7-2007

A Framework for Dynamizing Succinct Data Structures

Ankur Gupta
Butler University, agupta@butler.edu

Wing K. Hon

Rahul Shah

Jeffery S. Vitter

Recommended Citation

Ankur Gupta, Wing-Kai Hon, Rahul Shah, and Jeffery Scott Vitter. A Framework for Dynamizing Succinct Data Structures. In Proceedings of International Colloquium on Automata, Languages, and Programming (ICALP), Wroclaw, Poland, July 2007.

A Framework for Dynamizing Succinct Data Structures*

Ankur Gupta[1], Wing-Kai Hon[2], Rahul Shah[1], and Jeffrey Scott Vitter[1]

[1] Department of Computer Sciences, Purdue University, West Lafayette, IN 47907–2066, USA ({agupta, rahul, jsv}@cs.purdue.edu).
[2] Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan (wkhon@cs.nthu.edu.tw).

Abstract. We present a framework to dynamize succinct data structures, to encourage their use over non-succinct versions in a wide variety of important application areas. Our framework can dynamize most state-of-the-art succinct data structures for dictionaries, ordinal trees, labeled trees, and text collections. Of particular note is its direct application to XML indexing structures that answer *subpath* queries [2]. Our framework focuses on achieving information-theoretically optimal space along with near-optimal update/query bounds.

As the main part of our work, we consider the following problem central to text indexing: Given a text T over an alphabet Σ, construct a compressed data structure answering the queries $char(i)$, $rank_s(i)$, and $select_s(i)$ for a symbol $s \in \Sigma$. Many data structures consider these queries for static text T [5, 3, 16, 4]. We build on these results and give the best known query bounds for the dynamic version of this problem, supporting arbitrary insertions and deletions of symbols in T.

Specifically, with an amortized update time of $O(n^\epsilon)$, any static succinct data structure D for T, taking $t(n)$ time for queries, can be converted by our framework into a dynamic succinct data structure that supports $rank_s(i)$, $select_s(i)$, and $char(i)$ queries in $O(t(n) + \log\log n)$ time, for any constant $\epsilon > 0$. When $|\Sigma| = \text{polylog}(n)$, we achieve $O(1)$ query times. Our update/query bounds are near-optimal with respect to the lower bounds from [13].

1 Introduction

The new trend in indexing data structures is to compress and index data in one shot. The ultimate goal of these compressed indexes is to retain near-optimal query times (as if not compressed), yet still take near-optimal space (as if not an index). A few pioneer results are [6, 5, 3, 15, 4, 2]; there are many others. For compressed text indexing, see Navarro and Mäkinen's excellent survey [11].

Progress in compressed indexing has also expanded to more combinatorial structures, such as trees and subsets. For these *succinct* data structures, the

* Support was provided in part by the National Science Foundation through research grants CCF–0621457 and IIS–0415097.

emphasis is to store them in terms of the information-theoretic (combinatorial) minimum required space with fast query times [15, 9, 7]. Compressed text indexing makes heavy use of succinct data structures for set data, or *dictionaries*.

The vast majority of succinct data structuring work is concerned largely with static data. Although the space savings is large, the main deterrent to a more ubiquitous use of succinct data structures is their notable lack of support for dynamic operations. Many settings require indexing and query functionality on dynamic data: XML documents, web pages, CVS projects, electronic document archives, etc. For this type of data, it can be prohibitively expensive to rebuild a static index from scratch each time an update occurs. The goal is then to answer queries efficiently, perform updates in a reasonable amount of time, and *still* maintain a compressed version of the dynamically-changing data.

In that vein, there have been some results on dynamic succinct bitvectors (dictionaries) [14, 8, 12]. However, these data structures either perform queries in far from optimal time (in query-intensive environments), or allow only a limited range of dynamic operations ("flip" operations only). Here, we consider the more general update operations consisting of arbitrary insertion and deletion of bits, which is a central challenge in dynamizing succinct data structures for a variety of applications. We define the *dynamic text dictionary* problem: Given a dynamic text T of n symbols drawn from an alphabet Σ, construct a data structure (index) that allows the following operations for any symbol $s \in \Sigma$:

- $rank_s(i)$ tells the number of s symbols up to the ith position in T;
- $select_s(i)$ gives the position in T of the ith s;
- $char(i)$ returns the symbol in the ith position of T;
- $insert_s(i)$ inserts s before the position i in T;
- $delete(i)$ deletes the ith symbol from T.

When $|\Sigma| = 2$, the above problem is called the *dynamic bit dictionary* problem. For the static case, [15] solves the bit dictionary problem using $nH_0 + o(n)$ bits of space and answers rank and select queries in $O(1)$ time, where H_0 is the 0th order empirical entropy of the text T. The best known time bounds for the dynamic problem are given by [12], achieving $O(\log n)$ for all operations.[3]

The text dictionary problem is a key tool in text indexing data structures. For the static case, Grossi et al. [5] present a wavelet tree structure that answers queries in $O(\log |\Sigma|)$ time and takes $nH_0 + o(n \log |\Sigma|)$ bits of space. Golynski et al. [4] improve the query bounds to $O(\log \log |\Sigma|)$ time, although they take more space, namely, $n \log |\Sigma| + o(n \log |\Sigma|)$ bits of space. Nevertheless, their data structure presents the best query bounds for this problem.

Developing a dynamic text dictionary based on the wavelet structure can be done readily using dynamic bit dictionaries (as is done in [12]) since updates to a particular symbol s only affect the data structures for $O(\log |\Sigma|)$ groups of symbols according to the hierarchical decomposition of the alphabet Σ. The solution to this problem is given by Mäkinen and Navarro [12], with an update/query bound of $O(\log n \log |\Sigma|)$. These bounds are far from optimal, especially in query-intensive settings. On the other hand, the best known query bounds for static

[3] There is another data structure proposed in [8], requiring non-succinct space.

text dictionaries are given by [4], which treats each symbol in Σ individually; an update to symbol s could potentially affect Σ different data structures, and thus may be hard to dynamize.

We list the following contributions of our paper:

- We develop a general framework to dynamize many succinct data structures like ordinal trees, labeled trees, dictionaries, and text collections. Our framework can transform any static succinct data structure D for a text T into a dynamic succinct data structure. Precisely, if D supports $rank_s$, $select_s$, and *char* queries in $O(t(n))$ time and takes $s(n)$ bits of space, the dynamic data structure supports queries in $O(t(n) + \log\log n)$ time and updates in amortized $O(n^\epsilon)$ time and takes just $s(n) + o(n)$ bits of space.
- Our results represent near-optimal tradeoffs for update/query times for the dynamic text (and bit) dictionary problem. (For lower bound, see [13].)
- We provide the first succinct data structure for the *dynamic bit dictionary* problem. Our data structure takes $nH_0 + o(n)$ bits of space and requires $O(\log\log n)$ time to support $rank_s$, $select_s$, and *char* queries while supporting updates to the text T in amortized $O(n^\epsilon)$ time.
- We provide the first near-optimal result for the *dynamic text dictionary* problem on a dynamic text T. Our data structure requires $n\log|\Sigma| + o(n\log|\Sigma|)$ bits of space and supports queries in $O(\log\log n)$ time and updates in $O(n^\epsilon)$ time. When $|\Sigma| = \text{polylog}(n)$, we can improve our query time to $O(1)$.
- Our framework can dynamize succinct data structures for labeled trees, text collections, and XML documents.

2 Preliminaries

We summarize several important static structures that we will use in achieving the dynamic results. The proofs of their construction are omitted due to space constraints. In the rest of this paper, we refer to a static bit or text dictionary D, that requires $s(n)$ bits and answers queries in $t(n)$ time.

Lemma 1 ([15]). *For a bitvector (i.e., $|\Sigma| = 2$) of length n, there exists a static data structure D called* RRR *solving the* bit dictionary *problem supporting rank, select, and char queries in $t(n) = O(1)$ time using $s(n) = nH_0 + O(n\log\log n/\log n)$ bits of space, while taking only $O(n)$ time to construct.* □

Lemma 2 ([5]). *For a text T of length n drawn from alphabet Σ, there exists a static data structure D called the* wavelet tree *solving the* text dictionary *problem supporting $rank_s$, $select_s$, and char queries in $t(n) = O(\log|\Sigma|)$ time using $s(n) = nH_0 + o(n\log|\Sigma|)$ bits of space, while taking $O(nH_0)$ time to construct. When $|\Sigma| = \text{polylog}(n)$, we can support queries in $t(n) = O(1)$ time.* □

Lemma 3 ([4]). *For a text T of length n drawn from alphabet Σ, there exists a static data structure D called* GMR *that solves the* text dictionary *problem supporting $select_s$ queries in $t_1(n) = O(1)$ time and rank and char queries in $t_2(n) = O(\log\log|\Sigma|)$ time using $s(n) = n\log|\Sigma| + o(n\log|\Sigma|)$ bits of space, while taking $O(n\log n)$ time to construct.* □

We also use the following static data structure called prefix-sum (PS) as a building block for achieving our dynamic result. Suppose we are given a non-negative integer array $A[1..t]$ such that $\sum_i A[i] \leq n$. We define the partial sums $P[i] = \sum_{j=1}^{i} A[i]$. Note that P is a sorted array, such that $0 \leq P[i] \leq P[j] \leq n$ for all $i < j$. A prefix-sum (PS) structure on A is a data structure that supports the following operations:

- $sum(j)$ returns the partial sum $P[j]$;
- $findsum(i)$ returns the index j such that $sum(j) \leq i < sum(j+1)$.

Lemma 4. *Let $A[1 \ldots t]$ be a non-negative integer array such that $\sum_i A[i] \leq n$. There exists a data structure PS on A that supports sum and findsum in $O(\log \log n)$ time using $O(t \log n)$ bits of space and can be constructed in $O(t)$ time. In the particular case where $x \leq A[i] \leq cx$ for all i, where x is a positive integer and $c \geq 1$ is a positive constant integer, sum and findsum can be answered in $O(1)$ time.* □

We also make use of a data structure called the Weight Balanced B-tree (WBB tree), which was used in [14, 8]. We use this structure with Lemma 4 to achieve $O(1)$ time. A WBB tree is a B-tree defined with a *weight-balance condition*. A weight-balance condition means that for any node v at level i, the number of leaves in v's subtree is between $0.5b^i + 1$ and $2b^i - 1$, where b is the fanout factor. Insertions and deletions on the WBB tree can be performed in amortized $O(\log_b n)$ time while maintaining the weight-balance condition.

We use the WBB tree since it ensures that $x \leq A[i] \leq cx$ where c is a positive constant integer, thus allowing constant-time search at each node. However, a simple B-tree would require $O(\log \log n)$ time in this situation. Also, WBB trees are a crucial component of the onlyX structure, described in Section 3.3. WBB trees are also used in Section 3.1 (although B-trees could be used here).

3 Data Structures

Our solution is built with three main data structures:

- *BitIndel*: bitvector supporting insertion and deletion, described in Section 3.1;
- *StaticRankSelect*: static text dictionary structure supporting $rank_s$, $select_s$, and *char* on a text T;
- *onlyX*: non-succinct dynamic text dictionary, described in Section 3.3

We use StaticRankSelect to maintain the original text T; we can use any existing structure such as GGV or GMR mentioned in Section 2. For ease of exposition, unless otherwise stated, we shall use GMR [4] in this section. We keep track of newly inserted symbols N in onlyX such that after every $O(n^{1-\epsilon} \log n)$ update operations performed, updates are merged with the StaticRankSelect structure. Thus, onlyX never contains more than $O(n^{1-\epsilon} \log n)$ symbols. We maintain onlyX using $O(n^{1-\epsilon} \log^2 n) = o(n)$ bits of space. Finally, since merging N with T requires $O(n \log n)$ time, we arrive at an amortized $O(n^{\epsilon})$ time for updating these data structures. BitIndel is used to translate positions p_t from the old text T to the new positions $p_{\hat{t}}$ from the current text $\hat{T}$. (We maintain $\hat{T}$ implicitly through the use of BitIndel structures, StaticRankSelect, and onlyX.)

3.1 Bitvector Dictionary with Indels: BitIndel

In this section, we describe a data structure (BitIndel) for a bitvector B of original length n that can handle insertions and deletions of bits anywhere in B while still supporting *rank* and *select* on the updated bitvector B' of length n'. The space of the data structure is $n'H_0 + o(n')$. When $n' = O(n)$, our structure supports these updates in $O(n^\epsilon)$ time and *rank* and *select* queries in $O(\log \log n)$ time. (In [8], Hon et al. propose a non-succinct BitIndel structure taking $n'+o(n')$ bits of space.)

Formally, we define the following update operations that we support on the current bitvector B' of length n': $insert_b(i)$ inserts the bit b in the ith position, $delete(i)$ deletes the bit located in the ith position, and $flip(i)$ flips the bit in the ith position.

We defer the details until the full paper. The idea is to use a B-tree over $\Theta(n^\epsilon)$-sized chunks of the bitvector, which are stored using an RRR structure. This B-tree is constant-height and needs prefix-sum data structures in its internal nodes for fast access.

Lemma 5. *Given a bitvector B' with length n' and original length n, we can create a data structure that takes $n'H_0 + o(n')$ bits and supports rank and select in $O((\log_n n') \log \log n)$ time, and indel in $O(n^\epsilon \log_n n')$ time. When $n' = O(n)$, our time bounds become $O(\log \log n)$ and $O(n^\epsilon)$ respectively.* □

The prefix sum data structure used inside the B-tree is the main bottleneck to query times, allowing us only $O(\log \log n)$ time access. However, if we store three WBB-trees, then separately in each of them the special condition from Lemma 4 can be met allowing us $O(1)$ queries on prefix sum structures. This allows us to obtain the following lemma.

Lemma 6. *Given a bitvector B' with length n' and original length n, we can create a data structure that takes $3n'H_0 + o(n')$ bits and supports rank and select in $O(\log_n n')$ time, and indel in $O(n^\epsilon \log_n n')$ amortized time. When $n' = O(n)$, our time bounds become $O(1)$ and $O(n^\epsilon)$ respectively.* □

If we change our BitIndel structure such that the bottom-level RRR [15] data structures are built on $[\log^2 n, 2\log^2 n]$ bits each and set the B-tree fanout factor $b = 2$, we can obtain $O(\log n)$ update time with $O(\log n)$ query time. In this sense, our BitIndel data structure is a generalization of [12].

3.2 Insert-X-Delete-any: inX

Let $\mathtt{x}$ be a symbol other than those in alphabet Σ. In this section, we describe a data structure on a text T of length n supporting $rank_s$ and $select_s$ that can handle $delete(i)$ and $insert_{\mathtt{x}}(i)$. That is, only x can be inserted to T, while any characters can be deleted from T. Notice that insertions and deletions will affect the answers returned for symbols in the alphabet Σ. For example, T may be $\mathtt{abcaab}$, where $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. Here, $rank_{\mathtt{a}}(4) = 2$ and $select_{\mathtt{a}}(3) = 5$. Let $\hat{T}$ be the current text after some number of insertions and deletions of symbol $\mathtt{x}$. Initially, $\hat{T} = T$. After some insertions, the current $\hat{T}$ may be $\mathtt{axxxbcaxabx}$.

Notice that $rank_{\mathtt{a}}(4) = 1$ and $select_{\mathtt{a}}(3) = 9$. We represent $\hat{T}$ by the text T', such that when the symbols of the original text T are deleted, each deleted symbol is replaced by a special symbol `d` (whereas if x is deleted, it is just deleted from T'). Continuing the example, after some deletions of symbols from T, T' may be `axxxddaxabx`. Notice that $rank_{\mathtt{a}}(4) = 1$ and $select_{\mathtt{a}}(3) = 7$.

We define an *insert vector* I such that $I[i] = \mathbf{1}$ if and only if $T'[i] = \mathtt{x}$. Similarly, we define a *delete vector* D such that $D[i] = \mathbf{1}$ if and only if $T'[i] = \mathtt{d}$. We also define a delete vector D_s for each symbol s such that $D_s[i] = \mathbf{1}$ if and only if the ith s in the original text T was deleted. The text T' is merely a conceptual text: we refer to it for ease of exposition but we actually maintain $\hat{T}$ instead.

To store $\hat{T}$, we store T using the StaticRankSelect data structure and store all of the I, D, D_s bitvectors using the constant time BitIndel structure. Now, we describe $\hat{T}.insert_{\mathtt{x}}(i)$, $\hat{T}.delete(i)$, $\hat{T}.rank_s(i)$, and $\hat{T}.select_s(i)$:

$\hat{T}.insert_{\mathtt{x}}(i)$. First, we convert position i in $\hat{T}$ to its corresponding position i' in T' by computing $i' = D.select_{\mathbf{0}}(i)$. Then we must update our various vectors. We perform $I.insert_{\mathbf{1}}(i')$ on our insert vector, and $D.insert_{\mathbf{0}}(i')$ on our delete vector.

$\hat{T}.delete(i)$. First, we convert position i in $\hat{T}$ to its corresponding position i' in T' by computing $i' = D.select_{\mathbf{0}}(i)$. If i' is newly-inserted (i.e., $I[i'] = \mathbf{1}$), then we perform $I.delete(i')$ and $D.delete(i')$ to reverse the insertion process from above. Otherwise, we first convert position i' in T' to its corresponding position i'' in T by computing $i'' = I.rank_{\mathbf{0}}(i')$. Let $s = T.char(i'')$. Finally, to delete the symbol, we perform $D.flip(i')$ and $D_s.flip(j)$, where $j = T.rank_s(i'')$.

$\hat{T}.rank_s(i)$. First, we convert position i in $\hat{T}$ to its corresponding position i' in T' by computing $i' = D.select_{\mathbf{0}}(i)$. If $s = \mathtt{x}$, return $I.rank_{\mathbf{1}}(i')$. Otherwise, we first convert position i' in T' to its corresponding position i'' in T by computing $i'' = I.rank_{\mathbf{0}}(i')$. Finally, we return $D_s.rank_{\mathbf{0}}(j)$, where $j = T.rank_s(i'')$.

$\hat{T}.select_s(i)$. If $s = \mathtt{x}$, compute $j = I.select_{\mathbf{1}}(i)$ and return $D.rank_{\mathbf{0}}(j)$. Otherwise, we compute $k = D_s.select_{\mathbf{0}}(i)$ to determine i's position among the s symbols from T. We then compute $k' = T.select_s(k)$ to determine its original position in T. Now the position k' from T needs to be mapped to its appropriate location in $\hat{T}$. Similar to the first case, we perform $k'' = I.select_{\mathbf{0}}(k')$ and return $D.rank_{\mathbf{0}}(k'')$, which corresponds to the right position of $\hat{T}$.

$\hat{T}.char(i)$. First, we convert position i in $\hat{T}$ to its corresponding position i' in T' by computing $i' = D.select_{\mathbf{0}}(i)$. If $I[i'] = 1$, return `x`. Otherwise, we convert position i' in T' to its corresponding position i'' in T by computing $i'' = I.rank_{\mathbf{0}}(i')$ and return $T.char(i'')$.

Space and Time. As can be seen, each of the *rank* and *select* operations requires a constant number of accesses to BitIndel and StaticRankSelect structures, thus taking $O(1)$ time to perform. The *indel* operations require $O(n^{\epsilon})$ update time, owing to the BitIndel data structure. The space required for the above data structures comes from the StaticRankSelect structure, which requires $s(n) = O(n \log |\Sigma| + o(n \log |\Sigma|))$ bits of space, and the many BitIndel structures,

whose space can be bounded by $3\log\binom{n'}{n} + 6\log\binom{n'}{n''} + o(n') + O((n'/n^{\epsilon})\log n')$ bits where n'' is number of deletes. If n'' and $n' - n$ are bounded by $n^{1-\epsilon}$, then this expression is $o(n)$ bits.

Theorem 1. *Let T be a dynamic text of original length n and current length n', with characters drawn from an alphabet Σ. Let n'' be the number of deletions. If the number of updates is $O(n^{1-\epsilon})$, We can create a data structure using GMR that takes $n\log|\Sigma| + o(n\log|\Sigma|)$ bits of space and supports $rank_s(i)$ and $select_s(i)$ in $O(1)$ time and $insert_{\mathtt{x}}(i)$ and $delete_s(i)$ in $O(n^{\epsilon})$ time.*

3.3 onlyX-structure

Let T be the dynamic text that we want to maintain, where symbols of T are drawn from alphabet Σ. Let n' be the current length of T, and we assume that $n' = O(n)$. In this section, we describe a data structure for maintaining a dynamic array of symbols that supports $rank_s$ and $select_s$ queries in $O((\log_n n')(t(n) + \log\log n))$ time, for any fixed ϵ with $0 < \epsilon < 1$; here, we assume that the maximum number of symbols in the array is $O(n)$. Our data structure takes $O(n'\log n)$ bits; for each update (i.e., insertion or deletion of a symbol), it can be done in amortized $O(n^{\epsilon})$ time.

We describe how to apply the WBB tree to maintain T while supporting $rank_s$ and $select_s$ efficiently, for any $s \in \Sigma$.[4] In particular, we choose $\epsilon < 1$ and store the symbols of T in a WBB W with fanout factor $b = n^{\delta}$ where $\delta = \epsilon/2$ such that the ith (leftmost) leaf of W stores $T[i]$. Each node at level 1 will correspond to a substring of T with $O(b)$ symbols, and we will maintain a static text dictionary for that substring so that $rank_s$ and $select_s$ are computed for that substring in $t(n) = O(\log\log|\Sigma|)$ time. In each level-ℓ node v_ℓ with $\ell \geq 2$, we store an array *size* such that $size[i]$ stores the number of symbols in the subtree of its ith (leftmost) child. To have fast access to this information at each node, we build a PS structure to store *size*. Also, for each symbol s that appears in the subtree of v_ℓ, v_ℓ is associated with an *s-structure*, which consists of three arrays: pos_s, num_s, and ptr_s. The entry $pos_s[i]$ stores the index of v_ℓ's ith leftmost child whose subtree contains s. The entry $num_s[i]$ stores the number of s in v_ℓ's ith leftmost child whose subtree contains s. The entry $ptr_s[i]$ stores a pointer to the s-structure of v_ℓ's ith leftmost child whose subtree contains s.

The arrays in each s-structure ($size_s$, pos_s, and num_s) are stored using a PS data structure so that we can support $O(\log\log n)$-time *sum* and *findsum* queries in $size_s$ or num_s, and $O(\log\log n)$-time *rank* and *select* queries in pos_s. (These *rank* and *select* operations are analogous to *sum* and *findsum* queries, but we refer to them as *rank* and *select* for ease of exposition.) The list ptr_s is stored in a simple array.

[4] One may think of using a B-tree instead of a WBB-tree. However, in our design, a particular node in the WBB tree will need to store auxiliary information about every symbol in the subtree under that node. In the worst case, this auxiliary information will be as big as the size of the subtree. If we use a B-tree, the cost of updating a particular node cannot bounded by $O(n^{\epsilon})$ time in the amortized case.

We also maintain another B-tree B with fanout n^δ such that each leaf ℓ_s corresponds to a symbol s that is currently present in the text T. Each leaf stores the number of (nonzero) occurrences of s in T, along with a pointer to its corresponding s-structure in the root of W. The height of B is $O(\log_{n^\epsilon} |\Sigma|) = O(1)$, since we assume $|\Sigma| \leq n$.

Answering *char(i)*. We can answer this query in $O(\log \log n)$ time by maintaining a B-tree with fanout $b = n^\delta$ over the text. We call this tree the *text B-tree*.

Answering *rank$_s$(p)*. Recall that $rank_s(p)$ tells the number of occurrences of s in $T[1..p]$. We first query B to determine if s occurs in T. If not, return 0. Otherwise, we follow the pointer from B to its s-structure. We then perform $r.size_s.findsum(p)$ to determine the child c_i of root r from W that contains $T[p]$. Suppose that $T[p]$ is in the subtree rooted at the ith child c_i of r. Then, $rank_s$ consists of two parts: the number of occurrences $m_1 = r.num_s.sum(j)$ (with $j = r.pos_s.rank(i-1)$) in the first $i-1$ children of r, and m_2, the number of occurrences of s in c_i. If $r.pos_s.rank(i) \neq j+1$ (c_i contains no s symbols), return m_1. Otherwise, we retrieve the s-structure of c_i by its pointer $r.ptr[j+1]$ and continue counting the remaining occurrences of s before $T[p]$ in the WBB tree W. We will eventually return $m_1 + m_2$.

The above process either `(i)` stops at some ancestor of the leaf of $T[p]$ whose subtree does not contain s, in which case we can report the desired rank, or `(ii)` it stops at the level-1 node containing $T[p]$, in which case the number of remaining occurrences can be determined by a $rank_s$ query in the static text dictionary in $t(n) = O(\log \log |\Sigma|)$ time. Since it takes $O(\log \log n)$ time to check the B-tree B at the beginning, and it takes $O(\log \log n)$ time to descend each of the $O(1)$ levels in the WBB-tree to count the remaining occurrences, the total time is $O(\log \log n)$.

Answering *select$_s$(j)*. Recall that $select_s(j)$ tells the number of symbols (inclusive) before the jth occurrence of s in T. We follow a similar procedure to the above procedure for $rank_s$. We first query B to determine if s occurs at least j times in T. If not, we return -1. Otherwise, we discover the ith child c_i of root r from W that contains the jth s symbol. We compute $i = r.pos_s.select(r.num_s.findsum(j))$ to find out c_i.

Then, $select_s$ consists of two parts: the number of symbols $m_1 = r.size.sum(i)$ in the first $i-1$ children of r, and m_2, the number of symbols in c_i before the jth s. We retrieve the s-structure of c_i by its pointer $r.ptr[r.num_s.findsum(j)]$ and continue counting the remaining symbols on or before the jth occurrence of s in T. We will eventually return $m_1 + m_2$. The above process will stop at the level-1 node containing the jth occurrence of s in T, in which case the number of symbols on or before it maintained by this level-1 node can be determined by a $select_s$ query in the static text dictionary in $t(n) = O(\log \log |\Sigma|)$ time. With similar time analysis as in $rank_s$, the total time is $O(\log \log n)$.

Updates. We can update the text B-tree in $O(n^\epsilon)$ time. We use a naive approach to handle updates due to the insertion or deletion of symbols in T: For each list in the WBB-tree and for each static text dictionary that is affected,

we rebuild it from scratch. In the case that no split, merge, or merge-then-split operation occurs in the WBB-tree, an insertion or deletion of s at $T[p]$ will affect the static text dictionary containing $T[p]$, and two structures in each ancestor node of the leaf containing $T[p]$: the *size* array and the s-structure corresponding to the inserted (deleted) symbol. The update cost is $O(n^\delta \log n) = O(n^\epsilon)$ for the static text dictionary and for each ancestor, so in total it takes $O(n^\epsilon)$ time.

If a split, merge, or merge-then-split operation occurs at some level-ℓ node v_ℓ in the WBB-tree, we need to rebuild the *size* array and s-structures for all newly created nodes, along with updating the *size* array and s-structures of the parent of v_ℓ. In the worst case, it requires $O(n^{(\ell+1)\epsilon} \log n)$ time. By the property of WBB trees, the amortized update takes $O(n^\epsilon)$ time.

In summary, each update due to an insertion or deletion of symbols in T can be done in amortized $O(n^\epsilon)$ time.

Space complexity. The space for the text B-tree is $O(n \log |\Sigma| + n^{1-\epsilon} \log n)$ bits. The total space of all $O(n^{1-\epsilon})$ static text dictionaries can be bounded by $s(n) = O(n \log |\Sigma|)$ bits.

For the space of the s-structures, it seems like it is $O(|\Sigma| n^{1-\epsilon} \log n)$ bits at the first glance, since there are $O(n^{1-\epsilon})$ nodes in W. This space however is not desirable, since $|\Sigma|$ can be as large as n. In fact, a closer look of our design reveals that each node in W only maintains s-structures for those s that appears in its subtree. In total, each character of T contributes to at most $O(1)$ s-structures, thus incurring only $O(\log n)$ bits. The total space for s structures is thus bounded by $O(n \log n)$ bits.

The space for the B-tree B (maintaining distinct symbols in T) is $O(|\Sigma| \log n)$ bits, which is at most $O(n \log n)$ bits. In summary, the total space of the above dynamic rank-select structure is $O(n \log n)$ bits.

Summarizing the above discussions, we arrive at the following theorem.

Theorem 2. *For a dynamic text T of length at most $O(n)$, we can maintain a data structure on T using GMR to support rank_s, select_s, and char $O(t(n) + \log\log n) = O(\log\log n)$ time, and insertion/deletion of a symbol in amortized $O(n^\epsilon)$ time. The space of the data structure is $O(n \log n)$ bits.* □

Theorem 3. *Suppose that $|\Sigma| = \mathrm{polylog}(n)$. For a dynamic text T of length at most $O(n)$, we can maintain a data structure on T using the wavelet tree to support rank_s, select_s, and char in $O(t(n)) = O(1)$ time, and insertion/deletion of a symbol in amortized $O(n^\epsilon)$ time. The space of the data structure is $O(|\Sigma| n \log n)$ bits, and the working space to perform the updates at any time is $O(n^\epsilon)$ bits.* □

3.4 The Final Data Structure

Here we describe our final structure, which supports insertions and deletions of any symbol. To do this, we maintain two structures: our inX structure on $\hat{T}$ and the onlyX structure, where all of the new symbols are actually inserted and maintained. After every $O(n^{1-\epsilon} \log n)$ update operations, the onlyX structure is merged into the original text T and a new T is generated. All associated data structures are also rebuilt. Since this construction process could take at

most $O(n \log n)$ time, this cost can be amortized to $O(n^\epsilon)$ per update. The StaticRankSelect structure on T takes $s(n) = n \log |\Sigma| + o(n \log |\Sigma|)$ bits of space. With this frequent rebuilding, all of the other supporting structures take only $o(n)$ bits of space.

We augment the above two structures with a few additional BitIndel structures. In particular, for each symbol s, we maintain a bitvector I_s such that $I_s[i] = \mathbf{1}$ if and only if the ith occurrence of s is stored in the onlyX structure. With the above structures, we quickly describe how to support $rank_s(i)$ and $select_s(i)$.

For $rank_s(i)$, we first find $j = inX.rank_s(i)$. We then find $k = inX.rank_{\mathbf{x}}(i)$ and return $j + onlyX.rank_s(k)$. For $select_s(i)$, we first find whether the ith occurrence of c belongs to the inX structure or the onlyX structure. If $I_s[i] = \mathbf{0}$, this means that the ith item is one of the original symbols from T; we query $inX.select_s(j)$ in this case, where $j = I_s.rank_{\mathbf{0}}(i)$. Otherwise, we compute $j = I_s.rank_{\mathbf{1}}(i)$ to translate i into its corresponding position among new symbols. Then, we compute $j' = onlyX.select_s(j)$, its location in $\hat{T}$ and return $inX.select_x(j')$.

Finally, we show how to maintain I_s during updates. For $delete(i)$, compute $\hat{T}[i] = s$. We then perform $I_s.delete(inX.rank_s(i))$. For $insert_s(i)$, after inserting s in $\hat{T}$, we insert it into I_s by performing $I_s.insert_{\mathbf{1}}(inX.rank_s(i))$. Let $n_{\mathbf{x}}$ be the number of symbols stored in the onlyX structure. We can bound the space for these new BitIndel data structures using RRR [15] and Jensen's inequality by $\lceil \log \binom{n'}{n_{\mathbf{x}}} \rceil + o(n') = O(n^{1-\epsilon} \log^2 n) + o(n) = o(n)$ bits of space. Thus, we arrive at the following theorem.

Theorem 4. *Given a text T of length n drawn from an alphabet Σ, we create a data structure using GMR that takes $s(n) = n \log |\Sigma| + o(n \log |\Sigma|) + o(n)$ bits of space and supports $rank_s(i)$, $select_s(i)$, and $char(i)$ in $O(\log \log n + t(n)) = O(\log \log n + \log \log |\Sigma|)$ time and $insert(i)$ and $delete(i)$ updates in $O(n^\epsilon)$ time.* □

For the special case when $|\Sigma| = \text{polylog}(n)$, we may now use [10] as the StaticRankSelect structure, and the Constant Time BitIndel as the BitIndel structure. For the onlyX structure, we can use a similar improvement (using separate select structures for each symbol $s \in \Sigma$) as with BitIndel to achieve $O(1)$ time queries. The space required is $o(n)$ if merging is performed every $O(n^{1-\epsilon})$ update operations. We defer the details of this modification until the full paper. Then, we achieve the following theorem.

Theorem 5. *Given a text T of length n drawn from an alphabet Σ, with $|\Sigma| = \text{polylog}(n)$, we create a data structure using the wavelet tree that takes $s(n) + o(n) = nH_0 + o(n \log |\Sigma|) + o(n)$ bits of space and supports $rank_s(i)$, $select_s(i)$, and $char(i)$ in $O(t(n)) = O(1)$ time and $insert(i)$ and $delete(i)$ updates in $O(n^\epsilon)$ time.* □

We skip the details about the memory allocation issues for our dynamic structures and rebuilding space issues. However, the overhead for these issues can be shown to be $o(n)$ bits of additional space.

4 Dynamizing Ordinal Trees, Labeled Trees, and the XBW Transform

In this section, we describe applications of our BitIndel data structure and our dynamic multi-symbol rank/select data structure to dynamizing ordinal trees, labeled trees, and the XBW transform [2].

Ordinal Trees. An *ordinal tree* is a rooted tree where the children are ordered and specified by their rank. An ordinal tree can be represented by the Jacobson's LOUDS representation [1] using just *rank* and *select*. Thus, we can use our BitIndel data structure to represent any ordinal tree with the following operations: $v.parent()$, which returns the parent node of v in T; $v.child(i)$, which returns the ith child node of v; $v.insert(k)$, which inserts the kth child of node v; and $v.delete(k)$, which removes the kth child of node v.

Lemma 7. *For any ordinal tree T with n nodes, there exists a dynamic representation of it that takes at most $2n+O(n \log\log n/\log n)$ bits of space and supports updates in amortized $O(n^\epsilon)$ time and navigational queries in $O(\log\log n)$ time. Alternatively, we can take $6n + O(n \log\log n/\log n)$ bits of space and support navigational queries in just $O(1)$ time.* □

Labeled Trees, Text Collections, and XBW. A labeled tree T is a tree where each of the n nodes is associated with a label from alphabet Σ. To ease our notation, we will also number our symbols from $[0, |\Sigma|-1]$ such that the sth symbol is also the sth lexicographically-ordered one. We'll call this symbol s. We are interested in constructing a data structure such that it supports the following operations in T: $insert(P)$, which inserts the path P into T; $v.delete()$, which removes the root-to-v path for a leaf v; $subpath(P)$, which finds all occurrences of the path P; $v.parent()$, which returns the parent node of v in T; $v.child(i)$, which returns the ith child node of v; and $v.child(s)$, which returns any child node of v labeled s.

Ferragina et al. [2] propose an elegant way to solve the static version of this problem by performing an XBW transform on the tree T, which produces an XBW text S. They show that storing S is sufficient to support the desired operations on T efficiently, namely navigational queries in $O(\log|\Sigma|)$ time and $subpath(P)$ queries in $O(|P|\log|\Sigma|)$ time.

In the dynamic case when we want to support insert or delete of a path of length m, we observe that either operation corresponds to an update of this XBW text S at m positions. Using our dynamic framework, we can then maintain a dynamic version of this text S and achieve the following result using GMR.

Theorem 6 (Dynamic XBW). *For any ordered tree T, there exists a dynamic succinct representation of it using the XBW transform [2] that takes at most $s(n)+2n = n\log|\Sigma|+o(n\log|\Sigma|)+2n$ bits of space, while supporting navigational queries in $O(t(n)+\log\log n) = O(\log\log n)$ time. The representation can also answer a $subpath(P)$ query in $O(m(t(n)+\log\log n)) = O(m\log\log n)$ time, where m is the length of path P. The update operations $insert(P)$ and $delete()$*

at node u for this structure take $O(n^\epsilon + m(t(n) + \log\log n))$ amortized time, where m is the length of the path P being inserted or deleted. □

Acknowledgements. We would like to thank Gonzalo Navarro and S. Muthukrishnan for helpful discussions and reviews of this work.

References

1. D. Benoit, E. Demaine, I. Munro, R. Raman, V. Raman, and S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.
2. P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pages 184–196, 2005.
3. P. Ferragina and G. Manzini. On compressing and indexing data. *JACM*.
4. A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *SODA*, pages 368–373, 2006.
5. R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *SODA*, January 2003.
6. R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *STOC*, volume 32, May 2000.
7. T. Hagerup, P. Miltersen, and R. Pagh. Deterministic dictionaries. *Journal of Algorithms*, 41(1):353–363, 2001.
8. W. K. Hon, K. Sadakane, and W. K. Sung. Succinct data structures for searchable partial sums. In *ISAAC*, pages 505–516, 2003.
9. G. Jacobson. Succinct static data structures. Technical Report CMU-CS-89-112, Dept. of Computer Science, Carnegie-Mellon University, January 1989.
10. G. Navarro, P. Ferragina, G. Manzini, and V. Mäkinen. Succinct representation of sequences and full-text indexes. *TALG*, 2006. To appear.
11. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 2006. To appear.
12. G. Navarro and V. Mäkinen. Dynamic entropy-compressed sequences and full-text indexes. In *CPM*, pages 306–317, 2006.
13. M. Patrascu and E. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM Journal on Computing*, 35(4):932–963, 2006.
14. R. Raman, V. Raman, and S. Rao. Succinct dynamic data structures. In *WADS*, pages 426–437, 2001.
15. R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *SODA*, pages 233–242, 2002.
16. K. Sadakane and R. Grossi. Squeezing succinct data structures into entropy bounds. In *SODA*, pages 1230–1239, 2006.