserting each new character, $T[n]$, $T[n-1]$, ..., $T[1]$, at the proper positions. Equivalently, this corresponds to initializing an empty collection and then inserting a single text T using Theorem 10.1. The result is also stated as the compressed construction of a static FM-index [20], a compressed index that consists essentially of a (static) wavelet tree of T^{bwt}. Our new representation improves upon the best previous result on compressed space [49].

THEOREM 10.3. *The Alphabet-Friendly FM-index [20], as well as the BWT [12], of a text $T[1,n]$ over an alphabet of size σ, can be built using $nH_h(T) + o(n \lg \sigma)$ bits, simultaneously for all $1 \leq h \leq (\alpha \lg_\sigma n) - 1$ and any constant $0 < \alpha < 1$, in time $O(n \lg n / \lg\lg n)$. It can also be built within the same time and $nH_0(T) + o(n \lg \sigma) + O(\sigma \lg n)$ bits, for any alphabet size σ.*

We are using Theorem 10.1 for the case $h > 0$, and Theorem 8.1 to obtain a less alphabet-restrictive result for $h = 0$ (in this case, we do not split the text into superblocks of $O(\sigma \lg^2 n)$ symbols, but just use a single sequence). Note that, although insertion times are amortized in those theorems, this result is worst-case because we compute the sum of all the insertion times.

This is the first time that $o(n \lg n)$ time complexity is obtained within compressed space. Other space-conscious results that achieve better time complexity (but more space) are Okanohara and Sadakane [51], who achieved optimal $O(n)$ time within $O(n \lg \sigma \lg\lg_\sigma n)$ bits, and Hon et al. [31], who achieved $O(n \lg\lg \sigma)$ time and $O(n \lg \sigma)$ bits. On the other hand, Kärkkäinen [34] may obtain less space but more time: $O(n \log n + nv)$ time and $O(n \log n / \sqrt{v})$ bits on top of the raw data, for any parameter v.

10.3. Handling General Alphabets. Our time results do not depend on the alphabet size σ, yet our space does, in a way that ensures that σ does not interfere with the results as long as $\sigma = o(n)$ (so $\sigma \lg n = o(n \lg \sigma)$).

Let us now consider the case where the alphabet Σ is much larger than the *effective* alphabet of the string, that is, the set of symbols that actually appear in S at a given point in time. Let us now use $s \leq n$ to denote the effective alphabet size. Our aim is to maintain the space within $nH_0(S) + o(n \lg s) + O(s \lg n)$ bits, even when the symbols come from a large universe $\Sigma = [1..|\Sigma|]$, or even from a general ordered universe such as $\Sigma = \mathbb{R}$ or $\Sigma = \Gamma^*$ (i.e., Σ are words over another alphabet Γ).

Our mappings SN and NS of Section 7 give a simple way to handle a sequence over an unbounded ordered alphabet. By changing SN to a custom structure to search Σ, and storing elements of Σ in array NS, we obtain the following results, using respectively Theorems 8.1 and 9.1.

THEOREM 10.4. *A dynamic string $S[1,n]$ over a general alphabet Σ can be stored in a structure using $nH_0(S) + o(n \lg s) + O(s \lg n) + \mathcal{S}(s)$ bits and supporting queries* access, rank *and* select *in time $O(\mathcal{T}(s) + \lg n / \lg\lg n)$. Insertions and deletions of symbols are supported in $O(\mathcal{U}(s) + \lg n / \lg\lg n)$ amortized time. Here $s \leq n$ is the*

number of distinct symbols of Σ occurring in S, $\mathcal{S}(s)$ is the number of bits used by a dynamic data structure to search over s elements in Σ plus to refer to s elements in Σ, $\mathcal{T}(s)$ is the worst-case time to search for an element among s of them in Σ, and $\mathcal{U}(s)$ is the amortized time to insert/delete symbols of Σ in the structure.

THEOREM 10.5. *A dynamic string $S[1,n]$ over a general alphabet Σ can be stored in a structure using $nH_0(S) + o(n \lg s) + O(s \lg n) + \mathcal{S}(s)$ bits and supporting queries* access *and* select *in time $O(\mathcal{T}(s) + \lg n/\lg\lg n)$ and* rank *in time $O(\mathcal{T}(s) + \lg n)$. Insertions and deletions of symbols are supported in $O(\mathcal{U}(s) + \lg n)$ time. Here $s \leq n$ is the number of distinct symbols of Σ occurring in S, $\mathcal{S}(s)$ is the number of bits used by a dynamic data structure to search over s elements in Σ plus to refer to s elements in Σ, $\mathcal{T}(s)$ is the time to search for an element among s of them in Σ, and $\mathcal{U}(s)$ is the time to insert/delete symbols of Σ in the structure. All times are worst-case*

For example, a sequence of arbitrary real numbers can be handled with a balanced search tree for the alphabet data structure that adds $O(\lg s)$ time to the operations. A large integer range $\Sigma = [1..|\Sigma|]$, instead, can be handled with a predecessor data structure that adds $O((\lg\lg|\Sigma|)^2)$ to the times, and $O(s \lg|\Sigma|)$ further bits. When the identity of the symbols is not important, one can handle a contiguous alphabet $[1..s]$, and only insert new symbols $s+1$. In this case there is no penalty for letting the alphabet grow dynamically.

The only case where a previous solution with dynamic alphabet exists [27] is for the case $\Sigma = \Gamma^*$ on an alphabet Γ. Here we can store the effective set of strings in a data structure by Franceschini and Grossi [21], so that operations involving a string a take time $O(|a| + \lg\gamma + \lg n/\lg\lg n)$, where γ is the number of symbols of Γ actually in use. With Theorem 10.5 we obtain worst-case time $O(|a| + \lg\gamma + \lg n)$. The previous dynamic data structure [27] requires time $O(|a| \lg\gamma \lg n)$ (although its space could be lower than ours).

Handling general alphabets impacts on various dynamic representations that build on access, rank and select operations on strings, where alphabet dynamism is essential. For example, it allows inserting/deleting both objects and labels in binary relations [4], inserting/deleting nodes in graphs [18], inserting/deleting both rows and columns in discrete grids [10], and inserting new words and deleting words from text collections in positional [6] and non-positional [3] inverted indexes [1].

11. Conclusions and Further Challenges. We have obtained $O(\lg n/\lg\lg n)$ time for all the operations that handle a dynamic sequence on an arbitrary alphabet $[1..\sigma]$, matching lower bounds that apply to binary alphabets [22], and using zero-order compressed space. Our structure is faster than the best previous work [29, 49] by a factor of $\Theta(\lg\sigma/\lg\lg n)$ when the alphabet is larger than polylogarithmic. The query times are worst-case, yet the update times are amortized. We also show that it is possible to obtain worst-case for all the operations, although times for rank and updates raises to $O(\lg n)$. Finally, we show how to handle general and infinite alphabets. Our result can be applied to a number of problems and improve previous upper bounds on those; we have described several ones.

The lower bounds [22] are valid also for amortized times, so our amortized-time solution is optimal, yet our worst-case solution could be not. The main remaining challenge is to determine whether it is possible to attain the optimal $O(\lg n/\lg\lg n)$ worst-case time for all the operations.

Another interesting challenge is to support a stronger set of update operations, such as block edits, concatenations and splits in the sequences. Navarro and Sadakane [49] support those operations within time $O(\sigma \lg^{1+\varepsilon} n)$. While it seems feasible to

achieve, in our structure, $O(\sigma \lg n)$ time by using blocks of $\Theta(\lg^2 n)$ bits, the main hurdle is the difficulty of mimicking the same splits and concatenations on the list maintenance data structures we use [33, 44].

REFERENCES

[1] R. Baeza-Yates and B. Ribeiro, *Modern Information Retrieval*, Addison-Wesley, 2nd ed., 2011.

[2] J. Barbay, F. Claude, T. Gagie, G. Navarro, and Y. Nekrich, *Efficient fully-compressed sequence representations*, Algorithmica, 69 (2014), pp. 232–268.

[3] J. Barbay, F. Claude, and G. Navarro, *Compact binary relation representations with rich functionality*, Information and Computation, 232 (2013), pp. 19–37.

[4] J. Barbay, A. Golynski, I. Munro, and S. S. Rao, *Adaptive searching in succinctly encoded binary relations and tree-structured documents*, Theoretical Computer Science, 387 (2007), pp. 284–297.

[5] J. Barbay, M. He, I. Munro, and S. S. Rao, *Succinct indexes for strings, binary relations and multi-labeled trees*, ACM Transactions on Algorithms, 7 (2011), p. article 52.

[6] J. Barbay and G. Navarro, *Compressed representations of permutations, and applications*, in Proc. 26th International Symposium on Theoretical Aspects of Computer Science (STACS), 2009, pp. 111–122.

[7] D. Belazzougui and G. Navarro, *New lower and upper bounds for representing sequences*, in Proc. 20th Annual European Symposium on Algorithms (ESA), LNCS 7501, 2012, pp. 181–192.

[8] D. Blandford and G. Blelloch, *Compact representations of ordered sets*, in Proc. 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2004, pp. 11–19.

[9] G. Blelloch, *Space-efficient dynamic orthogonal point location, segment intersection, and range reporting*, in Proc. 19th Annual ACM-SIAM Symposium on Discrete Algorithms, (SODA), 2008, pp. 894–903.

[10] P. Bose, M. He, A. Maheshwari, and P. Morin, *Succinct orthogonal range search structures on a grid with applications to text indexing*, in Proc. 11th International Symposium on Algorithms and Data Structures (WADS), 2009, pp. 98–109.

[11] N. Brisaboa, A. Fariña, S. Ladra, and G. Navarro, *Implicit indexing of natural language text by reorganizing bytecodes*, Information Retrieval, 15 (2012), pp. 527–557.

[12] M. Burrows and D. Wheeler, *A block sorting lossless data compression algorithm*, Tech. Report 124, Digital Equipment Corporation, 1994.

[13] H. Chan, W.-K. Hon, T.-H. Lam, and K. Sadakane, *Compressed indexes for dynamic text collections*, ACM Transactions on Algorithms, 3 (2007), p. article 21.

[14] H. Chan, W.-K. Hon, and 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.

[15] B. Chazelle and L. Guibas, *Fractional cascading: I. a data structuring technique*, Algorithmica, 1 (1986), pp. 133–162.

[16] ———, *Fractional cascading: II. applications*, Algorithmica, 1 (1986), pp. 163–191.

[17] D. Clark, *Compact Pat Trees*, PhD thesis, University of Waterloo, Canada, 1996.

[18] F. Claude and G. Navarro, *Extended compact Web graph representations*, in Algorithms and Applications (Ukkonen Festschrift), LNCS 6060, Springer, 2010, pp. 77–91.

[19] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan, *Compressing and indexing labeled trees, with applications*, Journal of the ACM, 57 (2009), p. article 4.

[20] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro, *Compressed representations of sequences and full-text indexes*, ACM Transactions on Algorithms, 3 (2007), p. article 20.

[21] G. Franceschini and R. Grossi, *A general technique for managing strings in comparison-driven data structures*, in Proc. 31st International Colloquium on Automata, Languages and Computation (ICALP), LNCS 3142, 2004, pp. 606–617.

[22] M. Fredman and M. Saks, *The cell probe complexity of dynamic data structures*, in Proc. 21st Annual ACM Symposium on Theory of Computing (STOC), 1989, pp. 345–354.

[23] A. Golynski, I. Munro, and S. S. Rao, *Rank/select operations on large alphabets: a tool for text indexing*, in Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2006, pp. 368–373.

[24] R. González and G. Navarro, *Improved dynamic rank-select entropy-bound structures*, in Proc. 8th Latin American Symposium on Theoretical Informatics (LATIN), LNCS 4957, 2008, pp. 374–386.

[25] R. González and G. Navarro, *Rank/select on dynamic compressed sequences and applications*, Theoretical Computer Science, 410 (2009), pp. 4414–4422.
[26] R. Grossi, A. Gupta, and J. Vitter, *High-order entropy-compressed text indexes*, in Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2003, pp. 841–850.
[27] R. Grossi and G. Ottaviano, *The wavelet trie: maintaining an indexed sequence of strings in compressed space*, in Proc. 31st ACM Symposium on Principles of Database Systems (PODS), 2012, pp. 203–214.
[28] A. Gupta, W.-K. Hon, R. Shah, and J. Vitter, *A framework for dynamizing succinct data structures*, in Proc. 34th International Colloquium on Automata, Languages and Computation (ICALP), 2007, pp. 521–532.
[29] M. He and I. Munro, *Succinct representations of dynamic strings*, in Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE), LNCS 6393, 2010, pp. 334–346.
[30] W.-K. Hon, K. Sadakane, and W.-K. Sung, *Succinct data structures for searchable partial sums*, in Proc. 14th Annual International Symposium on Algorithms and Computation (ISAAC), LNCS 2906, 2003, pp. 505–516.
[31] ———, *Breaking a Time-and-Space Barrier in Constructing Full-Text Indices*, SIAM Journal of Computing, 38 (2009), pp. 2162–2178.
[32] ———, *Succinct data structures for searchable partial sums with optimal worst-case performance*, Theoretical Computer Science, 412 (2011), pp. 5176–5186.
[33] H. Imai and T. Asano, *Dynamic segment intersection search with applications*, in Proc. 25th Symposium on Foundations of Computer Science (FOCS), 1984, pp. 393–402.
[34] Juha Kärkkäinen, *Fast BWT in small space by blockwise suffix sorting*, Theoretical Computer Science, 387 (2007), pp. 249–257.
[35] J. Kärkkäinen and S. J. Puglisi, *Fixed block compression boosting in FM-indexes*, in Proc. 18th International Symposium on String Processing and Information Retrieval (SPIRE), LNCS 7024, 2011, pp. 174–184.
[36] S. Lee and K. Park, *Dynamic rank-select structures with applications to run-length encoded texts*, in Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM), LNCS 4580, 2007, pp. 95–106.
[37] ———, *Dynamic rank/select structures with applications to run-length encoded texts*, Theoretical Computer Science, 410 (2009), pp. 4402–4413.
[38] V. Mäkinen and G. Navarro, *Dynamic entropy-compressed sequences and full-text indexes*, in Proc. 17th Annual Symposium on Combinatorial Pattern Matching (CPM), LNCS 4009, 2006, pp. 307–318.
[39] ———, *Dynamic entropy-compressed sequences and full-text indexes*, ACM Transactions on Algorithms, 4 (2008), p. article 32.
[40] C. Makris, *Wavelet trees: a survey*, Computer Science and Information Systems, 9 (2012), pp. 585–625.
[41] G. Manzini, *An analysis of the Burrows-Wheeler transform*, Journal of the ACM, 48 (2001), pp. 407–430.
[42] K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*, EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1984.
[43] K. Mehlhorn and S. Näher, *Dynamic fractional cascading*, Algorithmica, 5 (1990), pp. 215–241.
[44] C. Mortensen, *Fully-dynamic two dimensional orthogonal range and line segment intersection reporting in logarithmic time*, in Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2003, pp. 618–627.
[45] I. Munro, *An implicit data structure supporting insertion, deletion, and search in $O(\log n)$ time*, Journal of Computer and Systems Sciences, 33 (1986), pp. 66–74.
[46] ———, *Tables*, in Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), LNCS 1180, 1996, pp. 37–42.
[47] G. Navarro, *Wavelet trees for all*, Journal of Discrete Algorithms, 25 (2014), pp. 2–20.
[48] G. Navarro and V. Mäkinen, *Compressed full-text indexes*, ACM Computing Surveys, 39 (2007), p. article 2.
[49] G. Navarro and K. Sadakane, *Fully-functional static and dynamic succinct trees*, ACM Transactions on Algorithms, 10 (2014), p. article 16. Previously arXiv:0905.0768.
[50] Y. Nekrich, *A dynamic stabbing-max data structure with sub-logarithmic query time*, in Proc. 22nd International Symposium on Algorithms and Computation, (ISAAC), 2011, pp. 170–179.
[51] D. Okanohara and K. Sadakane, *A linear-time Burrows-Wheeler transform using induced sorting*, in Proc. 16th International Symposium on String Processing and Information

Retrieval (SPIRE), LNCS 5721, 2009, pp. 90–101.

[52] M. Patrascu, *Lower bounds for 2-dimensional range counting*, in Proc. 39th Annual ACM Symposium on Theory of Computing (STOC), 2007, pp. 40–46.

[53] R. Raman, V. Raman, and S. S. Rao, *Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets*, ACM Transactions on Algorithms, 3 (2007), p. article 8.

[54] R. Raman and S. S. Rao, *Succinct dynamic dictionaries and trees*, in Proc. 30th International Colloquium on Automata, Languages and Computation (ICALP), LNCS 2719, 2003, pp. 357–368.

[55] N. Välimäki and V. Mäkinen, *Space-efficient algorithms for document retrieval*, in Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM), LNCS 4580, 2007, pp. 205–215.