Dynamic Entropy-Compressed Sequences and Full-Text Indexes

Veli Mäkinen[1] * and Gonzalo Navarro[2] **

[1] Department of Computer Science, University of Helsinki, Finland.
vmakinen@cs.helsinki.fi
[2] Department of Computer Science, University of Chile.
gnavarro@dcc.uchile.cl

Abstract. Given a sequence of n bits with binary zero-order entropy H_0, we present a dynamic data structure that requires $nH_0 + o(n)$ bits of space, which is able of performing *rank* and *select*, as well as inserting and deleting bits at arbitrary positions, in $O(\log n)$ worst-case time. This extends previous results by Hon et al. [ISAAC 2003] achieving $O(\log n/\log\log n)$ time for *rank* and *select* but $\Theta(\text{polylog}(n))$ amortized time for inserting and deleting bits, and requiring $n + o(n)$ bits of space; and by Raman et al. [SODA 2002] which have constant query time but a static structure. In particular, our result becomes the *first* entropy-bound dynamic data structure for *rank* and *select* over bit sequences.
We then show how the above result can be used to build a dynamic full-text self-index for a collection of texts over an alphabet of size σ, of overall length n and zero-order entropy H_0. The index requires $nH_0 + o(n\log\sigma)$ bits of space, and can count the number of occurrences of a pattern of length m in time $O(m\log n\log\sigma)$. Reporting the *occ* occurrences can be supported in $O(occ\log^2 n\log\sigma)$ time, paying $O(n)$ extra space. Insertion of text to the collection takes $O(\log n\log\sigma)$ time per symbol, which becomes $O(\log^2 n\log\sigma)$ for deletions. This improves a previous result by Chan et al. [CPM 2004]. As a consequence, we obtain an $O(n\log n\log\sigma)$ time construction algorithm for a compressed self-index requiring $nH_0 + o(n\log\sigma)$ bits *working space* during construction.

1 Introduction and Related Work

The study of compressed data structures aims to represent classical structures like trees, graphs, text indexes, etc., in the smallest possible space without challenging the functionality of the structure; the original operations should be supported efficiently without decompressing the whole structure.

One of the most commonly appearing structures are the *rank* and *select* dictionaries for bit vectors: $rank(A, i)$ gives the number of bits set up to position i in bit vector $A = a_1a_2\cdots a_n$, $a_k \in \{0, 1\}$; $select(A, j)$ is the inverse, giving the

* Funded by the Academy of Finland under grant 108219.
** Partially Funded by Fondecyt Grant 1-050493, Chile.

position i containing the j-th bit set in A. We study the dynamic version of these dictionaries, where one can *insert* or *delete* a bit at any position.

Dynamic *rank* and *select* dictionaries have been studied before [13, 9], as a special case of so-called *Searchable Partial Sums with Indels* problem. The best current result [9] requires $n + o(n)$ bits of space, $O(\log_b n)$ time for *rank* and *select*, and $O(b)$ amortized time for *insert* and *delete*, for $b = \Omega(\text{polylog}(n))$.

In this paper we improve some aspects of this result by achieving $O(\log n)$ worst-case time complexity for all the operations, over a data structure that requires $nH_0+o(n)$ bits of space, where $0 \leq H_0 \leq 1$ is the binary zero-order entropy of A. This space has been previously achieved only for static data structures [14], with constant time for *rank* and *select* but no support for updates. Ours is the *first* entropy-bound dynamic data structure answering *rank* and *select* queries. Moreover, our result works under weaker assumptions on the RAM model than the previous results on dynamic settings.

The indexed string matching problem is that of, given a long text $T[1, n]$ over an alphabet Σ of size σ, building a data structure called *full-text index* on it, to solve two types of queries: (a) Given a short pattern $P[1, m]$ over Σ, *count* the occurrences of P in T; (b) *locate* those *occ* positions in T. There are several classical full-text indexes requiring $O(n \log n)$ bits of space which can answer counting queries in $O(m \log \sigma)$ time (like suffix trees [1]) or $O(m + \log n)$ time (like suffix arrays [11]). Both locate each occurrence in constant time once the counting is done. Similar complexities are obtained with modern compressed data structures [6, 8, 7], requiring space $nH_k(T) + o(n \log \sigma)$ bits (for some small k), where $H_k(T) \leq \log \sigma$ is the k-th order empirical entropy of T.[1] These indexes are often called *entropy-compressed self-indexes* refering to their space requirement and to their ability to work without the text.

The main building block in entropy-compressed self-indexes is function *rank*, or more precisely, its generalization to non-binary sequences: $rank_c(A, i)$ counts the number of times symbol c appears in a given sequence A up to position i. Our dynamic entropy-compressed binary *rank* structure can be extended into a dynamic entropy-compressed *symbol rank* structure using *wavelet trees* [8]. This dynamic structure takes $nH_0 + o(n \log \sigma)$ bits of space, where H_0 is the empirical zero-order entropy of the sequence. It supports the same operations as binary *rank* with $O(\log \sigma)$ slowdown in queries. Plugging this structure in the dynamic self-index of Chan, Hon, and Lam [4], we obtain a dynamic entropy-compressed self-index occupying $nH_0 + o(n \log \sigma)$ bits on a text collection of overall length n. Our structure can count the number of occurrences of a pattern of length m in time $O(m \log n \log \sigma)$. Insertion of a text to the collection takes $O(\log n \log \sigma)$ time per symbol. Deletion takes $O(\log^2 n \log \sigma)$ time. These operations are $O(\log \sigma)$ times slower than with the original index of Chan et al., but we obtain a significant space saving: Their index takes $O(n\sigma)$ bits while ours takes $O(n \log \sigma)$ bits in general.

As a consequence, we obtain an $O(n \log n \log \sigma)$ time construction algorithm for a compressed self-index called *succinct suffix array* (SSA) [10] requiring

[1] In this paper log stands for $\log_2$.

$nH_0 + o(n \log \sigma)$ bits *working space* during construction (the same as the final structure). This is the first construction algorithm for a *FM-index* [6] variant, whose working space depends on the entropy. For another self-index called *LZ-index* [12], there is a recent entropy-bound construction algorithm [2].

2 Definitions

To simplify notation, we ignore roundings. When refering to number of bits, we use simply $\log n$ to refer to $\lfloor (\log n) + 1 \rfloor$. That is, $\log \log n$ bits means actually $\lfloor (\log \lfloor (\log n) + 1 \rfloor) + 1 \rfloor$ bits. Similarly $(\log n)/2$ is the integer nearest to $\lfloor (\log n) + 1 \rfloor / 2$, and so on.

We assume our sequence $A = a_1 \dots a_n$ to be drawn from an alphabet $\{0, 1, \dots \sigma - 1\}$. Let n_c denote the number of occurrences of symbol c in A, i.e., $n_c = |\{i \mid a_i = c\}|$. Then the zero-order *empirical entropy* is defined as $H_0(A) = \sum_{0 \le c < \sigma} \frac{n_c}{n} \log \frac{n}{n_c}$.

We assume a random access machine with word size w; typical arithmetic operations on w-bit integers are assumed to take constant time. We make the standard assumption that $\log n = \Theta(w)$ (in the full version, we show that this can be weakened to $\log n = O(w)$ without changing the results).

We study the following problems:
The *Dynamic Sequence with Indels* problem is to maintain a (virtual) sequence $A = a_1 \dots a_n$, $a_i \in \{0, 1, \dots, \sigma - 1\}$, supporting the operations:

- $rank_c(A, i)$ returns the number of occurrences of symbol c in $a_1 \cdots a_i$;
- $select_c(A, j)$ returns the index i containing j-th occurrence of c;
- $insert(A, c, i)$ inserts $c \in \{0, 1, \dots \sigma - 1\}$ between a_i and a_{i+1}; and
- $delete(A, i)$ deletes a_i from the sequence.

The *Dynamic Bit Vector with Indels* problem is a restriction of the above to alphabet $\{0, 1\}$. Then we use short-hand notation $rank(A, i) = rank_1(A, i)$ and $select(A, i) = select_1(A, i)$. Notice that $rank_0(A, i) = i - rank_1(A, i)$, but same does not apply for $select_0(A, j)$; we consider this case separately.

3 Previous Results

3.1 Entropy-Bound Structures for Bit Vectors

Raman et al. [14] proposed a data structure to solve *rank* and *select* queries in constant time over a static bit vector $A = a_1 \dots a_n$ with binary zero-order entropy H_0. The structure requires $nH_0 + o(n)$ bits.

The idea is to split A into *superblocks* $S_1 \dots S_{n/s}$ of $s = \log^2 n$ bits. Each superblock S_i is in turn divided into $2 \log n$ blocks $B_i(j)$, of $t = (\log n)/2$ bits each, thus $1 \le j \le s/t$. Each such block B_i is said to belong to *class* c if it has exactly c bits set, for $0 \le c \le t$. For each class c, a universal table G_c of $\binom{t}{c}$ entries is precomputed. Each entry corresponds to a possible block belonging to

class c, and it stores all the local *rank* answers for that block. Overall all the G_c tables add up $2^t = \sqrt{n}$ entries, and $O(\sqrt{n}\ \text{polylog}(n))$ bits.

Each block $B_i(j)$ of the sequence is represented by a pair $D_i(j) = (c, o)$, where c is its class and o is the index of its corresponding entry in table G_c. A block of class c thus requires $\log(c+1)+\log\binom{t}{c}$ bits. The first term is $O(\log\log n)$, whereas all the second terms add up $nH_0 + O(n/\log n)$ bits. To see this, note that $\log\binom{t}{c_1} + \log\binom{t}{c_2} \leq \log\binom{2t}{c_1+c_2}$, and that $nH_0 \geq \log\binom{t(n/t)}{c_1+\ldots+c_{n/t}}$. The pairs $D_i(j)$ are of variable length and are all concatenated into a single sequence.

Each superblock S_i stores a pointer P_i to its first block description in the sequence (that is, the first bit of $D_i(1)$) and the *rank* value at the beginning of the superblock, $R_i = rank(A, (i-1)s)$. P and R add up $O(n/\log n)$ bits. In addition, S_i contains s/t numbers $L_i(j)$, giving the initial position of each of its blocks in the sequence, relative to the beginning of the superblock. That is, $L_i(j)$ is the position of $D_i(j)$ minus P_i. Similarly, S_i stores s/t numbers $Q_i(j)$ giving the *rank* value at the beginning of each of its blocks, relative to the beginning of the superblock. That is, $Q_i(j) = rank(A, (i-1)s + (j-1)t) - R_i$. As those relative values are $O(\log n)$, sequences L and Q require $O(n \log\log n/\log n)$ bits.

To solve $rank(A, p)$, we compute the corresponding superblock $i = 1 + \lfloor p/s \rfloor$ and block $j = 1 + \lfloor (p - (i-1)s)/t \rfloor$. Then we add the *rank* value of the corresponding superblock, R_i, the relative *rank* value of the corresponding block, $Q_i(j)$, and complete the computation by fetching the description (c, o) of the block where p belongs (from bit position $P_i + L_i(j)$) and performing a (precomputed) local *rank* query in the universal table, $rank(G_c(o), p-(i-1)s-(j-1)t)$.

The overall space requirement is $nH_0 + O(n \log\log n/\log n)$ bits, and *rank* is solved in constant time. We do not cover *select* because it is not necessary to follow this paper.

3.2 Dynamic Structures for Bit Vectors

Hon et al. [9] show how to handle a bit vector $A = a_1 \ldots a_n$ in $n + o(n)$ bits of space, so that *rank* and *select* can be solved in $O(\log_b n)$ time, while insertions and deletions to the sequence can be handled in $O(b)$ amortized time, for any parameter $b = \Omega(\text{polylog}(n))$. Hence, they provide a solution to the *Dynamic Bit Vector with Indels* problem. Their main structure is a weight-balanced B-tree (WBB) [5, 13].

Our goal is to obtain $nH_0 + o(n)$ bits of space and $O(\log n)$ worst-case time for all the operations above. We build over a simplified version of their structure, which uses standard balanced trees and achieves $O(\log n)$ time and $O(n)$ bits of space [4]. We assume red-black trees in the following, as we later use the property of constant number of rotations to rebalance the tree.

Consider a balanced binary tree on A whose left-most leaf contains bits $a_1 a_2 \cdots a_{\log n}$, second left-most leaf contains bits $a_{\log n+1} a_{\log n+2} \cdots a_{2\log n}$, and so on. Each node v contains counters $p(v)$ and $r(v)$ telling the number of positions stored and the number of bits set in the subtree rooted at v, respectively. Note that this tree, with all its $\log n$-size pointers and counters, requires $O(n)$ bits.

To perform $rank(A, i)$, we enter the tree to find the leaf containing position i. We start with $rank \leftarrow 0$. If $p(left(v)) \geq i$ we enter the left subtree, otherwise we enter the right subtree with $i \leftarrow i - p(left(v))$ and $rank \leftarrow rank + r(left(v))$. In $O(\log n)$ time we reach the desired leaf and complete the rank query in $O(\log n)$ time by scanning the bit sequence corresponding to that node. For *select* we proceed similarly, except that the roles of $p()$ and $r()$ are reversed. For $select_0$ the computation is analogous.

Insertions and deletions are handled by entering to the correct leaf like in *rank*, and replacing its bit-sequence with the new content. Then the $p(v)$ and $r(v)$ counters in the path from the leaf to the root are changed accordingly. To keep the tree balanced, the leaves can be split and merged on updates: When a leaf is updated to contain $2 \log n$ bits, it is split into two leaves each containing $\log n$ bits. When a leaf is updated to contain $(\log n)/2$ bits, it is merged with its sibling. If this merging produces a leaf with more than $2 \log n - 1$ bits, this leaf is again split into two equal-size halves. After splitting and merging, the tree needs to be rebalanced and the counters updated in the nodes on the way to the root.

To obtain $n + o(n)$ bits of space instead of $O(n)$, we can use the superblock-block hierarchy from the previous section: The tree is built on the superblocks, i.e., each leaf corresponds to a $\log^2 n$-length superblock of A. A precomputed table G is used to answer *rank* queries for each $(\log n)/2$-length bit-sequence. Then one can scan through the $\log^2 n$-length superblock summing up *rank* answers to each $(\log n)/2$-length block in constant time until reaching the block containing the query position. The remaining bits can be read one-by-one to complete the *rank* query inside a superblock in $O(\log n)$ time. Answering *select* is similar. The problem, however, is that we cannot allocate $2 \log^2 n$ space for a superblock that will hold only $\log^2 n$ bits, as otherwise we could spend as much as $2n$ bits for the blocks. To obtain $n + o(n)$ space one must force very tight usage of the leaf space: spending $(1 + \epsilon) \log^2 n$ bits, for any constant $\epsilon > 0$, is forbidden. This is problematic because bit insertions on a leaf would cause an overflow propagation to the next leaves that cannot be fixed with a constant number of block splits. The complete solution is quite involved, and we present it in the next sections, already coupled with the technique to achieve $nH_0 + o(n)$ bits (the reader can easily simplify it to obtain the $n + o(n)$ bits solution that already improves [9] in some aspects). We must also pay attention to the case where $\log n$ changes.

4 Dynamic Entropy-Bound Structures for Bit Vectors

We design a data structure to represent a bit sequence $A = a_1 \dots a_n$ of binary zero-order entropy H_0, using $nH_0 + o(n)$ bits of space and performing operations *rank*, *select*, *insert* and *delete* all in $O(\log n)$ time. Hence, we show that the *Dynamic Bit Vector with Indels* problem can be solved using less than $\Theta(n)$ space on compressible sequences, without sacrificing the logarithmic time bound on the operations.

4.1 High-level hierarchy

We maintain the universal tables G_c as in Section 3.1, but this time they store only the explicit content of the blocks. This still requires $O(\sqrt{n}\ \mathrm{polylog}(n))$ bits of space.

We also divide A into blocks and superblocks, except that this time superblocks do not span a constant amount of bits of A, but of its (compressed) representation. That is, each superblock S will maintain $s = f(n)\log n$ bits (for some $f(n) = O(\mathrm{polylog}(n))$ to be determined later), and this will correspond to as many (complete) blocks as can be represented with s bits considering their D, L, and Q entries. Blocks are still of $t = (\log n)/2$ bits. Since each L and Q value requires $O(\log\log n)$ bits, and a D entry may require up to $t + O(\log\log n)$ bits, a superblock may handle from $O(f(n))$ to $O(f(n)\log n/\log\log n)$ blocks. Similarly, a block can have up to $O(\log n)$ unused space, because the next block does not fit in it. This unused space adds up $O(n/f(n))$ bits overall. Otherwise the space usage is the same as in the static case.

4.2 Operations inside a superblock

A $rank(S_i, p)$ query inside a superblock is handled in $O(\log n)$ time by adding the corresponding $Q_i(j)$ entry to $rank(G_c(o), p')$, where $j = 1 + \lfloor p/t \rfloor$, $p' = p - (j-1)t$, and (c, o) is found at position $L_i(j)$ in the memory area of the superblock. Here, $rank(G_c(o), p')$ is computed in $O(\log n)$ time by a bitwise scan over $G_c(o)$. A $select(S_i, p)$ query is solved in time $O(\log n)$ by binary searching Q_i for the largest $rank$ value not exceeding p, and then a bitwise scan for query $select(G_c(o), p')$. Computation for $select_0$ is analogous.

To insert a bit q at position p of S_i, we essentially recompute the superblock by brute force. However, we must be careful so as to work only $O(f(n))$ time per superblock. For example, we cannot decompress, modify, and then recompress the superblock because that way we could work $O(f(n)\log n/\log\log n)$ time (as the uncompressed superblock can be up to $O(f(n)\log^2 n/\log\log n)$ bits long).

We first determine the block j where the insertion is to take place, that is, $j = 1 + \lfloor p/t \rfloor$. All the $D_i(1 \ldots j-1)$, $L_i(1 \ldots j)$, and $Q_i(1 \ldots j)$ entries are direcly copied into a new memory area where the updated representation of S_i is to be built. On a RAM machine this copying can be done in $O(f(n))$ time.

Modifying each block in constant time. The block $D_i(j) = (c, o)$ to modify starts at position $L_i(j)$ within the superblock. We use $G_c(o)$ to obtain the uncompressed content of this block. Let $B = b_1 \ldots b_t$ be the bits of this block, and let $p' = p - (j-1)t$ be the position to insert the bit q within B. Thus we compute $B' = b_1 \ldots b_{p'-1} q b_{p'+1} \ldots b_{t-1}$ and save b_t for later. To compress B' in constant time we use another universal table H, which is indexed by numbers of t bits and stores, at each entry, the c and o value of the corresponding binary vector. H requires $O(\sqrt{n}\ \mathrm{polylog}(n))$ bits, and gives $H(B') = (c', o')$ in constant time. This description $D_i(j)' = (c', o')$ is appended at the updated copy of S_i we are constructing.

We must now take care of the remaining blocks to the right. We have a bit b_t that fell off B. In addition we must shift the values L_i to the right by $|o'| - |o|$ and Q_i by $q - b_t$. To perform all this propagation in $O(f(n))$ time, we use yet another universal table $J(b, l, q, x)$, where b is a bit to insert at the beginning of the next block, $l = O(\text{polylog}(n))$ is the next L_i value, $q = O(\text{polylog}(n))$ is the next Q_i value, and x is the sequence of the first t bits of $D_i(j+1\ldots)$. If $J(b, l, q, x) = (D', L', Q', b', l', q')$, this means that, if we decode from x as many integral blocks as we can, append bit b at the beginning, and recode them, we obtain sequence D'. Their corresponding positions, starting in l, are encoded in L', and their corresponding ranks, starting at q, are encoded in Q'. Furthermore, bit b' falls off at the end of D', the next L_i value should be l', and the next Q_i value should be q'. Another table $V(x) = r'$ tells us how many bits we could use from x, so we can advance in the processing of sequence D_i by r' bits.

Therefore, after having modified the j-th block, we start by assigning $r = L_i(j)$ and obtain $J(b_t, L_i(j) + |o'| - |o|, Q_i(j) + q - b_t, D_i[r\ldots])$ and $V(D_i[r\ldots])$. Then we copy D', L', and Q' to the updated version of S_i we are building, and continue with $J(b', l', q', D_i[r + r'\ldots])$ and $V(D_i[r + r'\ldots])$, until processing the whole superblock. At the end, we rewrite S with its updated version. Note that we still have one overflown bit.

Tables J and V require $O(\sqrt{n}\ \text{polylog}(n))$ bits, and they process $\Theta(\log n)$ bits of the superblock in constant time (each two applications it must be possible to process at least t bits of D_i), plus the time necessary to write the modified superblock. As there are $O(f(n)\log n)$ bits in the superblock, we can process the whole superblock in $O(f(n))$ time using J and V, plus the size of the new superblock measured in $\Theta(\log n)$-size chunks.

Let us consider how much can the superblock grow by the insertion of a single bit. If a new block is started, we need $O(\log\log n)$ more bits. In addition, the D entry of a block may grow because its (c, o) descriptor changes. The maximum value of $\log\binom{t}{c+1} - \log\binom{t}{c}$ is $\log t$, achieved when $c = 0$. Propagated over $O(f(n)\log n/\log\log n)$ blocks, the sequence of D values might be increased by $O(f(n)\log n)$ bits. This is as large as a whole superblock, and means that a single bit insertion might double the size of the superblock in some extreme cases. For example, if the sequence is $(0^t1^t)^r$, all the c values will be 0 or t, and the o indexes will be empty, thus we will store $f(n)\log n/\log\log n$ blocks in the superblock. If we now insert a 1 at the beginning of the sequence, each o descriptor becomes $\log t = O(\log\log n)$ bits wide, which adds up $f(n)\log n$ extra bits. Still, the new superblock is also $O(f(n)\log n)$ size and can be output using J and V in $O(f(n))$ time.

Overflow to the next superblock. At the end of the operation, it might be that the new sequence does not fit within the s bits allocated to the superblock. If so, we take out as many blocks as necessary from the end of the superblock, so as to move them to the beginning of the next superblock. We have seen that we might have to move up to $O(f(n)\log n)$ bits. In addition we must insert the excess bit at the next superblock (after the blocks we are moving, if any).

The process completely rewrites the next superblock S'. We move the overflowing D, L and Q entries to the beginning of S', but the L and Q values moved must be shifted. This has to be done by chunks of $\Theta(\log n)$ bits using a universal table to ensure $O(f(n))$ overall time. Then we must insert the carry bit at the beginning of the original entries of S', which in addition must be shifted to account for the blocks moved from the overflowing superblock. This can be carried out in $O(f(n))$ time using tables J and V. Yet, this bit insertion may produce another $O(f(n)\log n)$-bits overflow, in addition to the original $O(f(n)\log n)$ bits. The exponential growth is avoided because we can create a new superblock as soon as we have enough overflown bits. The propagation can thus be carried out in $O(f(n))$ time per superblock rewritten/created. Yet, we still need a mechanism to prevent that the propagation continues too far.

Limiting the propagation of overflows. Every $f(n)$ superblocks we permit the formation of a *partial* superblock, which reserves $f(n)\log n$ bits but might be partially full, and in addition permits having at the end an underfilled block (with less than t bits). This partial block needs some care to be correctly handled, such as padding it with dummy bits to obtain a representation in G, taking care of its real length, and so on. Partial superblocks waste $O(n/f(n))$ bits overall, and ensure that we never traverse more than $f(n)$ superblocks in the overflow process. Thus the overall insertion work is $O(f(n)^2)$.

To ensure the desired density of partial superblocks, we first check whether there is a partial superblock among the next $2f(n)$ superblocks. If there is one, we carry out the propagation up to it. Otherwise, we propagate $f(n)$ superblocks and create a new partial superblock. In both cases we work over $O(f(n))$ superblocks, and guarantee that every partial superblock is $f(n)$ superblocks away from any other. We note that partial superblocks may end up overflowing, at which point they are not considered partial anymore. We can create a new partial superblock immediately following it, as it is already ensured that the new partial superblock is far away from others.

Note that, when a partial superblock overflows, its last block can still be partial. This is not a real problem, because we are creating next a new partial superblock containing that partial block at the end, plus sufficient complete blocks from the end of the overflowing superblock.

Controlling the underflow. For deletions we proceed similarly, using a table J' very similar to J: J' deletes the first bit of the blocks represented by x and adds bit b at their end. The bit b we give to J' is obtained in constant time using G, as the first bit of $D_i[r + V(x)...]$. Also, we ensure that superblocks are as full as possible. If some space is left at the end of the superblock, we check that the first blocks from the next superblock can be moved back, and propagate the underflow similarly as the overflows. If we reach a partial superblock, no further propagation of underflows is necessary. If after $2f(n)$ attempts we do not reach a partial superblock, we permit the underflow at the $f(n)$-th superblock and

declare it partial. On the other hand, a partial superblock that gets empty must be deleted.

Note that, because of the changes in $|o|$ widths, an insertion can actually produce an underflow and a deletion can produce an overflow. This is not problematic. Overall (still not considering how to manage superblocks), we have $O(n/f(n))$ extra space and $O(f(n)^2)$ insertion/deletion time. We can choose, for example, $f(n) = \sqrt{\log n}$ to obtain $O(\log n)$ time and $O(n/\sqrt{\log n})$ space.

4.3 Global *rank* and *select*

We have seen how to perform *rank* and *select* inside a superblock in $O(\log n)$ time. To perform the global *rank* and *select* we can use the balanced tree on the superblocks as explained in Section 3.2. Finding the correct superblock takes $O(\log n)$ time, hence the whole query takes $O(\log n)$ time.

Inserting and deleting bits from this tree requires rewriting the $p()$ and $r()$ values from the affected superblock(s) through the root. Creation and deletion of superblocks and internal tree nodes is easily handled together with the maintenance of $r()$ and $p()$. We note, however, that we permit that a single update affects $O(f(n))$ superblocks. Once the leaf to be inserted or deleted is located, the red-black tree needs constant time to rebalance, so this adds up $O(f(n))$ time per insertion. As for propagating the red-black coloring and updating the $r()$ and $p()$ values through the root, note that those $O(f(n))$ superblocks are contiguous in the tree and therefore their total number of ancestors do not exceed $f(n) + O(\log n) = O(\log n)$. It is not hard to organize the updates to work $O(\log n)$ time overall.

4.4 Changing $\log n$

Our result so far assumes that $\log n$ stays constant during the operations. This value fixes the superblock/block hierarchy and the global preprocessed tables. This assumption can be removed in two ways: (1) performing a global rebuild whenever $\log n$ changes; (2) maintaining partial structures ready for values $(\log n) - 1$, $\log n$, and $(\log n) + 1$ (which we call the *previous*, *current*, and *next*).

Approach (1) is easy to implement. We can rebuild all structures in $O(n)$ time when necessary to accommodate the new value of $\log n$. Amortized over all insertions and deletions, this costs only $O(1)$ time per operation.

Approach (2) is more complex but is inspired on a standard mechanism to convert amortized complexity into worst-case complexity. The idea is to split the current elements among the *previous*, *current*, and *next* structures, so that the first elements are in *previous*, the last are in *next*, and *current* holds the middle elements. It is trivial to run *rank* and *select* queries on this split structure. Initially, all the elements are in *current*, and the other two are empty. Upon an insertion, the size of *next* must grow by 2 and *previous* must shrink by 1 unless it is already empty; a deletion must cause the opposite effect; and *current* acts as a variable-size buffer.

To achieve this, let us denote $x \to y$ or $x \leftarrow y$ the movement of one element among structures, for $x, y \in \{p, c, n\}$, e.g. $p \leftarrow c$ means moving the first element of *current* to *previous*. If the source structure is empty, the movement is just ignored. Then, we insert (delete) in the proper structure and then, depending on where the insertion (deletion) point lies, we move elements as follows:

- *previous*: $p \to c$, $p \to c$, $c \to n$, $c \to n$ ($c \leftarrow n$, $c \leftarrow n$, $p \leftarrow c$, $p \leftarrow c$).
- *current*: $p \to c$, $c \to n$, $c \to n$ ($c \leftarrow n$, $c \leftarrow n$, $p \leftarrow c$).
- *next*: $p \to c$, $c \to n$ ($c \leftarrow n$, $p \leftarrow c$).

It is easy to see that, after n net insertions, *next* will hold all the $2n$ elements, and that after $n/2$ net deletions, *prev* will hold all the $n/2$ remaining elements. This is true even if the insertions and deletions are intermixed. When *next* holds all the elements, it becomes *current* and the new *previous* and *next* structures are empty; similarly when *previous* holds all the elements. At those points, precisely, $\log n$ has changed its value. The space requirement is still $nH_0 + o(n)$.

The only remaining problem is that we do not have time to build the new G, J, etc. tables, as we would need them immediately available to handle the new *next* or *previous* structure. For this sake, we maintain all the time 5 versions of those tables, for $(\log n) - 2 \ldots (\log n) + 2$. As we move to $(\log n) + 1$, we have immediately available the required tables for $(\log n)$, $(\log n) + 1$ and $(\log n) + 2$. The construction of the tables for $(\log n) + 3$ is easily spread during the next $O(\sqrt{n}$ polylog $n)$ insertions, building just a new cell at the time. These insertions are much less than the necessary to make $\log n$ grow again. If, instead, $\log n$ shrinks back, we just abandon the partial table construction. Thus we achieve the following result:

Theorem 1. *The Dynamic Bit Vector with Indels problem can be solved using* $nH_0 + O(n/\sqrt{\log n})$ *bits of space supporting the operations rank, select, insert, and delete in* $O(\log n)$ *worst-case time.*

5 Extensions and Applications

5.1 General alphabets

Theorem 1 can be extended to the *Dynamic Sequence with Indels* problem using wavelet trees [8]. The wavelet tree is a balanced binary tree built on the alphabet symbols, containing bit vectors in its internal nodes. When these node bit vectors are preprocessed for the *Dynamic Bit Vector with Indels* problem (taking some care on the sub-linear terms [7]), we obtain the following result.

Theorem 2. *The Dynamic Sequence with Indels problem can be solved using* $nH_0 + o(n \log \sigma)$ *bits of space supporting the operations rank, select, insert, and delete, in* $O(\log n \log \sigma)$ *worst-case time. Here* H_0 *is the zero-order entropy of the sequence and* σ *its alphabet size.*

5.2 Dynamic Full-Text Indexes

Chan, Hon, and Lam [4] show how to use a solution to *Dynamic Sequence with Indels* problem to obtain a dynamic full-text index. The idea is to simulate the *backward search* algorithm of Ferragina and Manzini [6]: After preprocessing a text T, the backward search algorithm finds the number of occurrences of a given pattern P in T in $O(|P|)$ steps. One step essentially makes two *rank* queries to the *Burrows-Wheeler* transform [3] of T, $A = bwt(T)$. We note $H_0(A) = H_0(T)$ as the transform is a permutation.

They [4] show that one can dynamically maintain a collection of texts, by keeping a data structure supporting *rank*, *insert* and *delete* on the Burrows-Wheeler transform of the concatenation of the texts in the collection (symbol 0 is reserved for separating two texts). We can as a black box replace their `COUNT` structure (that takes $O(n\sigma)$ bits, supporting the operations in $O(\log n)$ time) with the structure in Theorem 2 to obtain the following result.

Theorem 3. *A dynamic collection of texts $\mathcal{C} = \{T_1, T_2, \ldots, T_m\}$, where each $T_i \in \{1, 2, \ldots \sigma - 1\}^*$, can be maintained in $nH_0(\mathcal{C}) + o(n \log \sigma)$ bits supporting counting of occurrences of a pattern P in $O(|P| \log n \log \sigma)$ time, inserting a text T in $O(|T| \log n \log \sigma)$ time, and deleting a text T in $O(|T| \log^2 n \log \sigma)$ time. Here n is the length of concatenation $C = 0T_1 0T_2 \cdots 0T_m$ of $\mathcal{C}$, and $H_0(\mathcal{C}) = H_0(C)$. We assume that $\mathcal{C}$ starts initially empty.*

The index can be extended to support reporting the occurrences using the `MARK` structure of [4]. This structure takes $O(n)$ bits, and with our *rank* structure can be used to report each occurrence in $O(\log^2 n \log \sigma)$ time.

As a consequence, we obtain an $O(n \log n \log \sigma)$ time construction algorithm for a compressed self-index requiring $nH_0 + o(n \log \sigma)$ bits *working space* during construction: This is obtained by just inserting text T to the empty collection. This index can be converted to a more efficient static self-index, like a *succinct suffix array* [10], within the same time bound. The static structure requires the same $nH_0 + o(n \log \sigma)$ bits, but the counting of pattern occurrences can then be done in $O(|P|)$ time if $\sigma = O(\text{polylog}(n))$, and $O(|P| \log \sigma / \log \log n)$ in general.

6 Conclusions

We have introduced the *first* entropy-bound dynamic data structure answering *rank* and *select* queries on bit arrays. We can represent a vector of n bits with zero-order entropy H_0 using $nH_0 + o(n)$ bits of space, so that we can answer *rank* and *select* queries, as well as inserting and deleting bits, in $O(\log n)$ worst-case time. This improves in several aspects the best existing solution to the *Searchable Partial Sums with Indels Problem* [9] for the case of bit sequences: we achieve logarithmic worst-case bounds for insertions and deletions (previous solution achieved $\Theta(\text{polylog}(n))$ amortized time) and require less than n bits on compressible sequences. We apply these results to compressed full-text self-indexing, achieving the first FM-index-like structure that can be built within

zero-order entropy space. This index permits insertion and deletion of texts with better bounds than previous solutions [4].

Our result works under weaker assumptions on the RAM model than the previous results on dynamic settings. We assumed $\log n = \Theta(w)$ to simplify matters; in the full version, this assumption will be loosened to $\log n = O(w)$. This complicates the memory allocation, as we can not e.g. represent tree pointers in $O(\log n)$ bits, when $\log n = o(w)$. However, our results remain unchanged under the weaker model.

The succinct suffix array (SSA) constructed using the dynamic index can currently only count the pattern occurrences, unless paying $O(n)$ bits extra space for the `MARK` structure of [4]. We plan to study whether this could be improved to $o(n)$ bits. We plan also to study general searchable partial sums, and larger alphabets with multiary wavelet trees [7] to improve the time bounds in Theorem 2 by a $\log \log n$ factor.

References

1. A. Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, NATO ISI Series, pages 85–96. Springer-Verlag, 1985.
2. D. Arroyuelo and G. Navarro. Space-efficient construction of LZ-index. In *Proc. ISAAC'05*, LNCS 3827, pages 1143–1152, 2005.
3. M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report Technical Report 124, Digital Equipment Corporation, 1994.
4. W.-L. Chan, W.-K. Hon, and T.-W. Lam. Compressed index for a dynamic collection of texts. In *Proc. CPM'04*, LNCS 3109, pages 445–456, 2004.
5. P. Dietz. Optimal algorithms for list indexing and subset rank. In *Proc. WADS'89*, pages 39–46, 1989.
6. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. FOCS'00*, pages 390–398, 2000.
7. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representation of sequences and full-text indexes. *ACM Transactions on Algorithms*, 2006. To appear. Preliminary versions in *Proc. SPIRE 2004* and Tech. Rep. TR/DCC-2004-5, Dept. of Computer Science Univ. of Chile, `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/sequences.ps.gz`.
8. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA'03*, pages 841–850, 2003.
9. W.-K. Hon, K. Sadakane, and W.-K. Sung. Succinct data structures for searchable partial sums. In *Proc. ISAAC'03*, LNCS 2906, pages 505–516, 2003.
10. V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.
11. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, pages 935–948, 1993.
12. G. Navarro. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms (JDA)*, 2(1):87–114, 2004.
13. R. Raman, V. Raman, and S. Srinivasa Rao. Succinct dynamic data structures. In *Proc. WADS'01*, pages 426–437, 2001.
14. R. Raman, V. Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. SODA'02*, pages 233–242, 2002.