Improved Dynamic Rank-Select Entropy-Bound Structures

Rodrigo González* and Gonzalo Navarro**

Dept. of Computer Science, University of Chile.
{rgonzale,gnavarro}@dcc.uchile.cl

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

1 Introduction and Related Work

Compressed data structures aims at representing classical data structures such as sequences, trees, graphs, etc., in little space while keeping the functionality of the structure. That is, compressed data structures should operate without the need to decompress them. This is a very active area of research stimulated by today's steep memory hierarchies and large available data sizes. See e.g. [16].

One of the most useful structures are the bit vectors with *rank* and *select* operations: $rank(B, i)$ gives the number of 1-bits in $B[1, i]$ and $select(B, i)$ gives the position of the i-th 1 in B. This generalizes to sequences $T[1, n]$ over an alphabet Σ of size σ, where one aims at a (hopefully compressed) representation efficiently supporting the following operations:

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

Improvements in *rank*/*select* operations on sequences have a great impact on many other succinct data structures, especially on those aimed at text indexing [6, 3, 11, 16], but also labeled trees, structured text, binary relations, graphs, and others [1, 2, 8].

The first structure providing support for *rank*/*select* on a sequence of symbols was the *wavelet tree* [7, 5]. Wavelet trees are perfectly balanced static trees of height $\log \sigma$ (logarithms are in base 2 by default). They answer the three queries in $O(\log \sigma)$ time, by working $O(1)$ per tree level. They store a bitmap of length n per level, which is preprocessed for constant-time binary *rank*/*select* queries. Their total space requirement is $n \log \sigma + o(n \log \sigma)$, where the extra sublinear term is the space needed by the binary *rank*/*select* structures [15]. By representing those bitmaps in

* Supported in part by Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile.
** Supported in part by Fondecyt Grant 1-050493, Chile.

compressed form [19] the constant-time *rank*/*select* queries are retained and the space becomes $nH_0(T) + o(n \log \sigma)$, where $H_0(T)$ is the zero-order empirical entropy of T (that is, $\sum_{c \in \Sigma} \frac{n_c}{n} \log \frac{n}{n_c}$, where c occurs n_c times in T). Since the wavelet tree gives $access(T, i)$ to any symbol $T[i]$, it can be used to *replace* T.

A stronger version of wavelet trees are *multiary wavelet trees* [3], which achieve the same space but improve the query times to $O(1 + \frac{\log \sigma}{\log \log n})$. The trick is to make the tree m-ary for, say, $m = \sqrt{\log n}$, so that its height is reduced. Now the tree does not store a bitmap per level, but rather a sequence over an alphabet of size m. They show how to do *rank*/*select* on those sequences in constant time for such a small m.

In [11] they add dynamic capabilities to these sequences, by adding operations

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

They represent dynamic bitmaps B using $nH_0(B) + o(n)$ bits of space and solve all operations in $O(\log n)$ time. This is done with a binary tree that stores $\Theta(\log^2 n)$ bits at the leaves, and at internal nodes stores summary *rank*/*select* information on the subtrees. For larger alphabets, a wavelet tree using dynamic bitmaps yields a dynamic sequence representation that takes $nH_0(T) + o(n \log \sigma)$ bits and solves all the operations in time $O(\log n \log \sigma)$.

Very recently, in [10] they manage to improve the time complexities of this solution. They show that the $O(\log n)$ time complexities can be achieved for alphabets of size up to $\sigma = O(\log n)$. They combine this tool with a multiary wavelet tree to achieve $O(\log n(1 + \frac{\log \sigma}{\log \log n}))$ time.

The key to the success of [10] is a clever detachment of two roles of tree leaves that are entangled in [11]: In the latter, the leaves are the memory allocation unit (that is, whole leaves are allocated or freed), and also the information summarization unit (that is, the tree maintains information up to leaf granularity, and the rest has to be collected by sequentially scanning a leaf). In [10] leaves are not the memory allocation unit, but handle an internal linked list with smaller memory allocation units. This permits moving symbols to accommodate the space upon insertions/deletions within a leaf, without having to update summarization information for the data moved. This was the main bottleneck that prevented the use of larger alphabets in $O(\log n)$ time in [11].

However, compared to [11], the work in [10] has several weaknesses: (1) it is not compressed, but rather takes $n \log \sigma + O(n) + o(n \log \sigma)$ bits of space; (2) in addition to not compressing T, the extra space includes an $O(n)$ term, as shown; (3) times are amortized, not worst-case.

In this paper we show that it is possible to obtain the best from both worlds. We combine the works in [11, 10] to obtain a structure that (1) takes $nH_0(T) + o(n \log \sigma)$ bits of space, and (2) performs all the operations in $O(\log n(1 + \frac{\log \sigma}{\log \log n}))$ worst-case time. (This is achieved even for the case where $\lceil \log n \rceil$ changes and so does the length of the structure pointers in order to maintain the promised space bounds.) The result becomes the most efficient dynamic representation of sequences, both in time and space, and its benefits have immediate applications to other succinct data structures such as compressed text indexes, as we show at the end.

This combination is by no means simple. Some parts are easy to merge, such as the role detachment for leaves [10] with the compressed representation of sequences [3] and multi-ary wavelet trees, plus the memory management techniques to support changes of $\lceil \log n \rceil$ within the same worst-case time bounds and no extra space [11]. However, others require new algorithmic ideas. In [10] they spend $O(n)$ extra bits in bitmaps that maintain leaf-granularity information on *rank*/*select*. We show that this can be replaced by dynamic partial sums, which use sublinear space. However, we

need σ partial sums and cannot afford to update them individually upon a leaf insertion/deletion. Hence we create a new structure where a collection of σ sequences are maintained in synchronization, and this can be of independent interest. The second problem was that leaf splitting/merging in [10] triggered too many updates to summarization data, which could not be handled in $O(\log n)$ worst-case time, only in $O(\log n)$ amortized time. To get rid of this problem we redefined the leaf fill ratio invariants, preferring a weaker condition that still ensures that leaves are sufficiently full and can be maintained within the $O(\log n)$-worst-case-time bound.

Finally, we remind that there is a static sequence representation [6] that requires $n \log \sigma + n\, o(\log \sigma)$ bits and answers the queries in $O(\log \log \sigma)$ time. There has been work on dynamizing this structure [8], so that the query times are $O(\frac{1}{\alpha} \log \log n)$, in exchange for insertion/deletion times for the form $O(n^\alpha)$ for constant $0 < \alpha < 1$.

We recall that our results (and all the mentioned ones) assume a RAM model with word size $w = \Omega(\log n)$, so that operations on $O(\log n)$ contiguous bits can be carried out in constant time. For the dynamic structures, we always allocate $\omega(\log n)$-bit chunks of the same size, which can be handled in constant time and asymptotically no extra space with simple memory allocation algorithms.

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

2 Collection of Searchable Partial Sums with Indels

The *Searchable Partial Sums with Indels* problem [9] consists in maintaining a sequence S of nonnegative integers $s_1, \ldots, s_n$, each one of $k = O(\log n)$ bits, supporting the following queries and operations:

- $sum(S, i)$ is $\sum_{l=1}^{i} s_l$.
- $search(S, y)$ is the smallest i' such that $sum(S, i') \geq y$.
- $update(S, i, x)$ updates s_i to $s_i + x$ (x can be negative as long as the result is not).
- $insert(S, i, x)$ inserts a new integer x between s_{i-1} and s_i.
- $delete(S, i)$ deletes s_i from the sequence.

It is possible to handle all these operations using $kn + o(kn)$ bits of space and $O(\log n)$ time for all the operations [11]. We now define an extension of this problem, that we call *Collection of Searchable Partial Sums with Indels*. This problem consists in maintaining a collection of σ sequences $C = \{S^1, \ldots, S^\sigma\}$ of nonnegative integers $\{s_i^j\}_{1 \leq j \leq \sigma, 1 \leq i \leq n}$, each one of $k = O(\log n)$ bits. We support the following queries and operations:

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

Next we show how to carry out all of these queries/operations in $O(\sigma + \log n)$ time, using $O(\sigma k n)$ bits of space.

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

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

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

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

For the next operations, we note that a leaf has at most $m = \lfloor \frac{2\log^2 n}{\sigma k} \rfloor$ integers from any sequence. Then a subsequence of a given sequence has at most mk bits. So if we copy a subsequence in chunks of $\frac{1}{2}\log n$ bits, the subsequence will be copied in $1 + \frac{2mk}{\log n} = O(1 + \frac{\log n}{\sigma})$ time in the RAM model (this requires shifting bits, which in case it is not supported by the model can be handled using small universal tables of the kind of Y). As we have σ sequences, we can copy a given subsequence of them all in $O(\sigma + \log n)$ time. The next operations are solved by a constant number of these copying operations.

Operation $insert(C, i)$. We traverse the tree similarly to *sum*, updating $p(v)$ as we traverse the tree. That is, we increase $p(v)$ by 1 each time we go left from v. Then we copy the leaf arrived at

[1] If $\sigma k > \frac{1}{2}\log^2 n$, we just store σk bits per leaf. All the algorithms in the sequel are simplified and the complexities are maintained.

to a new memory area, adding a 0 between s_{i-1}^j and s_i^j for all j. This is done by first copying the subsequences $\ldots s_{i-1}^j$ for all j, then adding 0 to each sequence, and finally copying the subsequences $s_i^j \ldots$ for all j. As we have just explained, this can be done in $O(\sigma + \log n)$ time.

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

Operation delete(C, i). We traverse the tree similarly to *sum*, updating $p(v)$ while we traverse the tree. That is, we decrease $p(v)$ by 1 each time we go left from v. Then we copy the leaf to a new memory area, deleting s_i^j for all j, similarly to *insert*, in $O(\sigma + \log n)$ time.

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

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

The space requirement is at most $4\sigma kn$ bits for all the leaves. For each internal node we have two pointers, a $p(\cdot)$ counter, and σ $r^j(\cdot) \leq 2^k \cdot n$ counters, totalizing $O(\log n) + \sigma(k + \log n) = O(\sigma \log n)$ bits per node. So, all the internal nodes use $O(\frac{\sigma kn}{\log^2 n}\sigma \log n) = O(\frac{\sigma^2 kn}{\log n})$ bits. We have proved our claim.

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

We note that we have actually assumed that $w = \Theta(\log n)$ in our space computation (as we have used w-bit system pointers). The general case $w = \Omega(\log n)$ can be addressed using exactly the same techniques developed in [11], using a more refined memory management with pointers of $(\log n) \pm 1$ bits, and splitting the sequence into three in a way that retains the worst-case complexities.

We also note that the space can be improved to $\sigma kn(1 + O(\frac{\sigma}{\log n}))$ by using a finer memory allocation policy for the leaves, just as done in the next sections for sequences. The simpler result suffices for the use we make of Theorem 1 in this paper.

3 Uncompressed Dynamic Rank-Select Structures for a Small Alphabet

For a small alphabet of size $\sigma = O(\log n)$, we construct a red-black tree over $T[1, n]$ where each leaf contains a non-empty *superblock* of size up to $2\log^2 n$ bits. We will introduce invariants that guarantee that there are at most $\frac{2n\log\sigma}{\log^2 n}$ superblocks. Each internal node v stores counters $r(v)$ and $p(v)$, where $r(v)$ is the number of superblocks in the left subtree and $p(v)$ is the number of symbols stored in the left subtree. For each superblock i, we maintain s_i^j, the number of occurrences of symbol j in superblock i. We store all these sequences of numbers using a *Collection of Searchable Partial Sums with Indels*, C. The length of each sequence will be at most $\frac{2n\log\sigma}{\log^2 n}$ integers, $\sigma = O(\log n)$ and $k = O(\log\log n)$. So the partial sums operate in $O(\log n)$ worst-case time.

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

A superblock storing less than $\log^2 n$ bits will be called *sparse*. The operations *insert* and *delete* will maintain the invariant that no two consecutive sparse superblocks may exist. This ensures that every consecutive pair of superblocks holds at least $\log^2 n$ bits from T, and hence that there are at most $\frac{2n\log\sigma}{\log^2 n}$ superblocks.

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

- The text itself uses $n\log\sigma$ bits of space.
- The *Collection of Searchable Partial Sums with Indels* uses $O(\frac{n\log\log n\log\sigma}{\log n})$ bits of space.
- Each pointer of the linked list of blocks uses $O(\log n)$ bits and we have $O(\frac{n\log\sigma}{\sqrt{\log n}\log n})$ blocks, totalizing $O(\frac{n\log\sigma}{\sqrt{\log n}})$ bits.
- The last block in each superblock is not necessarily fully used. We have at most $\frac{2n\log\sigma}{\log^2 n}$ superblocks, each of which can waste a full block of size $\sqrt{\log n}\log n$ bits, totalizing $O(\frac{n\log\sigma}{\sqrt{\log n}})$ bits.
- Inside each block, we can lose at most $\log\sigma$ bits due to symbol misalignment, totalizing $O(\frac{n\log^2\sigma}{\sqrt{\log n}\log n}) = O(\frac{n\log\log n\log\sigma}{\sqrt{\log n}\log n})$ bits.
- The internal tree counters use $O(\frac{n\log\sigma}{\log^2 n}\cdot\log n) = O(\frac{n\log\sigma}{\log n})$ bits.

Now we show how to carry out all of the queries/operations in $O(\log n)$ time.

First, it is important to notice that each block can be scanned or shifted in $\sqrt{\log n}$ time, using tables that process chunks of $\frac{1}{2}\log n$ bits (again, if $\sigma > \frac{1}{2}\log n$ we can process each symbol individually within the time bounds). Given that there are at most $O(\sqrt{\log n})$ blocks in a superblock, we can scan or shift elements within a block in $O(\log n)$ time, even considering block boundaries.

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

Computing $rank_c(T, i)$. We find the leaf containing the i-th position, just as for *access*. Then we scan superblock sb from the first block summing up the occurrences of c up to the position pos, using a table Z to sum the c's. We add to this quantity $sum(C, c, sb-1)$, the number of times that c appears before superblock sb. The *rank* query takes in total $O(\log n)$ time. Table Z is of the same spirit of Y and requires $O(\sigma\sqrt{n}\ \text{polylog}(n)) = O(\sqrt{n}\ \text{polylog}(n))$ bits.

Computing $select_c(T, i)$. We calculate $j = search(C, c, i)$; this way we know that the i-th c belongs to superblock j and it is the i'-th appearance of c within superblock j, for $i' = i - sum(C, c, j-1)$. Then we traverse the tree to find the leaf containing superblock j. We start with $sb \leftarrow j$ and $pos \leftarrow 0$. if $r(v) \geq sb$ we enter the left subtree, otherwise we enter the right subtree with $sb \leftarrow sb - r(v)$ and $pos \leftarrow pos + p(v)$. We reach the leaf that represents superblock j in $O(\log n)$ time. Then we scan superblock j from the first block, searching for the position of the i'-th appearance of symbol c within superblock j, using table Z. To this position we add pos to obtain the final result. The *select* query takes in total $O(\log n)$ time.

Operation $insert_c(T, i)$. We obtain sb and pos just like in the *access* query, except that we start with $pos \leftarrow i - 1$, so as to insert right after position $i - 1$. Then, if superblock sb contains room for one more symbol, we insert c at the pos-th position of sb, by shifting the symbols through the blocks as explained. We also carry out $update(C, c, sb, 1)$ and retraverse the path from the root to sb adding 1 to $p(v)$ each time we go left from v.

If this insertion causes an overflow in the last block, we simply add a new block at the end of the linked list to hold the trailing symbol (which is usually not the same symbol inserted of course). In this case we finish in $O(\log n)$ time.

If, instead, the superblock is full, we cannot carry out the insertion yet. We first try to move one symbol to the previous superblock (if it exists). We check how many symbols does superblock $sb - 1$ have: we traverse the tree searching for it, and deduce its size from the $r(v)$ counters in the tree. If it can hold one more symbol, we $insert_d(T, \cdot)$ the first symbol d of superblock sb into superblock $sb - 1$. This recursive invocation to *insert* will not overflow superblock $sb - 1$. Now, we $delete_d(T, \cdot)$ the symbol d moved from block sb, and this cannot cause an underflow of sb. Now that we have made room to carry out the original insertion, we rerun $insert_c(T, i)$ and it will not overflow again. This takes $O(\log n)$ time.

If superblock $sb-1$ is also full or does not exist, then we are entitled to create a sparse superblock between $sb - 1$ and sb, without breaking the invariant on sparse superblocks. We create such an empty superblock and $insert_d(T, \cdot)$ symbol d into it. Finally we rerun $insert_c(T, i)$ as before.

To create the new superblock, we search for superblock sb, updating $r(v)$ to $r(v) + 1$ each time we go left from v. When we arrive at the leaf of sb we create a new node μ with $r(\mu) = 1$ and $p(\mu) = 0$. Its left child is the new empty superblock and its right child is sb. We also execute $insert(C, sb)$.

Finally, we check if we need to rebalance the tree. If it is needed, it can be done with one rotation and $O(\log n)$ red-black tag updates, given that we use a red-black tree. After a rotation we need to update $r(\cdot)$ and $p(\cdot)$ only for three nodes. These updates can be done in $O(1)$ time and the whole *insert* operation takes $O(\log n)$ time.

Operation $delete(T, i)$. We obtain sb and pos just as in the *access* query, updating $p(v)$ to $p(v) - 1$ each time we go left from v. Then we delete the pos-th position (let c be the symbol deleted) of the

sb-th superblock, by shifting the symbols back through the blocks. If this deletion empties the last block, we free it. In any case we call $update(C, c, sb, -1)$ on the partial sums.

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

If superblock *sb* becomes empty, we search for it just like in the *access* query, updating $r(v)$ to $r(v) - 1$ each time we go left from v, in $O(\log n)$ time. When we arrive at the leaf that represents superblock *sb* we delete it. Then we do operation $delete(C, sb)$. Finally, we check if we need to rebalance the tree. If needed, this can be done with one rotation and $O(\log n)$ red-black tag updates, just as for insertion. After a rotation we also need to update $r(\cdot)$ and $p(\cdot)$ only for three nodes. These updates take constant time.

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

If superblock $sb - 1$ is sparse, we $delete(T, \cdot)$ its last symbol d, and $insert_d(T, \cdot)$ it in the beginning of superblock *sb*. This recursive call brings no problems because $sb - 1$ is already sparse, and we restore the non-sparse status of *sb*. The action is symmetric if $sb - 1$ is not sparse but $sb + 1$ is.

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

Theorem 2. *Given a text T of length n over a small alphabet of size $\sigma = O(\log n)$, the Dynamic Sequence with Indels problem under RAM model with word size $w = \Omega(\log n)$ can be solved using $n \log \sigma + O(\frac{n \log \sigma}{\sqrt{\log n}})$ bits of space, supporting the queries access, rank, select, insert and delete in $O(\log n)$ worst-case time.*

We note again that we have actually assumed that $w = \Theta(\log n)$ in our space computation, and that the general case $w = \Omega(\log n)$ can be obtained using exactly the same techniques developed in [11, Sections 4.5, 4.6, and 6.4].

4 Compressed Dynamic Rank-Select Structures

Theorem 2 can be extended to use a compressed sequence representation, by just changing the way we store/manage the blocks. The key idea is to detach the *representational* and the *actual* (i.e., compressed) sizes of the storage units at different levels.

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

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

In logical terms, a superblock is be divided into *subblocks* representing $\frac{1}{2}\log n$ bits (that is, $\frac{1}{2}\log_\sigma n$ symbols[2]) from T. We represent each subblock using the (c, o)-pair encoding of [3]: The c part is of fixed width and tells how many occurrences of each alphabet symbol are there in the subblock; whereas the o part is of variable width and gives the identifier of the subblock among those sharing the same c component. Each c component uses at most $\sigma \log\log n$ bits; while the o components use at most $O(\log n)$ bits each, and overall add up to $nH_0(T) + O(n \log\sigma / \log n)$ bits [3, Section 3.1].

In a block of $\sqrt{\log n}\log n$ bits, we store as many subblocks as they fit, wasting at most $\sigma \log\log n + O(\log n)$ unused bits at the end. The universal tables (like T) used to sequentially process the blocks in chunks of $\Theta(\log n)$ bits must now be modified to process the sequence subblock-wise. This is complex because an insertion in a subblock introduces a displacement that propagates over all the subblocks of the block, which must be completely recomputed and rewritten (and it can even cause the actual size of the whole superblock to double!). Fortunately all those tedious details have been already sorted out in [11, Sections 5.2, 6.1, and 6.2], where their superblocks play the role of our blocks, and their tree rearrangements are not necessary for us because we are within a leaf now. Their "partial blocks" mechanism is also not useful for us, because we can tolerate those propagations to extend over all the blocks of our superblocks. Hence only the last block of our superblocks is not as full as misalignments permit.

The time achieved in [11] is $O(1)$ per $\Theta(\log n)$ actual bits. Even in the worst case (where compression does not work at all in the superblock), the number of actual bits will be $\frac{2\log^2 n}{\frac{1}{2}\log n}(\sigma \log\log n + O(\log n)) = O(\log^2 n + \sigma \log n \log\log n)$, and thus the time to solve any query or carry out any update on a superblock will be $O(\log n + \sigma \log\log n)$.

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

- The text itself (as a sequence of pairs (c, o)) uses $nH_0(T) + O((\sigma n \log\log n)/\log_\sigma n)$ bits of space.
- Inside each block, we can lose at most $O(\sigma \log\log n + \log n)$ bits due to misalignments, totalizing $O(\frac{n \log\sigma(\sigma \log\log n + \log n)}{\sqrt{\log n \log n}})$ bits of space.
- The extra space for the tables to operate the (c, o) encoding is $O(\sqrt{n}\ \sigma\ \mathrm{polylog}(n))$ bits.

It can be seen that the time and space complexities depend sharply on σ. Thus the solution is indeed of interest only for rather small $\sigma = o(\log n / \log\log n)$. For such a small alphabet we have the following theorem.

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

Again, all the issues of varying $\lceil \log n \rceil$ and the case $w = \omega(\log n)$ are handled just as in [11, Sections 4.5, 4.6, and 6.4]

To extend our results for a large alphabet of size $\sigma = \Omega(\log n / \log\log n)$, we use a generalized ρ-ary wavelet tree [3] over T, where $\rho = \sqrt{\log n}$. This generalized wavelet tree has $O(\log_\rho \sigma) =$

[2] We have ignored floors and ceilings for simplicity.

$O(\frac{\log\sigma}{\log\log n})$ levels. We store on each level a sequence over an alphabet of size ρ, which can be handled using the dynamic solution of Theorem 3, for which ρ is small enough. Hence each query and operation takes $O(\log n)$ time per level, adding up $O(\log n \frac{\log\sigma}{\log\log n})$ worst-case time overall.

As shown in [3], the sum of the zero-order-entropy representations of the sequences at each level adds up to the zero-order entropy of T. In addition, the generalized ρ-ary wavelet tree handles changes in $\lceil \log n \rceil$ automatically, as this handling is encapsulated within each level.

We thus obtain our main theorem, where we have included the case of small σ as well.

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

5 Conclusions

We have shown that two existing solutions to the *Dynamic Sequence with Indels* problem [11, 10] can be merged so as to obtain the best from both. This merging is not trivial and involves some byproducts that can be of independent interest. We show now a couple of immediate consequences of our improved result.

In [11, 12] it is shown that a wavelet tree built over the Burrows-Wheeler Transform T^{bwt} of a text T, and compressed using the (c, o) pair technique achieves high-order entropy space, namely $nH_h(T) + o(n\log\sigma)$ for any $h + 1 \leq \alpha \log_\sigma n$ and constant $0 < \alpha < 1$, where $H_h(T)$ is the h-th order empirical entropy of T [14]. This is used in [11] to obtain a dynamic text index that handles a collection $\mathcal{C}$ of texts and permits searching for patterns, extracting text snippets, and inserting/deleting texts in/from the collection. Using the definitions of [11, Section 7] and using the same sampling step, we can state a stronger version of those theorems:

Theorem 5. *The Dynamic Text Collection problem can be solved with a data structure of size $nH_h(\mathcal{C}) + o(n\log\sigma) + O(\sigma^{h+1}\log n + m\log n + w)$ bits, simultaneously for all h. It supports counting of the occurrences of a pattern P in $O(|P|\log n(1 + \frac{\log\sigma}{\log\log n}))$ time, and inserting and deleting a text T in $O(|T|\log n(1 + \frac{\log\sigma}{\log\log n}))$ time. After counting, any occurrence can be located in time $O(\log n + \log_\sigma n \log\log n)$. Any substring of length ℓ from any T in the collection can be displayed in time $O(\log n + \log_\sigma n \log\log n + \ell(1 + \frac{\log\sigma}{\log\log n}))$. Here n is the length of the concatenation $\mathcal{C} = 0\ T_1 0\ T_2 \cdots 0\ T_m$, and we assume $\sigma = o(n)$. For $h \leq (\alpha\log_\sigma n) - 1$, for any constant $0 < \alpha < 1$, the space complexity simplifies to $nH_h(\mathcal{C}) + o(n\log\sigma) + O(m\log n + w)$ bits.*

When $\sigma = O(\text{polylog}(n))$ is not too large, the times above become $O(|P|\log n)$ for counting, $O(|T|\log n)$ for text insertion/deletion, $O(\log_\sigma n \log\log n)$ for locating, and $O(\log_\sigma n \log\log n + \ell)$ for displaying.

Another important application that derives from this one is the compressed construction of text indexes. For example, a variant of the FM-index [3] requires h-th entropy space once built, but in order to build it we need $O(n\log n)$ bits of space. The previous theorem can be used in order to build the FM-index of a text by starting with an empty collection and inserting the text T of interest. Our new results make this process faster.

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

We note that this is the same space required for the final FM-index [3]. On the other hand, a *suffix array* [13] of T is easily derived from our dynamic FM-index. When $\sigma = O(\text{polylog}(n))$, our FM-index construction time becomes $O(n \log n)$ worst-case time, and the suffix array is obtained with another $O(n \log n)$ time process. This was indeed the best known complexity to build the suffix array before the linear-time construction algorithms were discovered a few years ago [18]. Yet, all the classical algorithms require $O(n \log n)$ bits whereas our result requires $n \log \sigma(1 + o(1))$ bits even on uncompressible texts.

Existing lower bounds on dynamic partial sums [17] suggest that sub-logarithmic query times are not possible with $O(\text{polylog}(n))$ update times, and hence $O(\log n)$ is the best we can hope for if we want to minimize the maximum complexity over all the operations (we recall that other approaches favor faster queries and slower updates [8]). Hence we believe that our time results cannot be improved in this sense, unless an improvement on the *static* representation for large alphabets (currently a multi-ary wavelet tree) is devised.

Alternatively, one would like to improve the space to high-order entropy (not only for the *Dynamic Text Collection* problem, but for the *Dynamic Sequence with Indels* problem). This has not been achieved even if we disregard operations *rank* and *select* and is satisfied only with *access*, *insert*, and *delete*.

Finally, one could be interested in extending the problem. An intriguingly difficult extension is that of handling a varying alphabet, so that not only the sequence changes, but also new unexpected symbols might appear, drawn from a virtual alphabet that is too large to admit a static-shaped wavelet tree to handle it [4].

References

1. J. Barbay, A. Golynski, I. Munro, and S. Rao. Adaptive searching in succinctly encoded binary relations and tree-structured documents. In *Proc. 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 24–35, 2006.
2. J. Barbay, M. He, I. Munro, and S. Rao. Succinct indexes for strings, binary relations and multi-labeled trees. In *Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 680–689, 2007.
3. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 3(2):article 20, 2007.
4. J. Fischer and V. Mäkinen. Personal communications. 2007.
5. L. Foschini, R. Grossi, A. Gupta, and J. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. *ACM Transactions on Algorithms (TALG)*, 2(4):611–639, 2006.
6. A. Golynski, I. Munro, and S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 368–373, 2006.
7. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proceedings 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.
8. A. Gupta, W. Hon, R. Shah, and J. Vitter. Dynamic rank/select dictionaries with applications to XML indexing. *Theoretical Computer Science*, 2007. To appear.
9. W.-K. Hon, K. Sadakane, and W.-K. Sung. Succinct data structures for searchable partials sums. In *Proceedings ISAAC'03, LNCS 2906*, pages 505–516, 2003.
10. 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, pages 95–106, 2007.
11. V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 2007. To appear. Also as Technical Report TR/DCC-2006-10, Dept. of Computer Science,

Univ. of Chile, July 2006, ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/dynamic.ps.gz. Previous version in *Proc. 17th CPM*, LNCS 4009, pp. 307–318.
12. V. Mäkinen and G. Navarro. Implicit compression boosting with applications to self-indexing. In *Proc. 14th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 4726, pages 214–226. Springer, 2007.
13. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal of Computing*, 22:935–948, 1993.
14. G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.
15. I. Munro. Tables. In *Proceedings 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS 1180, pages 37–42, 1996.
16. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.
17. M. Patrascu and E. Demaine. Logarithmic lower bounds in the cell probe model. *SIAM Journal on Computing*, 35(4):932–963, 2006.
18. S. Puglisi, W. Smyth, and A. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2):article 4, 2007.
19. R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 233–242, 2002.