Succinct Representations of Dynamic Strings*

Meng He and J. Ian Munro

Cheriton School of Computer Science, University of Waterloo, Canada,
{mhe, imunro}@uwaterloo.ca

Abstract. The `rank` and `select` operations over a string of length n from an alphabet of size σ have been used widely in the design of succinct data structures. In many applications, the string itself must be maintained dynamically, allowing characters of the string to be inserted and deleted. Under the word RAM model with word size $w = \Omega(\lg n)$, we design a succinct representation of dynamic strings using $nH_0 + o(n) \cdot \lg \sigma + O(w)$ bits to support `rank`, `select`, `insert` and `delete` in $O(\frac{\lg n}{\lg \lg n}(\frac{\lg \sigma}{\lg \lg n} + 1))$ time[1]. When the alphabet size is small, i.e. when $\sigma = O(\texttt{polylog}(n))$, including the case in which the string is a bit vector, these operations are supported in $O(\frac{\lg n}{\lg \lg n})$ time. Our data structures are more efficient than previous results on the same problem, and we have applied them to improve results on the design and construction of space-efficient text indexes.

1 Introduction

Succinct data structures provide solutions to reduce the storage cost of modern applications that process large data sets, such as web search engines, geographic information systems, and bioinformatics applications. First proposed by Jacobson [1], the aim is to encode a data structure using space close to the information-theoretic lower bound, while supporting efficient navigation operations in them. This approach was successfully applied to many abstract data types, including bit vectors [1–3], strings [4–6], binary relations [7, 5], (unlabeled and labeled) trees [1, 8, 9, 7, 5, 10], graphs [1, 8, 11] and text indexes [4, 12, 6].

A basic building block for most succinct data structures is the pair of operations `rank` and `select`. In particular, we require a highly space-efficient representation of a string S of length n over an alphabet of size σ to support the fast evaluation of the following operations:

- $\texttt{access}(S, i)$, which returns the character at position i in the string S;
- $\texttt{rank}_\alpha(S, i)$, which returns the number of occurrences of character α in $S[1..i]$;
- $\texttt{select}_\alpha(S, i)$, which returns the position of the i^{th} occurrence of character α in the string S.

* This work was supported by NSERC of Canada and the Canada Research Chairs program.

[1] $\lg n$ denotes $\log_2 n$.

This problem has many applications such as designing space-efficient text indexes [4, 6], as well as representing binary relations [7, 5], labeled trees [9, 7, 5] and labeled graphs [11]. The case in which the string is a bit vector whose characters are 0's and 1's (i.e. $\sigma = 2$) is even more fundamental: A bit vector supporting `rank` and `select` is a key structure used in several approaches of representing strings succinctly [4, 7, 5, 13], and it is also used in perhaps most succinct data structures [1, 8, 9, 7, 5, 11].

Due to the importance of strings and bit vectors, researchers have designed various succinct data structures for them [1, 3–6] and achieved good results. For example, the data structure of Raman *et al.* [3] can encode a bit vector using $nH_0 + o(n)$ bits, where H_0 is the zero-order entropy of the bit vector[2], to support `access`, `rank` and `select` operations in constant time. Another data structure called wavelet tree proposed by Grossi *et al.* [4] can represent a string using $nH_0 + o(n) \cdot \lg \sigma$ bits to support `access`, `rank` and `select` in $O(\lg \sigma)$ time.

However, in many applications, it is not enough to have succinct static data structures that allow data to be retrieved efficiently, because data in these applications are also updated frequently. In the case of strings and bit vectors, the following two update operations are desired in many applications in addition to `access`, `rank` and `select`:

- $\texttt{insert}_\alpha(S, i)$, which inserts character α between $S[i-1]$ and $S[i]$;
- $\texttt{delete}(S, i)$, which deletes $S[i]$ from S.

In this paper, we design succinct representations of dynamic strings and bit vectors that are more efficient than previous results. We also present several applications to show how advancements on these fundamental problems yield improvements on other data structures.

1.1 Related Work

Blandford and Blelloch [14] considered the problem of representing ordered lists succinctly, and their result can be used to represent a dynamic bit vector of length n using $O(nH_0)$ bits to support the operations defined in Section 1 in $O(\lg n)$ time (note that $H_0 \leq 1$ holds for a bit vector). A different approach proposed by Chan *et al.* [12] can encode dynamic bit vectors using $O(n)$ bits to provide the same support for operations. Later Chan *et al.* [15] improved this result by providing $O(\lg n / \lg \lg n)$-time support for all these operations while still using $O(n)$ bits of space. Mäkinen and Navarro [13] reduced the space cost to $nH_0 + o(n)$ bits, but their data structure requires $O(\lg n)$ time to support operations. Recently, Sadakane and Navarro [10] designed a data structure for dynamic trees, and their main structure is essentially a bit vector that supports

[2] The zero-order (empirical) entropy of a string of length n over an alphabet of size σ is defined as $H_0 = \sum_{i=1}^{\sigma} (\frac{n_i}{n} \lg \frac{n}{n_i})$, where n_i is the number of times that the i^{th} character occurs in the string. Note that we always have $H_0 \leq \lg \sigma$. This definition also applies to a bit vector, for which $\sigma = 2$.

more types of operations. Their result can be used to represent a bit vector using $n + o(n)$ bits to support the operations we consider in $O(\lg n / \lg\lg n)$ time.

For the more general problem of representing dynamic strings of length n over alphabets of size σ, Mäkinen and Navarro [13] combined their results on bit vectors with the wavelet tree structure of Grossi *et al.* [4] to design a data structure of $nH_0 + o(n) \cdot \lg\sigma$ bits that supports `access`, `rank` and `select` in $O(\lg n \log_q \sigma)$ time, and `insert` and `delete` in $O(q \lg n \log_q \sigma)$ time for any $q = o(\sqrt{\lg n})$. Lee and Park [16] proposed another data structure of $n\lg\sigma + o(n) \cdot \lg\sigma$ to support `access`, `rank` and `select` in $O(\lg n(\frac{\lg\sigma}{\lg\lg n} + 1))$ worst-case time which is faster, but `insert` and `delete` take $O(\lg n(\frac{\lg\sigma}{\lg\lg n}+1))$ amortized time. Finally, González and Navarro [6] improved the above two results by designing a structure of $nH_0 + o(n)\cdot\lg\sigma$ bits to support all the operations in $O(\lg n(\frac{\lg\sigma}{\lg\lg n}+1))$ worst-case time.

Another interesting data structure is that of Gupta *et al.* [17]. For the same problems, they aimed at improving query time while sacrificing update time. Their bit vector structure occupies $nH_0 + o(n)$ bits and requires $O(\lg\lg n)$ time to support `access`, `rank` and `select`. It takes $O(n^\epsilon)$ amortized time to support `insert` and `delete` for any constant $0 < \epsilon < 1$. Their dynamic string structure uses $n\lg\sigma + \lg\sigma(o(n) + O(1))$ bits to provide the same support for operations (when $\sigma = O(\texttt{polylog}(n))$, `access`, `rank` and `select` take $O(1)$ time).

1.2 Our Results

We adopt the word RAM model with word size $w = \Omega(\lg n)$. Our main result is a succinct data structure that encodes a string of length n over an alphabet of size σ in $nH_0 + o(n) \cdot \lg\sigma + O(w)$ bits to support `access`, `rank`, `select`, `insert` and `delete` in $O(\frac{\lg n}{\lg\lg n}(\frac{\lg\sigma}{\lg\lg n} + 1))$ time. When $\sigma = O(\texttt{polylog}(n))$, all these operations can be supported in $O(\frac{\lg n}{\lg\lg n})$ time. Note that the $O(w)$ term in the space cost exists in all previous results, and we omit them in Section 1.1 for simplicity of presentation (in fact many papers simply ignore them). Our structure can also encode a bit vector of length n in $nH_0 + o(n) + O(w)$ bits to support the same operations in $O(\frac{\lg n}{\lg\lg n})$ time, matching the lower bound in [18]. Our solutions are currently the best to the problem, for both the general case and the special case in which the alphabet size is $O(\texttt{polylog}(n))$ or 2 (i.e. the string is a bit vector). The only previous result that is not comparable is that of Gupta *et al.* [17], since their solution is designed under the assumption that the string is queried frequently but updated infrequently.

We also apply the above results to design a succinct text index for a dynamic text collection to support text search, and the problem of reducing the required amount of working space when constructing a text index. Our dynamic string representation allows us to improve previous results on these problems [13, 6].

2 Preliminaries

Searchable Partial Sums. Raman *et al.* [19] considered the problem of representing a dynamic sequence of integers to support `sum`, `search` and `update`

operations. To achieve their main result, they designed a data structure for the following special case in which the length of the sequence is small, and we will use it to encode information stored as small sequences of integers in our data structures:

Lemma 1. *There is a data structure that can store a sequence, Q, of $O(\lg^{\epsilon} n)$ nonnegative integers of $O(\lg n)$ bits each*[3]*, for any constant $0 \leq \epsilon < 1$, using $O(\lg^{1+\epsilon} n)$ bits to support the following operations in $O(1)$ time:*

- $\texttt{sum}(Q, i)$*, which computes* $\sum_{j=1}^{i} Q[j]$*;*
- $\texttt{search}(Q, x)$*, which returns the smallest i such that* $\texttt{sum}(Q, i) \geq x$*;*
- $\texttt{update}(Q, i, \delta)$*, which updates $Q[i]$ to $Q[i] + \delta$, where $|\delta| \leq \lg n$.*

The data structure can be constructed in $O(\lg^{\epsilon} n)$ time, and it requires a precomputed universal table of size $O(n^{\epsilon'})$ bits for any fixed $\epsilon' > 0$.

Collections of Searchable Partial Sums. A key structure in the dynamic string representation of González and Navarro [6] is a data structure that maintains a set of sequences of nonnegative integers, such that `sum`, `search` and `update` can be supported on any sequence efficiently, while `insert` and `delete` are performed simultaneously on all the sequences at the same given position, with the restriction that only 0's can be inserted or deleted. More precisely, let $C = Q_1, Q_2, \cdots, Q_d$ to be a set of dynamic sequences, and each sequence, Q_j, has n nonnegative integers of $k = O(\lg n)$ bits each. The *collection of searchable partial sums with insertions and deletions (CSPSI)* problem is to encode C to support:

- $\texttt{sum}(C, j, i)$, which computes $\sum_{p=1}^{i} Q_j[p]$;
- $\texttt{search}(C, j, x)$, which returns the smallest i such that $\texttt{sum}(C, j, i) \geq x$;
- $\texttt{update}(C, j, i, \delta)$, which updates $Q_j[i]$ to $Q_j[i] + \delta$;
- $\texttt{insert}(C, i)$, which inserts 0 between $Q_j[i-1]$ and $Q_j[i]$ for all $1 \leq j \leq d$;
- $\texttt{delete}(C, i)$, which deletes $Q_j[i]$ from sequence Q_j for all $1 \leq j \leq d$, and to perform this operation, $Q_j[i] = 0$ must hold for all $1 \leq j \leq d$.

González and Navarro [6] designed a data structure of $kdn(1 + O(\frac{1}{\sqrt{\lg n}} + \frac{d}{\lg n}))$ bits to support all the above operations in $O(d + \lg n)$ time. This structure becomes succinct (i.e. using $dk(n + o(n))$ bits if $d = o(\lg n)$, when the operations can be supported in $O(\lg n)$ time. A careful reading of their technique shows that these results only work under the word RAM model with word size $w = \Theta(\lg n)$ (see our discussions after Lemma 5). We improve this data structure for small d, which is further used to design our succinct string representations.

[3] Raman *et al.* [19] required each integer to fit in one word. However, it is easy to verify that their proof is still correct if each integer requires $O(\lg n)$ bits, i.e. each integer can require up to a constant number of words to store.

3 Collections of Searchable Partial Sums

We follow the main steps of the approach of González and Navarro [6] to design a succinct representation of dynamic strings, but we make improvements in each step. We first improve their result for the CSPSI problem (Section 3), and then combine it with other techniques to improve their data structure for strings over small alphabets (Section 4). Finally, we extend the result on small alphabets to general alphabets (Section 5). Our main strategy of achieving these improvements is to divide the sequences into superblocks of appropriate size, and store them in the leaves of a B-tree (instead of the red-black tree in [6]). Similar ideas were applied to data structures for balanced parentheses [15, 10]. Our work is the first that successfully adapts it to integer sequences and character strings, and we have created new techniques to overcome some difficulties. Proofs omitted from this paper due to space constraints can be found in the full version [20].

In this section, we consider the CSPSI problem defined in section 2. We assume that $d = O(\lg^{\eta} n)$ for any constant $0 < \eta < 1$, and for the operation $\texttt{update}(C, j, i, \delta)$, we assume $|\delta| \leq \lg n$. Under these assumptions, we improve the result in [6] under the word RAM model with word size $\Omega(\lg n)$.

Data Structures. Our main data structure is a B-tree constructed over the given collection C. Let $L = \lceil \frac{\lceil \lg n \rceil^2}{\lg \lceil \lg n \rceil} \rceil$. Each leaf of this B-tree stores a *superblock* whose size is between (and including) $L/2$ and $2L$ bits, and each superblock stores the same number of integers from each sequence in C. More precisely, the content of the leftmost leaf is $Q_1[1..s_1]Q_2[1..s_1]\cdots Q_d[1..s_1]$, the content of the second leftmost leaf is $Q_1[s_1+1..s_2]Q_2[s_1+1..s_2]\cdots Q_d[s_1+1..s_2]$, and so on, and the indices $s_1, s_2, \cdots$ satisfy the following conditions because of requirement on the sizes of superblocks: $L/2 \leq s_1 kd \leq 2L, L/2 \leq (s_2 - s_1)kd \leq 2L, \cdots$.

Let $f = \lg^{\lambda} n$, where λ is a positive constant number less than 1 that we will fix later. Each internal node of the B-tree we construct has at least f and at most $2f$ children. We store the following $d+1$ sequences of integers for each internal node v (let h be the number of children of v):

- A sequence $P(v)[1..h]$, in which $P(v)[i]$ is the number of positions stored in the leaves of the subtree rooted at the i^{th} child of v for any sequence in C (note that this number is the same for all sequences in C);
- A sequence $R_j(v)[1..h]$ for each $j = 1, 2, \cdots, d$, in which $R_j(v)[i]$ is the sum of the integers from sequence Q_j that are stored in the leaves of the subtree rooted at the i^{th} child of v.

We use Lemma 1 to encode each of the $d+1$ sequences of integers for v.

We further divide each superblock into *blocks* of $\lceil \lceil \lg n \rceil^{3/2} \rceil$ bits each, and maintain the blocks for the same superblock using a linked list. Only the last block in the list can be partially full; any other block uses all its bits to store the data encoded in the superblock. This is how we store the superblocks physically.

To analyze the space cost of the above data structures, we have:

Lemma 2. *The above data structures occupy $kd(n+o(n))$ bits if the parameters λ and η satisfy $0 < \lambda < 1 - \eta$.*

Supporting `sum`, `search` *and* `update`. We discuss these three operations first because they do not change the size of C.

Lemma 3. *The data structures in this section can support* `sum`, `search` *and* `update` *in* $O(\frac{\lg n}{\lg\lg n})$ *time with an additional universal table of* $o(n)$ *bits.*

Proof. To support $\texttt{sum}(C, j, i)$, we perform a top-down traversal in the B-tree. In our algorithm, we use a variable r that is initially 0, and its value will increase as we go down the tree. We have another variable s whose initial value is i. Initially, let v be the root of the tree. As $P(v)$ stores the number of positions stored in the subtrees rooted at each child of v, the subtree rooted at the c^{th} child of v, where $c = \texttt{search}(P(v), i)$, contains position i. We also compute the sum of the integers from the sequence Q_j that are stored in the subtrees rooted at the left siblings of the c^{th} child of v, which is $y = \texttt{sum}(R_j(v), c-1)$, and we increase the value of r by y. We then set v to be its c^{th} child, decrease the value of s by $\texttt{sum}(P(v), c-1)$, and the process continues until we reach a leaf. At this time, r records the sum of the integers from the sequence Q_j that are before the first position stored in the superblock of the leaf we reach. As the height of this B-tree is $O(\frac{\lg n}{\lg\lg n})$, and the computation at each internal node takes constant time by Lemma 1, it takes $O(\frac{\lg n}{\lg\lg n})$ time to locate this superblock.

It now suffices to compute the sum of the first s integers from sequence Q_j that are stored in the superblock. This can be done by first going to the block storing the first integer in the superblock that is from Q_j, which takes $O(\frac{\sqrt{\lg n}}{\lg\lg n})$ time (recall that each block is of fixed size and there are $O(\frac{\sqrt{\lg n}}{\lg\lg n})$ of them in a superblock), and then read the sequence in chunks of $\lceil \frac{1}{2}\lg n \rceil$ bits. For each $\lceil \frac{1}{2}\lg n \rceil$ bits we read, we use a universal table A_1 to find out the sum of the $z = \lfloor \lceil \frac{1}{2}\lg n \rceil / k \rfloor$ integers stored in it in $O(1)$ time (the last $a = \lceil \frac{1}{2}\lg n \rceil \bmod k$ bits in this block are concatenated with the next $\lceil \frac{1}{2}\lg n \rceil - a$ bits read for table lookup). This table simply stores the result for each possible bit strings of length $\lceil \frac{1}{2}\lg n \rceil$. The last chunk we read may contain integers after $Q_j[i]$. To address the problem, we augment A_1 so that it is a two dimensional table $A_1[1..2^{\lceil \frac{1}{2}\lg n \rceil}][1..z]$, in which $A[b][g]$ stores for the b^{th} lexicographically smallest bit vector of length $\lceil \frac{1}{2}\lg n \rceil$, the sum of the first g integers of size k stored in it. This way the computation in the superblock can be done in $O(\frac{\lg n}{\lg\lg n})$ time, and thus $\texttt{sum}(C, j, i)$ can be supported in $O(\frac{\lg n}{\lg\lg n})$ time. The additional data structure we require is table A_1, which occupies $O(2^{\lceil \frac{1}{2}\lg n \rceil} \times \lfloor \lceil \frac{1}{2}\lg n \rceil / k \rfloor \times \lg n) = O((\sqrt{n}\lg^2 n)/k)$ bits. The operations `search` and `update` can be supported in a similar manner. □

Supporting `insert` *and* `delete`. We give a proof sketch of the following lemma on supporting `insert` and `delete`:

Lemma 4. *When* $w = \Theta(\lg n)$*, the data structures in this section can support* `insert` *and* `delete` *in* $O(\frac{\lg n}{\lg\lg n})$ *amortized time.*

Proof (sketch). To support $\texttt{insert}(C, i)$, we locate the leaf containing position i as we do for `sum`, updating $P(v)$'s along the way. We insert a 0 before the

i^{th} position of all the sequences by creating a new superblock, copying the data from the old superblock contained in this leaf to this new superblock in chunks of size $\lceil \lg n \rceil$, and adding 0's at appropriate positions when we copy. This takes $O(\frac{\lg n}{\lg \lg n} + d) = O(\frac{\lg n}{\lg \lg n})$ time. If the size of the new superblock exceeds $2L$, we split it into two superblocks of roughly the same size. The parent of the old leaf becomes the parent, v, of both new leaves, and we reconstruct the data structures for $P(v)$ and $R_j(v)$'s in $O(df) = o(\frac{\lg n}{\lg \lg n})$ time. This may make a series of internal nodes to overflow, and in the amortized sense, each split of the leaf will only cause a constant number of internal nodes to overflow. Thus we can support `insert` in $O(\frac{\lg n}{\lg \lg n})$ amortized time. The support for `delete` is similar.

Each `insert` or `delete` changes n by 1. This might change the value $\lceil \lg n \rceil$, which will in turn affect L, the size of blocks, and the content of A_1. As $w = \Theta(\lg n)$, L and the block size will only change by a constant factor. Thus if we do not change these parameters, all our previous space and time analysis still applies. The $o(n)$ time required to reconstruct A_1 each time $\lceil \lg n \rceil$ changes can be charged to at least $\Theta(n)$ `insert` or `delete` operations. □

As we use a B-tree, a new problem is to deamortize the support for `insert` and `delete`. We also need to consider the case in which the word size is $w = \Omega(\lg n)$. The following lemma presents our solution to the CSPSI problem:

Lemma 5. *Consider a collection, C, of d sequences of n nonnegative integers each ($d = O(\lg^{\eta} n)$ for any constant $0 < \eta < 1$), in which each integer requires k bits. Under the word RAM model with word size $\Omega(\lg n)$, C can be represented using $O(kdn + w)$ bits to support* `sum`, `search`, `update`, `insert` *and* `delete` *in $O(\frac{\lg n}{\lg \lg n})$ time with a buffer of $O(n \lg n)$ bits (for the operation* `update`(C, j, i, δ)*, we assume $|\delta| \leq \lg n$).*

Proof (sketch). To deamortize the algorithm for `insert` and `delete`, we first observe that the table A_1 can be built incrementally each time we perform `insert` and `delete`. Thus the challenging part is to re-balance the B-tree (i.e. to merge and split its leaves and internal nodes) after insertion and deletion. For this we use the *global rebuilding* approach of Overmars and van Leeuwen [21]. By their approach, if there exist two constant numbers $c_1 > 0$ and $0 < c_2 < 1$ such that after performing $c_1 n$ insertions and/or $c_2 n$ deletions without re-balancing the B-tree, we can still perform query operations in $O(\frac{\lg n}{\lg \lg n})$ time, and if the B-tree can be rebuilt in $O(f(n) \times n)$ time, we can support insertion or deletion in $O(\frac{\lg n}{\lg \lg n} + f(n))$ worse-case time using additional space proportional to the size of our original data structures and a buffer of size $O(n \lg n)$ bits. We first note that if we do not re-balance the B-tree after performing `delete` $c_2 n$ times for any $0 < c_2 < 1$, the time required to answer a query will not change asymptotically. This is however different for `insert`, and we use the approach of Fleischer [22] as in [10]. Essentially, in his approach, at most one internal node and one leaf is split after each insertion, which guarantees that the degree of any internal node does not exceed $4f$. This way after $\Theta(n)$ insertions, query operations can still be performed in $O(\frac{\lg n}{\lg \lg n})$ time. Finally, it takes $O(nd)$ time to construct the B-tree, so we can support `insert` and `delete` in $O(d + \frac{\lg n}{\lg \lg n}) = O(\frac{\lg n}{\lg \lg n})$ time.

To reduce the space overhead when $w = \omega(\lg n)$, we allocate a memory block whose size is sufficient for the new structure until another structure has to be built, and this increases the space cost by a constant factor. Then we can still use pointers of size $O(\lg n)$ bits (not $O(w)$ bits), and $O(w)$ bits are required to record the address of memory blocks allocated. □

Section 2 states that González and Navarro [6] designed a data structure of $kdn(1 + O(\frac{1}{\sqrt{\lg n}} + \frac{d}{\lg n}))$ bits. This is more compact, but it only works for the special case in which $w = \Theta(\lg n)$. González and Navarro [6] actually stated that their result would work when $w = \Omega(\lg n)$. This requires greater care than given in their paper. Their strategy is to adapt the approach of Mäkinen and Navarro [13] developed originally for a dynamic bit vector structure. To use it for the CSPSI problem, they split each sequence into three subsequences. The split points are the same for all the sequences in C. The set of left, middle, and right subsequences constitute three collections, and they build CSPSI structures for each of them. For each insertion and deletion, a constant number of elements is moved from one collection to another, which will eventually achieve the desired result with other techniques. Moving one element from one collection to another means that the first or the last integers of all the subsequences in one collection must be moved to the subsequences in another collection. However, their CSPSI structure only supports insertions and deletions of 0's at the same position in all subsequences in $O(\lg n)$ time, so moving one element cannot be supported fast enough. Thus their structure only works when $w = \Theta(\lg n)$. Their result can be generalized to the case in which $w = \Omega(\lg n)$ using the approach in Lemma 5, but the space will be increased to $O(kdn + w)$ bits and a buffer will be required.

4 Strings over Small Alphabets

In this section, we consider representing a dynamic string $S[1..n]$ over an alphabet of size $\sigma = O(\sqrt{\lg n})$ to support `access`, `rank`, `select`, `insert` and `delete`.

Data Structures. Our main data structure is a B-tree constructed over S. We again let $L = \lceil \frac{\lceil \lg n \rceil^2}{\lg \lceil \lg n \rceil} \rceil$. Each leaf of this B-tree contains a *superblock* that has at most $2L$ bits. We say that a superblock is *skinny* if it has fewer than L bits. The string S is initially partitioned into substrings, and each substring is stored in a superblock. We number the superblocks consecutively from left to right starting from 1. Superblock i stores the i^{th} substring from left to right. To bound the number of leaves and superblocks, we require that there do not exist two consecutive skinny superblocks. Thus there are $O(\frac{n \lg \sigma}{L})$ superblocks.

Let $b = \sqrt{\lg n}$, and we require that the degree of each internal node of the B-tree is at least b and at most $2b$. For each internal node v, we store the following data structures encoded by Lemma 1 (let h be the number of children of v):

- A sequence $U(v)[1..h]$, in which $U(v)[i]$ is the number of superblocks contained in the leaves of the subtree rooted at the i^{th} child of v;

– A sequence $I(v)[1..h]$, in which $I(v)[i]$ stores the number of characters stored in the leaves of the subtree rooted at the i^{th} child of v.

As in Section 3, each superblock is further stored in a list of blocks of $\lceil \lceil \lg n \rceil^{3/2} \rceil$ bits each, and only the last block in each list can have free space.

Finally for each character α, we construct an integer sequence $E_\alpha[1..t]$ in which $E_\alpha[i]$ stores the number of occurrences of character α in superblock i (t denotes the number of superblocks). We create σ integer sequences in this way, and we construct a CSPSI structure, E, for them using Lemma 5. Note that the buffered required in Lemma 5 to support operations on E takes $o(n)$ bits here as the length of the sequences in E is $O(n/L)$.

The above data structures occupy $n \lg \sigma + O(\frac{n \lg \sigma \lg \lg n}{\sqrt{\lg n}})$ bits.

Supporting `access`*,* `rank` *and* `select`*.* We first support query operations.

Lemma 6. *The data structures in this section can support* `access`*,* `rank` *and* `select` *in* $O(\frac{\lg n}{\lg \lg n})$ *time with a universal table of* $O(\sqrt{n}\texttt{polylog}(n))$ *bits.*

Proof. To support $\texttt{access}(S, i)$, we perform a top-down traversal in the B-tree to find the leaf containing $S[i]$. During this traversal, at each internal node v, we perform `search` on $I(v)$ to decide which child to traverse, and perform `sum` on $I(v)$ to update the value i. When we find the leaf, we follow the pointers to find the block containing the position we are looking for, and then retrieve the corresponding character in constant time. Thus `access` takes $O(\frac{\lg n}{\lg \lg n})$ time.

To compute $\texttt{rank}_\alpha(S, i)$, we first locate the leaf containing position i using the same process for `access`. Let j be the number of the superblock contained in this leaf, which can be computed using $U(v)$ during the top-down traversal. Then $\texttt{sum}(E, \alpha, j-1)$ is the number of occurrences of α in superblocks $1, 2, \cdots j-1$, which can be computed in $O(\frac{\lg n}{\lg \lg n})$ time by Lemma 5. To compute the number of occurrences of α up to position i inside superblock j, we read the content of this superblock in chunks of size $\lceil \frac{1}{2} \lg n \rceil$ bits. As with the support for `sum` in the proof of Lemma 3, this can be done in $O(\frac{\lg n}{\lg \lg n})$ time using a precomputed table A_2 of $O(\sqrt{n}\texttt{polylog}(n))$ bits. Our support for $\texttt{select}_\alpha(S, i)$ is similar. □

Supporting `insert` *and* `delete`*.* Careful tuning of the techniques for supporting insertions and deletions for the CSPSI problem yields the following lemma:

Lemma 7. *When* $w = \Theta(\lg n)$*, the data structures in this section can support* `insert` *and* `delete` *in* $O(\frac{\lg n}{\lg \lg n})$ *amortized time.*

To deamortize the support for `insert` and `delete`, we cannot directly use the global rebuilding approach of Overmars and van Leeuwen [21] here, since we do not want to increase the space cost of our data structures by a constant factor, and the use of a buffer is also unacceptable. Instead, we design an approach that deamortizes the support for `insert` and `delete` completely, and differently from the original global rebuilding approach of Overmars and van Leeuwen [21], our approach neither increases the space cost by a constant factor, nor requires

any buffer. We thus call our approach *succinct global rebuilding*. This approach still requires us to modify the algorithms for `insert` and `delete` so that after $c_1 n$ insertions ($c_1 > 0$) and $c_2 n$ deletions ($0 < c_2 < 1$), a query operation can still be supported in $O(\frac{\lg n}{\lg \lg n})$ time. We also start the rebuilding process when the number of `insert` and `delete` operations performed exceeds half the initial length of the string stored in the data structure. The main difference between our approach and the original approach in [21] is that during the process of rebuilding, we never store two copies of the same data, i.e. the string S. Instead, our new structure stores a prefix, S_p, of S, and the old data structure stores a suffix, S_s, of S. During the rebuilding process, each time we perform an insertion or deletion, we perform such an operation on either S_p or S_s. After that, we remove the first several characters from S_s, and append them to S_p. By choosing parameters and tuning our algorithm carefully, we can finish rebuilding after at most $n_0/3$ update operations, where n_0 is the length of S when we start rebuilding. During this process, we use both old and new data structures to answer queries in $O(\frac{\lg n}{\lg \lg n})$ time.

To reduce the space overhead when $w = \omega(\lg n)$, we adapt the approach of Mäkinen and Navarro [13]. We finally use the approach of González and Navarro [6] to compress our representation. Since their approach is only applied to the superblocks (i.e. it does not matter what kind of tree structures are used, since additional tree arrangement operations are not required when the number of bits stored in a leaf is increased due to a single update in their solution), we can use it here directly. Thus we immediately have:

Lemma 8. *Under the word RAM model with word size $w = \Omega(\lg n)$, a string S of length n over an alphabet of size $\sigma = O(\sqrt{\lg n})$ can be represented using $nH_0 + O(\frac{n \lg \sigma \lg \lg n}{\sqrt{\lg n}}) + O(w)$ bits to support* `access`, `rank`, `select`, `insert` *and* `delete` *in $O(\frac{\lg n}{\lg \lg n})$ time.*

5 Strings over General Alphabets

We follow the approach of Ferragina *et al.* [23] that uses a generalized wavelet tree to extend results on strings over small alphabets to general alphabets. Special care must be taken to avoid increasing the $O(w)$-bit term in Lemma 8 asymptotically. We now present our main result:

Theorem 1. *Under the word RAM model with word size $w = \Omega(\lg n)$, a string S of length n over an alphabet of size σ can be represented using $nH_0 + O(\frac{n \lg \sigma \lg \lg n}{\sqrt{\lg n}}) + O(w)$ bits to support* `access`, `rank`, `select`, `insert` *and* `delete` *operations in $O(\frac{\lg n}{\lg \lg n}(\frac{\lg \sigma}{\lg \lg n} + 1))$ time. When $\sigma = O(\texttt{polylog}(n))$, all the operations can be supported in $O(\frac{\lg n}{\lg \lg n})$ time.*

The following corollary is immediate:

Corollary 1. *Under the word RAM model with word size $w = \Omega(\lg n)$, a bit vector of length n can be represented using $nH_0 + o(n) + O(w)$ bits to support* `access`, `rank`, `select`, `insert` *and* `delete` *in $O(\frac{\lg n}{\lg \lg n})$ time.*

6 Applications

Dynamic Text Collection. Mäkinen and Navarro [13] showed how to use a dynamic string structure to index a text collection N to support string search. For this problem, n denotes the length of the text collection N when represented as a single string that is the concatenations of all the text strings in the collection (a separator is inserted between texts). González and Navarro [6] improved this result in [13] by improving the string representation. If we use our string structure, we directly have the following lemma, which improves the running time of the operations over the data structure in [6] by a factor of $\lg\lg n$:

Theorem 2. *Under the word RAM model with word size $w = \Omega(\lg n)$, a text collection N of size n consisting of m text strings over an alphabet of size σ can be represented in $nH_h + o(n)\cdot\lg\sigma + O(m\lg n + w)$ bits[4] for any $h \le (\alpha\log_\alpha n) - 1$ and any constant $0 < \alpha < 1$ to support:*

- *the counting of the number of occurrences of a given query substring P in N in $O(\frac{|P|\lg n}{\lg\lg n}(\frac{\lg\sigma}{\lg\lg n} + 1))$ time;*
- *After counting, the locating of each occurrence in $O(\lg^2 n(\frac{1}{\lg\lg n} + \frac{1}{\lg\sigma}))$ time;*
- *Inserting and deleting a text T in $O(\frac{|T|\lg n}{\lg\lg n}(\frac{\lg\sigma}{\lg\lg n} + 1))$ time;*
- *Displaying any substring of length l of any text string in N in $O(\lg^2 n(\frac{1}{\lg\lg n} + \frac{1}{\lg\sigma}) + \frac{l\lg n}{\lg\lg n}(\frac{\lg\sigma}{\lg\lg n} + 1))$ time.*

Compressed Construction of Text Indexes. Researchers have designed space-efficient text indexes whose space is essentially a compressed version of the given text, but the construction of these text indexes may still require a lot of space. Mäkinen and Navarro [13] used their dynamic string structure to construct a variant of FM-index [24] using as much space as what is required to encode the index. Their result was improved by González and Navarro [6], and the construction time can be further improved by a factor of $\lg\lg n$ using our structure:

Theorem 3. *A variant of a FM-index of a text string $T[1..n]$ over an alphabet of size σ can be constructed using $nH_h + o(n)\cdot\lg\sigma$ bits of working space in $O(\frac{n\lg n}{\lg\lg n}(\frac{\lg\sigma}{\lg\lg n} + 1))$ time for any $h \le (\alpha\log_\alpha n) - 1$ and any constant $0 < \alpha < 1$.*

7 Concluding Remarks

In this paper, we have designed a succinct representation of dynamic strings that provide more efficient operations than previous results, and we have successfully applied it to improve previous data structures on text indexing. As a string structure supporting rank and select is used in the design of succinct representations of many data types, we expect our data structure to play an important role in the future research on succinct dynamic data structures. We have also created some new techniques to achieve our results. We particularly think that the approach of succinct global rebuilding is interesting, and expect it to be useful for deamortizing algorithms on other succinct data structures.

[4] H_h is the h^{th}-order entropy of the text collection when represented as a single string.

References

1. Jacobson, G.: Space-efficient static trees and graphs. In: FOCS. (1989) 549–554
2. Clark, D.R., Munro, J.I.: Efficient suffix trees on secondary storage. In: SODA. (1996) 383–391
3. Raman, R., Raman, V., Satti, S.R.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Transactions on Algorithms **3**(4) (2007) 43
4. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: SODA. (2003) 841–850
5. Barbay, J., He, M., Munro, J.I., Rao, S.S.: Succinct indexes for strings, binary relations and multi-labeled trees. In: SODA. (2007) 680–689
6. González, R., Navarro, G.: Rank/select on dynamic compressed sequences and applications. Theoretical Computer Science **410**(43) (2009) 4414–4422
7. Barbay, J., Golynski, A., Munro, J.I., Rao, S.S.: Adaptive searching in succinctly encoded binary relations and tree-structured documents. Theoretical Computer Science **387**(3) (2007) 284–297
8. Munro, J.I., Raman, V.: Succinct representation of balanced parentheses and static trees. SIAM Journal on Computing **31**(3) (2001) 762–776
9. Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Compressing and indexing labeled trees, with applications. Journal of the ACM **57**(1) (2009)
10. Sadakane, K., Navarro, G.: Fully-functional succinct trees. In: SODA. (2010) 134–149
11. Barbay, J., Castelli Aleardi, L., He, M., Munro, J.I.: Succinct representation of labeled graphs. In: ISAAC, Springer-Verlag LNCS 4835 (2007) 316–328
12. Chan, H.L., Hon, W.K., Lam, T.W.: Compressed index for a dynamic collection of texts. In: CPM. (2004) 445–456
13. Mäkinen, V., Navarro, G.: Dynamic entropy-compressed sequences and full-text indexes. ACM Transactions on Algorithms **4**(3) (2008)
14. Blandford, D.K., Blelloch, G.E.: Compact representations of ordered sets. In: SODA. (2004) 11–19
15. Chan, H.L., Hon, W.K., Lam, T.W., Sadakane, K.: Compressed indexes for dynamic text collections. ACM Transactions on Algorithms **3**(2) (2007)
16. Lee, S., Park, K.: Dynamic rank/select structures with applications to run-length encoded texts. Theoretical Computer Science **410**(43) (2009) 4402–4413
17. Gupta, A., Hon, W.K., Shah, R., Vitter, J.S.: A framework for dynamizing succinct data structures. In: ICALP. (2007) 521–532
18. Fredman, M.L., Saks, M.E.: The cell probe complexity of dynamic data structures. In: STOC. (1989) 345–354
19. Raman, R., Raman, V., Rao, S.S.: Succinct dynamic data structures. In: WADS. (2001) 426–437
20. He, M., Munro, J.I.: Succinct representations of dynamic strings. CoRR **abs/1005.4652** (2010)
21. Overmars, M.H., van Leeuwen, J.: Worst-case optimal insertion and deletion methods for decomposable searching problems. Inf. Process. Lett. **12**(4) (1981) 168–173
22. Fleischer, R.: A simple balanced search tree with o(1) worst-case update time. Int. J. Found. Comput. Sci. **7**(2) (1996) 137–150
23. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Transactions on Algorithms **3**(2) (2007)
24. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: An alphabet-friendly FM-index. In: SPIRE, Springer-Verlag LNCS 3246 (2004) 150–160