Random Access in Persistent Strings

Philip Bille
Technical University of Denmark, DTU Compute, Lyngby, Denmark
phbi@dtu.dk

Inge Li Gørtz
Technical University of Denmark, DTU Compute, Lyngby, Denmark
inge@dtu.dk

Abstract

We consider compact representations of collections of similar strings that support random access queries. The collection of strings is given by a rooted tree where edges are labeled by an edit operation (inserting, deleting, or replacing a character) and a node represents the string obtained by applying the sequence of edit operations on the path from the root to the node. The goal is to compactly represent the entire collection while supporting fast random access to any part of a string in the collection. This problem captures natural scenarios such as representing the past history of an edited document or representing highly-repetitive collections. Given a tree with n nodes, we show how to represent the corresponding collection in $O(n)$ space and optimal $O(\log n / \log \log n)$ query time. This improves the previous time-space trade-offs for the problem. To obtain our results, we introduce new techniques and ideas, including a reduction to a new geometric line segment selection together with an efficient solution.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Data compression, data structures, persistent strings

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2020.48

Related Version A full version of the paper is available at `https://arxiv.org/abs/2006.15575`.

Funding Supported by the Danish Research Council (DFF – 4005-00267, DFF – 1323-00178).

Acknowledgements We thank the anonymous reviewers for their helpful comments that improved the presentation of this paper.

1 Introduction

The *random access problem* is to preprocess a data set into a compressed representation that supports fast retrieval of any part of the data without decompressing the entire data set. The random access problem is a well-studied problem for many types of data and compression schemes [1, 3, 5, 8, 9, 19, 31, 35, 41, 48, 53] and random access queries is a basic primitive in several algorithms and data structures on compressed data, see e.g., [7, 9, 23, 24, 25]

In this paper, we consider the random access problem on collections of strings where each string is the result of an *edit operation*, i.e., insert, delete, or replace a single character, from another string in the collection. Specifically, our collection is given by a rooted tree, called a *version tree*, where edges are labeled by an edit operation, the root represents the empty string, and a node represents the string obtained by applying the sequence of edit operation on the path from the root to the node (see Figure 1(a)). We call such a collection a *persistent string* since we can naturally view it as persistent versions of a single string. Given a node v and an index j, a random access query returns the character at position j in the string represented by v.

Random access in persistent strings captures natural scenarios for collections of similar strings. For instance, consider the problem storing and accessing the past history of edits in a document. Instead of explicitly storing all versions of the document, we can represent

the entire history compactly as a path of updates. Random access in a past version of the document then corresponds to a random access query on the corresponding node on the path. In our setup we can even support branching in the history of the document, as in version control systems, to form a tree of document histories. As another example, consider storing and accessing a collection of related genome sequences. If we know (a good approximation of) the edit distance between the pairs of genome sequences, we can construct a small version tree representing the collection from the minimum spanning tree of the pairs of distance. Again, random access in a sequence in the collection corresponds to a random access query on the corresponding node.

To the best our knowledge, no previous work has explicitly considered random access on persistent strings, but several well-known techniques and results can be combined to provide non-trivial bounds on the problem. In this paper, we introduce a new representation of persistent strings that supports random access. Our representation uses $O(n)$ space and supports random access queries in $O(\log n / \log \log n)$ time, where n is the number of nodes in the version tree (or equivalently the number of strings in the collection). This improves the best known combinations of time and space among all previous solutions. Furthermore, we prove that any solution that uses $n \log^{O(1)} n$ space needs $\Omega(\log n / \log \log n)$ query time, thus showing that our query time is optimal. To obtain our results, we introduce new techniques and ideas, including a reduction to a new geometric line segment selection problem together with an efficient solution to this problem.

1.1 Previous Work

To the best of our knowledge no previous work has explicitly considered supporting random access in persistent strings. However, several existing approaches can be applied or extended to obtain non-trivial solutions to the problem and several related models of repetitiveness have been proposed. We discuss these in the following. To state the bounds, let T be a version tree with n nodes representing a collection of n strings of total size N. Since any string represented by a node in T can be the result of at most n insertions we have that $N = O(n^2)$. Hence, naively we can solve the random access problem by explicitly storing all strings using $O(N) = O(n^2)$ space and $O(1)$ query time. With techniques from either persistent or compressed data structures we can significantly improve this as discussed below.

Persistent Data Structures and Dynamic Strings

Ordinary data structures are *ephemeral* in the sense that updating the data structure destroys the old version and only leaves the new version. A data structure is *persistent* if it preserves old versions of itself and allows queries and/or updates to them. In *partial persistence* we allow queries on all versions but only updates on the newest version, and in *full persistence* we allow queries and updates on all versions. Thus, in partial persistence the versions form a path whereas in full persistence the versions form a tree called the *version tree*. Persistent data structures is a classic data structural concept and were first formally studied by Driscoll et al. [16].

A *dynamic string* data structure supports the edit operations (insert, delete, and replace) and access to any character in the string. An immediate approach to solve the random access problem in persistent strings is to make a dynamic string data structure fully persistent. To do so, we simply traverse the version tree and perform the edit operations on the edges. To answer a random access query on a string represented by a node v we simply perform a persistent access operation on the version of the data structure corresponding to version v.

Depending on the dynamic string data structure we obtain different time-space trade-offs for the random access problem. A balanced binary search tree implements a dynamic string data structure using $O(\log n)$ time for all operations. Since binary search trees are constant degree pointer data structures a classic transformation by Driscoll et al. [16] immediately implies an $O(\log n)$ time solution for access. Since each persistent update to the binary search tree incurs $O(\log n)$ space overhead this leads to a total space of $O(n \log n)$. With a more careful implementation of binary search trees the space can be improved to $O(n)$ [16, 49].

Maintaining a dynamic string (often called the list representation or list indexing problem [14, 20]) is well-studied and closely connected to the partial sums problem. Dietz [14] presented the first solution achieving $O(\log n / \log \log n)$ time for access and updates and Fredman and Saks [20] showed in their seminal paper on cell probe complexity that this bound is optimal. Several variations and extension have been proposed [5, 6, 17, 32, 44, 45, 46]. However, all of these solutions rely on word RAM techniques and therefore incur an overhead of $\Omega(\log \log n)$ time to make them persistent [13] thus leading to a solution to the random access problem with query time $\Theta(\log n)$.

Compressed Representations

The classic Lempel-Ziv compression scheme (LZ77) [54] compresses an input string S by parsing S into z substrings $f_1 f_2 \dots f_z$, called *phrases*, in a greedy left-to-right order. Each phrase is either the first occurrence of a character or the longest substring that has at least one occurrence starting to the left of the phrase. By replacing each phrase by a reference to the previous occurrences we obtain a compressed representation of the string of length $O(z)$.

We can use LZ77 compression to efficiently store all versions of the persistent string in the random access problem. To do so, we write all the strings represented in the version tree T and concatenate them in order of increasing depth in T. The string represented by a node v can be formed from the string of the parent of v by at most 3 substrings, namely, the substrings before and after the edit operation and a new character in case of a replace or insert operation. Since we concatenate the strings in increasing depth it follows that the greedy LZ77 parsing uses at most $z = O(n)$ phrases.

To solve the random access problem on the persistent string we can convert the LZ77 compressed representation into a small grammar representation and then apply efficient random access results for grammars. Converting the LZ77 compressed string leads to a grammar of size $O(z \log(N/z)) = O(n \log n)$ [10, 47]. Using the best known trade-offs for random access in grammars, this leads to solutions using either $O(n \log n)$ space and $O(\log N) = O(\log n)$ query time [9] or $O(n \log^{1+\varepsilon} n)$ space and $O(\log N / \log \log N) = O(\log n / \log \log n)$ query time [3, 27]. We note that both of these results inherently need superlinear space for the conversion from LZ77 to grammars [10]. Furthermore, Verbin and Yu [53] showed that the latter query time is optimal. More precisely, they proved that any representation of an LZ77 compressed string using $z \log^{O(1)} N = n \log^{O(1)} n$ space must use $\Omega(\log N / \log \log N) = \Omega(\log n / \log \log n)$ time.

A related simpler model of compression is *relative compression* [50, 51] (see also [5, 12, 15, 33, 36, 37, 38, 39]), where we explicitly store a single reference string and compress a collection of strings as substrings of the reference string. A similar compression model is also proposed in [26, 40, 42, 43]. The relative compression model compresses efficiently if each string is the result of applying a small number of edits to the base string. In contrast, using persistent strings we can compress efficiently if each string is the result of editing *any* other string in the collection.

1.2 Setup and Results

Let T be a version tree with n nodes. Each node v of T represents a string $S(v)$ and each edge is labeled by one of the following edit operations:

- $\mathsf{replace}(k, \alpha)$: change the kth character to α.
- $\mathsf{insert}(k, \alpha)$: insert character α immediately after position k.
- $\mathsf{delete}(k)$: delete the character at position k.

The string represented by the root is the empty string ε, and the string represented by a non-root node v is the result of applying all edit operations on the path from the root to v on the empty string. Our goal is to preprocess T into a compact data structure that supports the query $\mathsf{access}(v, j)$, that returns $S(v)[j]$. Our main result is a new representation of persistent strings that achieves the following bound:

▶ **Theorem 1.** *Given a version tree T with n nodes we can solve the random access problem in $O(n)$ space and $O(\log n / \log\log n)$ time. Furthermore, we can report a substring of length ℓ using $O(\ell)$ additional time.*

Theorem 1 simultaneously matches the best known space and time bounds of the previous approaches. In particular, compared to the classic persistent binary search [16] we match the space while improving the $O(\log n)$ query time to $O(\log n / \log\log n)$. On the other hand, compared to the recent results on random access in grammar-compressed [3, 27] we match the query time while improving the space from $O(n \log^{1+\varepsilon} n)$ to linear. Furthermore, we show that Theorem 1 is optimal for any near-linear space solution.

▶ **Theorem 2.** *Any data structure that solves the random access problem on a version tree T with n nodes using $n \log^{O(1)} n$ space needs $\Omega(\log n / \log\log n)$ query time.*

Note that Theorem 2 holds even in the special case when T is a path such as in the example with storing and accessing the past history of edits in a document.

1.3 Techniques

To achieve our result we introduce several techniques and data structures of independent interest. First, we show how to reduce random access queries on a persistent string to a geometric problem on horizontal line segments, that we call the *segment selection problem*. The main idea is to traverse the version tree in a depth-first traversal and produce segments representing characters appearing in the versions of the persistent strings. The x-coordinates of the segments correspond to the traversal time interval and the y-coordinates correspond to the left-to-right ordering of the characters in the strings. We show how to construct segments such that at any point in time i, the segments crossing the vertical line through x-coordinate $2i$ corresponds to the string represented at the node in T first visited at time i. Thus, to answer a random access query on $S(v)[j]$ it suffices to answer a *segment selection query*, that given integers i and j, returns the jth segment crossing the vertical line at i', where i' is time corresponding to v and $i = 2i'$.

Next, we show how to efficiently solve the segment selection problem in linear space and $O(\log n / \log\log n)$ query time thus implying Theorem 1. To do so, the main idea is to build a balanced tree of degree $\Delta = O(\log^{\varepsilon} n)$ and of height $O(\log_{\Delta} n) = O(\log n / \log\log n)$ that stores the segments ordered by y-coordinate. Each internal node thus partitions the segments below it into Δ horizontal bands called *slabs*.

To answer a segment selection query (i, j) we traverse the tree to find the leaf containing the jth segment that crosses the vertical line at time i. To implement the traversal we need to determine at each node v the slab containing the desired segment among the segments

below v at the specified time i. The key challenge is to compactly represent the segments while achieving constant query time to find the correct slab at each node. Using well-known techniques we can solve this *slab selection problem* with an explicit representation of segments below v in constant time and $O(n_v)$ of space, where n_v is the number of segments below v. Unfortunately, this leads to a solution to segment selection that uses $O(n \log_\Delta n) = O(n \log n / \log\log n)$ space. We show how to compactly represent the segments to significantly improve the space to $O(n_v \log\log n)$ *bits* while simultaneously achieving constant time queries. In turn, this implies a solution to segment selection using $O(n)$ space and $O(\log_\Delta n) = O(\log n / \log\log n)$ query time.

Finally, we prove a matching lower bound for the random access in persistent strings problem by showing that any solution using $n \log^{O(1)} n$ space needs $\Omega(\log n / \log\log n)$ query time. To do so we show a simple reduction from the *range selection problem* [34] that holds even in the case when the version tree is a path.

1.4 Outline

We present the reduction from random access to segment selection in Section 2 and our solution to the slab selection problem in Section 3. We then use our slab selection data structure in our full data structure for the segment selection problem in Section 4. Plugging this into our reduction leads to Theorem 1. We show the lower bound in Section 5 and conclude with some open problems in Section 6.

2 Reducing Random Access to Segment Selection

In this section we show how to reduce the random access problem to the following natural geometric selection problem on line segments. Let L be a set of n horizontal line segments in the plane. The *segment selection problem* is to preprocess L to support the operation:

- segment-select(i, j): return the jth smallest segment (the segment with the jth smallest y-coordinate) among the segments crossing the vertical line through x-coordinate i.

We will view the x-axis as a timeline and often refer to an x-coordinate i as time i. We will show how to efficiently solve the segment selection problem in the following sections. Our reduction from the random access problem works as follows. Let T be an instance of the random access problem with n nodes and assume wlog. that T contains no edges labeled by replace. We can do so since we can always convert edges labeled by replace into two edges labeled by a delete and insert, thus at most doubling the size of the instance. We construct an instance L of segment selection as follows.

We first perform an Euler tour [52] of T to construct a sequence $\overline{S}_0, \ldots, \overline{S}_{2n-2}$ of strings corresponding to each time we meet a node in the Euler tour. We call these strings *marked strings* since each character in them will be either *marked* or *unmarked*. The marked strings are defined as follows. String $\overline{S}_0$ is the empty string. Suppose we have constructed $\overline{S}_0, \ldots, \overline{S}_{\ell-1}$ and let e be the edge visited at time ℓ in the Euler tour. We construct $\overline{S}_\ell$ from $\overline{S}_{\ell-1}$ according to the following cases (see Figure 1(b) for an example).

Case 1: Insertions. Suppose that e is labeled insert(i, α). If we traverse e in the downward direction, we insert character α as an unmarked character in $\overline{S}_{\ell-1}$ immediately to the right of the ith unmarked character to get $\overline{S}_\ell$. If we traverse e in the upwards direction we mark the same character that was inserted as an unmarked character in the earlier downwards traversal of e.

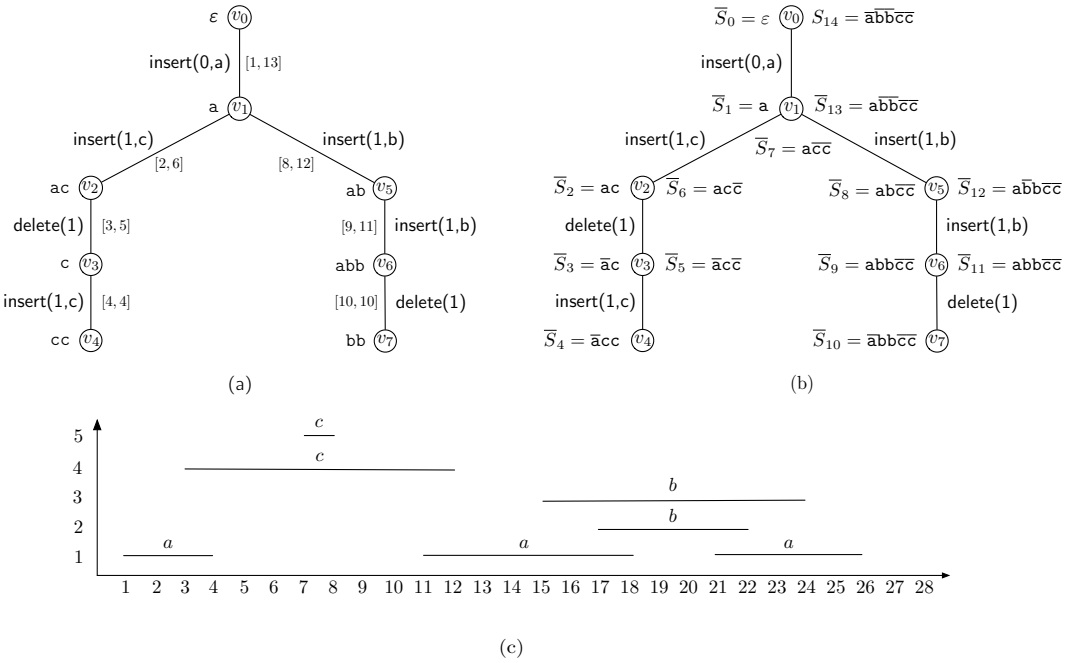

Figure 1 (a) A persistent string representing the collection $\{\epsilon, \mathtt{a}, \mathtt{ac}, \mathtt{c}, \mathtt{cc}, \mathtt{ab}, \mathtt{abb}, \mathtt{bb}\}$. The interval $I(e)$ is shown for each edge. (b) The marked strings of (a). The insertion edges are unmarked in the following intervals: (v_0, v_1) in $[1, 2] \cup [6, 9] \cup [11, 13]$, (v_1, v_2) in $[2, 6]$, (v_3, v_4) in $[4, 4]$, (v_1, v_5) in $[8, 12]$, and (v_5, v_6) in $[9, 11]$. (c) The segment selection instance corresponding to (a). The range of x-coordinates of segments are obtained by converting each interval $[i, j]$ above to $[2i - 1, 2j]$.

Case 2: Deletions. Suppose that e is labeled $\mathsf{delete}(i)$. If we traverse e in the downward direction, we mark the ith unmarked character in $\overline{S}_{\ell-1}$ to get $\overline{S}_\ell$. If we traverse e in the upward direction, we unmark the same character that was marked in the downward traversal of e.

Note that an insertion edge e traversed in the downward direction at time ℓ results in an insertion of a character, denoted $\mathrm{char}(e)$, in $\overline{S}_\ell$. Since $\mathrm{char}(e)$ is never removed from subsequent marked strings it appears in all subsequent strings $\overline{S}_\ell, \ldots, \overline{S}_{2n-2}$, but changes between being marked and unmarked. If a deletion edge e' changes $\mathrm{char}(e)$ from unmarked to marked we say that e' *deletes* $\mathrm{char}(e)$.

For an edge e in T, let $\mathrm{first}(e)$ and $\mathrm{last}(e)$ denote the first and last time, respectively, we visit v in the Euler tour of T, and let $I(e) = [\mathrm{first}(e), \mathrm{last}(e) - 1]$ denote the interval of e.

▶ **Lemma 3.** *Let e be an insertion edge in T that is traversed in the downward direction at time ℓ and let $e_1, \ldots, e_m$ be the edges in $T(v)$ that delete $\mathrm{char}(e)$. Then, $\mathrm{char}(e)$ is unmarked in all strings $\overline{S}_i$ where i is an integer in the interval $I(e) \setminus (I(e_1) \cup \cdots I(e_m))$ and marked in $\overline{S}_i$ for all other integers i in $[\ell, 2n - 2]$.*

Proof. We have that $\mathrm{char}(e)$ appears in $S_\ell, \ldots, S_{2n-2}$. The edge e inserts $\mathrm{char}(e)$ as unmarked in the interval $I(e)$ and each edge e' that deletes $\mathrm{char}(e)$, marks it in the interval $I(e')$. ◀

For instance, consider $e = (v_0, v_1)$ in Figure 1(a) that inserts an $\mathtt{a}$ which is then deleted by $e_1 = (v_3, v_2)$ and $e_2 = (v_7, v_6)$. Thus, $\mathtt{a}$ appears in the interval $[1, 13]$ and is unmarked in $I(e) \setminus (I(e_1) \cup I(e_2)) = [1, 13] \setminus ([3, 5] \cup [10, 10]) = [1, 2] \cup [6, 9] \cup [11, 13]$.

For a node v in T, let $\text{start}(v) = \text{first}((\mathsf{parent}(v), v))$ denote the first time we meet v in the Euler tour of T. For the root r we define $\text{start}(r) = 0$.

▶ **Lemma 4.** *For any v, the concatenation of the unmarked characters in $\overline{S}_{\text{start}(v)}$ is $S(v)$.*

Proof. From the Lemma 3, the unmarked characters in $\overline{S}_{\text{start}(v)}$ are those which have been inserted at an edge $(w, \mathsf{parent}(w))$ where w is ancestor of v and have not been marked by any deletion edge in between. By definition these are the same characters as $S(v)$. From the insertion ordering of the characters in the marked strings it follows that characters in $\overline{S}_{\text{start}(v)}$ and $S(v)$ appear in the same order. ◀

Next, we construct a set of labeled line segments L from $\overline{S}_{2n-2}$ as follows. Note that $\overline{S}_{2n-2}$ consists of all of the (marked) characters appearing at insertion edges in T. For each insertion edge e, define $\text{pos}(e)$ to be the position of $\text{char}(e)$ is $\overline{S}_{2n-2}$. For instance, in Figure 1(a) $\text{pos}((v_1, v_0)) = 1$ since $\mathtt{a}$ is at position 1 in $\overline{S}_{14}$. For each insertion edge e in T that is deleted by edges $e_1, \ldots, e_m$, we construct $m+1$ horizontal line segments corresponding to the $m+1$ time intervals where $\text{char}(e)$ is unmarked. These $m+1$ segments are all labeled by $\text{char}(e)$ and all have y-coordinate $\text{pos}(e)$. For an interval $[i, j]$ the corresponding segment has x-coordinates $2i-1$ and $2j$. We use $2i-1$ and $2j$ to ensure that all segments have length at least one and that no two segments share an endpoint. See Figure 1(b). For instance, the insertion edge $e = (v_0, v_1)$ has position 1 and two deletion edges producing the 3 segments in Figure 1(b) labeled $\mathtt{a}$. We have the following correspondence between T and L.

▶ **Lemma 5.** *Let T be a version tree and let L be the corresponding instance of the segment selection. Then, $S(v)$ is the concatenation labels of the segments crossing the vertical line at time $2 \cdot \text{start}(v)$ ordered by increasing y-coordinate.*

Proof. We first show that the vertical line at $2 \cdot \text{start}(v)$ crosses exactly the segments corresponding to unmarked characters in $\overline{S}_{\text{start}(v)}$. By the definition of the intervals and the segments it is enough to show that $i \leq \text{start}(v) \leq j$ if and only if $2i - 1 \leq 2 \cdot \text{start}(v) \leq 2j$. This follows immediately from the fact that i, j, and $\text{start}(v)$ are integers. By the definition of $\text{pos}(e)$ the order of the segments is the same as the order of the corresponding unmarked characters in $\overline{S}_{\text{start}(v)}$. Thus the segments crossing the vertical line at time $2 \cdot \text{start}(v)$ in increasing order is the concatenation of the unmarked characters in $\overline{S}_{\text{start}(v)}$. By Lemma 4 this is $S(v)$. ◀

Each edge in T increases the number of segments in L by at most 1 and hence L contains at most $n-1$ segments. To answer $\mathsf{access}(v, j)$ on T we compute $\mathsf{segment\text{-}select}(2 \cdot \text{start}(v), j)$ on L and return the corresponding label. By Lemma 5 this correctly returns $S(v)[j]$. In summary, we have the following result.

▶ **Lemma 6.** *Given a solution to the segment selection problem on n segments that uses $s(n)$ space and answers queries in $t(n)$ time, we can solve the random access problem in $O(s(n))$ space and $O(t(n))$ time.*

3 Selection in Slabs

In this section, we introduce the *slab selection problem* and present an efficient solution. Our data structure will be a key component in our full solution to the segment selection problem that we present in the next section. As before we will view the x-axis as a timeline and often refer to an x-coordinate i as time i.

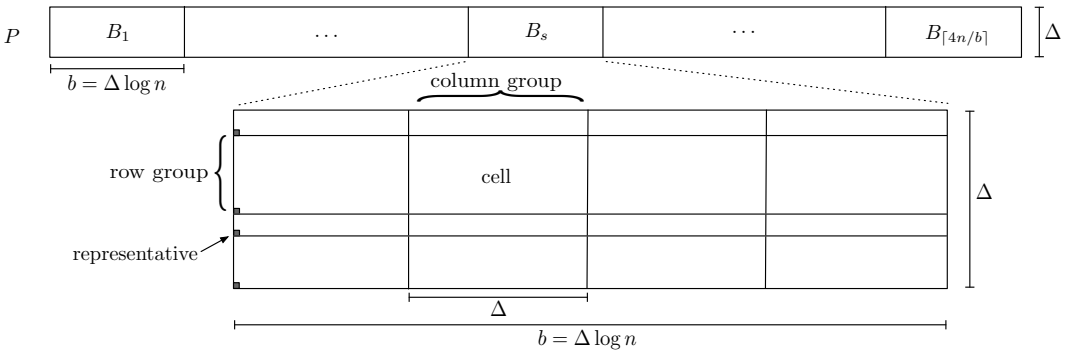

Figure 2 The grid P partitioned into blocks and a block of P partitioned into column groups, row groups, and cells.

Let L be a set of n segments given in the following "rank reduced" coordinates. The x-coordinates of the segment endpoints are unique integers from the set $\{1, \ldots, 4n\}$ and the y-coordinates are unique integers in $\{1, \ldots, n\}$. In particular, at every time at most one segment starts or ends. Note that the condition on the x-axis is satisfied in the reduction from Section 2. To satisfy the condition on the y-axis, we sort the segments according to their y-coordinate breaking ties according to their starting point on the x-axis, and use their rank in this ordering as y-coordinate. Note that this maintains the ordering among segments crossing the vertical at any time i.

We partition the segments L into $\Delta = O(\log^{\varepsilon} n)$, where $0 < \varepsilon < 1$, infinite horizontal bands $s_1, \ldots, s_\Delta$, called *slabs*. Each slab consists of $\lceil n/\Delta \rceil$ segments, except possibly s_Δ which may be smaller. The *slab selection problem* is to compactly represent L to support the following queries:

- slab-sum(i, j): return the total number of segments in slabs $s_1, \ldots, s_j$ crossing the vertical line through x-coordinate i.
- slab-select(i, j): return the smallest k such that slab-sum$(i, k) \geq j$.

The goal of this section is to construct a data structure for the slab selection problem that uses $O(n \log \log n)$ bits of space and answers slab-sum and slab-select queries in constant time. Note that if we explicitly represent each of the n segments, e.g., by their two x-coordinate endpoints and their y-coordinate, we need $\Omega(n \log n)$ bits even if we ignore how to support queries. We present a compact representation of the collection of segments that improves the space to $O(n \log \log n)$ bits and simultaneously achieves constant time queries.

Before presenting our data structure, we first convert the problem to a problem on a grid of prefix sums, define a decomposition on the grid, and show some key properties that we will need in our solution. We define a grid P of integers arranged in $4n$ columns and Δ rows such that the entries in column i represent the prefix sums of the number segments crossing at time i. We use $P(i, j)$ to denote the entry in column i and row j in P. More precisely, $P(i, j)$ contains the number of segments crossing i in slab s_1 to s_j. We have that slab-sum$(i, j) = P(i, j)$ and slab-select(i, j) corresponds to a predecessor query on column i, that is, computing the smallest k such that $P(i, k) \geq j$.

We decompose P as follows. Let $b = \Delta \log n$. We partition P into *blocks* $B_1, \ldots, B_{\lceil 4n/b \rceil}$ of b consecutive columns. We further partition each block B into groups of consecutive columns and rows called *column groups* and *row groups*, respectively (see Figure 2). The column groups are groups of Δ consecutive columns and the row groups are defined such that two adjacent rows are in the same row group if their leftmost entries differ by at most b.

Each rectangular subgrid in B given by the entries that are in the same column group and row group is called a *cell* of B. The *representative* of a row group in B is the bottom and leftmost position in the row group. The *representative* of a cell C in B is the representative of the row group of C. For any cell C in B, we define the *normalized cell*, denoted $\hat{C}$, to be C where all entries have been subtracted by the representative of C. We have the following properties of the construction.

▶ **Lemma 7.** *Let B be a block of the grid P. We have the following properties.*

(i) *Adjacent entries in a row differ by at most* 1.

(ii) *Adjacent entries in a column within the same row group differ by at most* $2b$.

(iii) *Entries in non-adjacent row groups differ by more than* b.

(iv) *Let r_j be the representative of row group j. Then, all entries in the first row of row group $j-1$ and below have values smaller than r_j and all entries in row group $j+1$ and above have values greater than r_j.*

Proof. (i) At any time at most one segment can start or end, which can only change the prefix sums in a column by ± 1. (ii) We have that adjacent entries in the leftmost column of the same row group differ by at most b. By (i) going left-to-right this difference can increase by at most 1 in each column. Since B has b columns the difference can be at most $2b$. (iii) Any two entries in the leftmost column in two non-adjacent row groups differ by more than $2b$. Each column contains at most one update and each update can reduce this difference by no more than 1. Hence, entries in non-adjacent row groups must differ by more than b. (iv) The difference between r_j and r_{j-1} is more than b. Consider the first row in row group $j-1$. Since B has b columns it follows from (i) that any entry in this row has value at most $r_{j-1}+b < r_j$. Since the grid contains prefix sums, the values in a column are non-decreasing. Thus, all entries below row group $j-1$ have values smaller r_j. Symmetrically, all entries in the first row of row group $j+1$ have value at least $r_{j+1}-b > r_j$. ◀

3.1 Data Structure

We store several data structures to represent P and support queries. For each block B we store the following.

- A predecessor data structure on the representatives of B. We use the fusion node structure for constant time predecessor queries on sets of polylogaritmic size due to Fredman and Willard [21, 22]. Since there are at most $\Delta = O(\log^{\varepsilon} n)$ representatives, this structure supports queries in constant time and uses $O(\Delta \log n) = O(b)$ bits of space.
- For each cell C, we store the leftmost column of the normalized cell $\hat{C}$. By Lemma 7 (i) the first entry in the leftmost column differs from the representative r by at most b. By Lemma 7 (ii) and since the height of C is at most Δ, the remaining entries in the leftmost column differ by at most $2b(\Delta-1)+b = O(b\Delta)$. We have $\log n$ column groups in B and thus the total height of all cells in B is $\Delta \log n = b$. Therefore, we can encode all leftmost columns in $O(b \log(b\Delta)) = O(b \log \log n)$ bits.
- For each column in B we store the difference from the previous column. We encode this as the number of the slab containing the update and a single bit indicating if the update is the start or end of a segment. This uses $b \lceil \log \Delta \rceil + 1 = O(b \log \log n)$ bits.

Combined we use $O(b \log \log n)$ bits for a block and thus $O(\frac{n}{b} b \log \log n) = O(n \log \log n)$ bits in total for P.

We will use our data structure to efficiently construct a compact encoding for any normalized cell $\hat{C}$. To do so, we combine the encoding of leftmost column of $\hat{C}$ and the encoding of the column differences/updates in the cell in left to right order.

We will use tabulation to support the following queries on normalized cells. Given a normalized cell $\hat{C}$ and integers i and j, define

- $\mathsf{access}(\hat{C}, i, j)$: return $\hat{C}(i, j)$.
- $\mathsf{predecessor}(\hat{C}, i, j)$: return the smallest k such that $\hat{C}(i, k) \geq j$.

We construct a single global table for each of the queries. The height of $\hat{C}$ is at most Δ, and by the argument above we can encode the leftmost column of $\hat{C}$ with $O(\Delta \log(b\Delta))$ bits. The rest of the columns are encoded by their difference from the previous column. Since the width of $\hat{C}$ is Δ this uses $O(\Delta \log \Delta)$ bits. Thus the encoding of $\hat{C}$ uses at most $O(\Delta \log(b\Delta) + \Delta \log \Delta) = O(\Delta \log \log n)$ bits. For access we encode the indices i and j using $O(\log \Delta)$ bits, and the answer in $O(\log(b\Delta))$ bits. Thus the total length of the encoding for an access query is $O(\Delta \log \log n) + \log \Delta + \log(b\Delta)$ bits. Hence, we can support access in constant time with a table of size $2^{O(\Delta \log \log n + \log \Delta + \log(b\Delta))} = 2^{O(\log^{\varepsilon} n \log \log n)} = o(n)$ bits (recall that $\varepsilon < 1$). We encode predecessor similarly except that the answer to the query can now be encoded in only $O(\log \Delta)$ bits. The total size the entire structure is $O(n \log \log n)$ bits.

3.2 Supporting Queries

We show how to implement $\mathsf{slab\text{-}sum}(i, j)$ and $\mathsf{slab\text{-}select}(i, j)$ in constant time. For both queries we find the block of P containing column i and the column group in the block corresponding to i. Since the blocks and column groups are evenly spaced this takes constant time. Let B be the block and let $r_1, \ldots, r_m$ be the sequence of representatives in B in increasing y-order. We then compute the predecessor r_ℓ of j among the representatives in constant time using the fusion node structure. This identifies the cell C_ℓ containing entry (i, j). To answer $\mathsf{slab\text{-}sum}(i, j)$, we compute the position (i', j') in C_ℓ corresponding to (i, j) and then compute the answer as

$$\mathsf{access}(\hat{C}_\ell, i', j') + r_\ell.$$

This correctly returns the value of $C(i, j)$ since $\hat{C}_\ell$ is normalized wrt r_ℓ.

To answer $\mathsf{slab\text{-}select}(i, j)$, we also consider the adjacent cells above and below C_ℓ, denoted $C_{\ell-1}$ and $C_{\ell+1}$, respectively. Since r_ℓ is the predecessor of j we have that $r_\ell \leq j < r_{\ell+1}$. By Lemma 7 (iii), entries in row groups below row group $\ell - 1$ have values smaller than r_ℓ and entries in row groups above row group $\ell + 1$ have values greater than $r_{\ell+1}$. Hence, the entry in column i containing the predecessor of j must be either in $C_{\ell-1}$, C_ℓ, or $C_{\ell+1}$. We can determine the correct cell in constant time using slab-sum queries on the topmost row of each of these cells. The correct cell is the lowest of these for which the slab-sum query returns a value of at least j. Let C denote the correct cell and let j'' be the topmost row in B in the row group immediately below C. We compute the answer as

$$\mathsf{predecessor}(\hat{C}, i', j - \mathsf{slab\text{-}sum}(i, j'')) + j''.$$

Both queries take constant time. In summary we have shown the following result.

▶ **Lemma 8.** *Let L be a set of n segments partitioned into $O(\log^{\epsilon} n)$ horizontal slabs. Then, we can solve the slab selection problem using $O(n \log \log n)$ bits of space and constant query time.*

4 Segment Selection

We now show how to solve segment selection in $O(n)$ space and $O(\log n/\log\log n)$ query time. In addition to our slab selection data structure from Section 3, we will also need a compact representation of strings that supports *rank* and *select queries* on polylogarithmic sized alphabets. Let A be a string of length n over an alphabet $[1,\sigma]$, and define the following queries:

- $\mathsf{rank}(A,i,j)$: return the number of occurrences of j in $X[1,i]$,
- $\mathsf{select}(A,i,j)$: return the position of the ith occurrence of character j.

Supporting rank and select on polylogarithmic sized alphabets is a well-studied problem, see e.g., [1, 2, 4, 18, 28, 29, 30]. Most of this work focuses on achieving constant time using succinct or compressed space. For our purposes we only need the following standard result which follows immediately from the above mentioned results.

▶ **Lemma 9.** *Let S be a string of length n from an alphabet of size $\sigma = O(\text{polylog}\, n)$. Then, we can represent S in $O(n\log\sigma)$ bits and support* rank *and* select *queries in $O(1)$ time.*

Next, we describe our data structure. Let L be a set of n segments. We assume that L is given in "rank space" as in the previous section. Otherwise, we can always convert L into this representation by standard rank reduction techniques. Let $\Delta = \log^{\varepsilon} n$, where $0 < \varepsilon < 1$. We construct a balanced tree R with degree Δ that stores the segments in L in the leaves in sorted y-order. The height of R is $O(\log_\Delta n) = O(\log n/\log\log n)$.

We introduce some helpful notation. Let v be an internal node with children $v_1,\ldots,v_\Delta$. The subtree rooted at v is denoted R_v, and the set of segments below v is denoted L_v. We let $n_v = |L_v|$. The endpoints in L_v are "rank reduced" to a grid of size $4n_v \times n_v$ in the following way. For an endpoint $p = (x,y)$ let r_x and r_y denote the rank of p when the endpoints are sorted by x-order and y-order, respectively. Then $p' = (2r_x - 1, r_y)$. Let L'_v denote the set of rank reduced segments. The *slab* of v, denoted $\text{slab}(v)$, is the narrowest infinite horizontal band containing L'_v. We number the slabs in increasing y-order. We partition the segments of L'_v into $\text{slab}(v_1),\ldots,\text{slab}(v_\Delta)$. At each internal node v we store the following:

- A string E_v of length $4n_v$ that, for each endpoint in L'_v in x-order, stores the slab containing it interleaved with 0's. More precisely, $E_v[i]$ is the number of the slab that contains the endpoint with x-coordinate i if i is odd, and 0 if i is even.
  We represent E_v as a rank/select structure according to Lemma 9. Since E_v is a string of length $4n_v$ over an alphabet of size Δ we use $O(n_v\log\Delta) = O(n_v\log\log n)$ bits of space and support rank and select queries in constant time.
- A slab selection structure according to Lemma 8 on L'_v with slabs $\text{slab}(v_1),\ldots,\text{slab}(v_\Delta)$. The slab selection structure uses $O(n_v\log\log n)$ bits of space and supports queries in constant time.

See Figure 3. At node v we use $O(n_v\log\log n)$ bits. Since each segment appears in $O(\log n/\log\log n)$ structures the total space is $O(n\frac{\log n}{\log\log n}\log\log n) = O(n\log n)$ bits.

To answer a $\mathsf{segment\text{-}select}(i,j)$ query we perform a top-down search in R starting at the root and ending at the leaf containing the jth segment that intersects the vertical line at time i. To guide the navigation, we compute *local parameters* i_v and j_v at each node v, such that i_v is the time in L_v that corresponds to time i in E, and j_v is the segment in L_v that corresponds to segment j in L. At the root r, we have $i_r = i$ and $j_r = j$. Consider an internal node v with children $v_1,\ldots,v_\Delta$ during the traversal. Given the local parameters i_v and j_v we compute the child to continue the search in and new local parameters. We first compute the slab containing the jth segment as

$$k = \mathsf{slab\text{-}select}(v,i_v,j_v) \qquad \text{and} \qquad j_{v_k} = j_v - \mathsf{slab\text{-}sum}(v,i_v,k-1)\ .$$

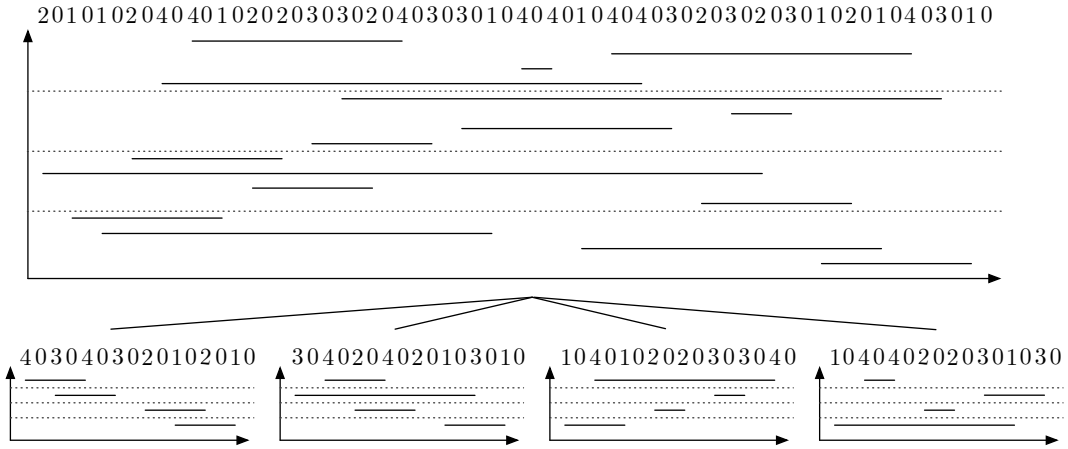

Figure 3 A node and its 4 children in the tree R corresponding to a partition of segments into $\Delta = 4$ slabs. The string E_v is shown at each node.

Thus, the search should continue in child v_k, and we subtract the number of segments in the previous slabs from j_v to get j_{v_k}. To compute i_{v_k} we first compute $r_k = \mathsf{rank}(E_v, i_v, k)$. Since i_v might not be a point in L_{v_k} we then set

$$i_{v_k} = \begin{cases} 2r_k - 1 & \text{if } E_v[i_v] = k \\ 2r_k & \text{otherwise} \end{cases}$$

By Lemma 8 and Lemma 9 each of the above steps takes constant time and hence the total time is $O(\log n / \log\log n)$. We have the following property of i_{v_k}.

▶ **Lemma 10.** *The segments from slab k in L'_v that are intersected by i_v are the same as the segments intersected by i_{v_k} in L'_{v_k}.*

Proof. Let I_v denote the set of segments from slab k in L'_v intersected by i_v and let I_{v_k} denote the set of segments in L'_{v_k} intersected by i_{v_k}. Let s a segment from L_{v_k} with x-coordinates (x_1, x_2) in L'_v and (x'_1, x'_2) in L'_{v_k}. Define $r_i = \mathsf{rank}(E_v, x_i, k)$. From the definition of the rank reduction we have $x'_i = 2 \cdot r_i - 1$. We will show that $s \in I_v$ iff $s \in I_{v_k}$.

First assume $s \in I_v$. Then $x_1 \leq i_v \leq x_2$, which implies that $\mathsf{rank}(E_v, x_1, k) \leq \mathsf{rank}(E_v, i_v, k) \leq \mathsf{rank}(E_v, x_2, k)$, that is $r_1 \leq r_k \leq r_2$. We need to prove that $x'_1 \leq i_{v_k} \leq x'_2$. If $E_v[i_v] = k$, i.e., i_v is an endpoint of a segment in slab k then it immediately follows that $x'_i = 2r_1 - 1 \leq 2r_k - 1 = i_{v_k}$ and similarly that $i_{v_k} \leq x'_2$. If $E_v[i_v] \neq k$ then i_{v_k} is not an endpoint in L_{v_k} and thus $x_1 < i_v < x_2$. This implies that $r_1 \leq r_k < r_2$. We have $r_1 = r_k$ in the case where x_1 is the rightmost endpoint in slab k smaller than i_v. It follows immediately that $2r_1 - 1 \leq 2r_k - 1 < 2r_k < 2r_2$, and therefore $x'_1 < i_{v_k} < x'_2$.

Assume $s \in I_{v_k}$. Then $x'_1 \leq i_{v_k} \leq x'_2$ and we want to prove that $x_1 \leq i_v \leq x_2$. We have $2r_1 - 1 \leq i_{v_k} \leq 2r_2 - 1$. We will first show that $r_1 \leq r_k \leq r_2$. There are two cases. If $E[i_v] = k$ then $i_{v_k} = 2r_k - 1$ and it follows immediately that $r_1 \leq r_k \leq r_2$. If $E[i_v] \neq k$ then $i_{v_k} = 2r_k$ and thus $2r_1 - 1 \leq i_{v_k} \leq 2r_2 - 1$ implies $2r_1 - 1 < 2r_k < 2r_2 - 1$ which again implies that $r_1 \leq r_k \leq r_2$. By definition of rank we have that $\mathsf{rank}(E_v, x_1, k) \leq \mathsf{rank}(E_v, i_v, k) \leq \mathsf{rank}(E_v, x_2, k)$ implies $x_1 \leq i_v \leq x_2$. ◀

In summary, we have the following result.

▶ **Theorem 11.** *Given a set of n horizontal segments in the plane, we can solve the segment selection problem in $O(n)$ space and $O(\log n / \log\log n)$ query time.*

Combined with the reduction in Lemma 6 we obtain a linear space and $O(\log n/\log\log n)$ time solution for the random access problem. To show Theorem 1 it only remains to show how to report a substring of length ℓ in $O(\log n/\log\log n+\ell)$ time. To do so we build the *hive graph* of Chazelle [11] on the segments. This uses $O(n)$ space and allows us to traverse the segments through the vertical line at time i above a given segment in sorted order in constant time per reported segment. To report a substring of length ℓ we simply perform the corresponding segment selection and traverse the ℓ segments above. By Lemma 5 this gives us the correct substring. This uses $O(\log n/\log\log n+\ell)$ time. This completes the proof of Theorem 1.

Finally, we show how to construct the random access data structure of Theorem 1 in $O(n\log n)$ time. Given a version tree T with n nodes it is straightforward to construct the corresponding instance of the segment selection problem L as described in Section 2 in $O(n)$ time in a single traversal of T. We then construct tree R over the segments in L recursively. At each node v we build the slab selection data structure from Section 3 consisting of n_v segments. To do so, we construct the grid, the predecessor data structure, and the compact encoding in $O(n_v)$ time. The global tables for the normalized cells need only be constructed once in $O(n)$ total time. Furthermore, we also need to build the rank/select data structure from Lemma 9. This can also be done in $O(n_v)$ time, and hence constructing these data structures on all nodes in R takes $O(n\log_\Delta n)=O(n\log n/\log\log n)$ time. Finally, constructing the hive graph can be done in $O(n\log n)$ time [11].

5 Lower Bound

We now prove the lower bound for the random access problem in Theorem 2. The *prefix selection problem* is to preprocess an array A of n unique integers to support prefix selection queries, that is, given integers i and j report the jth smallest integer in the subarray $A[1..i]$.

▶ **Lemma 12** (Jørgensen and Larsen [34]). *Any data structure that uses $n\log^{O(1)} n$ space on an input array of size n needs $\Omega(\log n/\log\log n)$ time to support prefix selection queries.*

Given an input array A to the prefix selection problem, we construct an instance T of the random access problem. Our reduction allows any prefix selection query on A to be answered by a single random access query on T. The reduction works even when T is a path without any deletions.

Let A be an array of length n consisting of unique integers in $\{1,\ldots,n\}$. Our instance T is a path of $n+1$ nodes $v_0,\ldots,v_n$ rooted at v_0. See Figure 4. Edge (v_{i-1},v_i) is labeled by $\mathsf{insert}(r_i,i)$, where r_i is the number of entries in $A[1..i]$ that are smaller $A[i]$. We have that $S(v_i)$ is permutation of indices in $\{1,..,i\}$ corresponding to the sorted order of $A[1..i]$, that is, $A[S(v_i)[1]]<\cdots<A[S(v_i)[i]]$. In particular, $S(v_i)[j]$ is the index of the jth smallest integer in $A[1..i]$. Hence, we can answer a prefix selection query $\mathsf{prefix\text{-}select}(i,j)$ by computing $\mathsf{access}(v_i,j)$. This completes the proof of Theorem 2.

6 Conclusion and Open Problems

We have initiated the study of persistent strings for storing and accessing compressed collections of similar strings. We have shown how to store a persistent string in linear space with optimal random access time. An interesting open problem is to make our solution *dynamic* by supporting insertion of new nodes in the version tree (representing new strings added to the collection). Another open problem is to improve our straightforward $O(n\log n/\log\log n)$ preprocessing time to optimal $O(n)$.

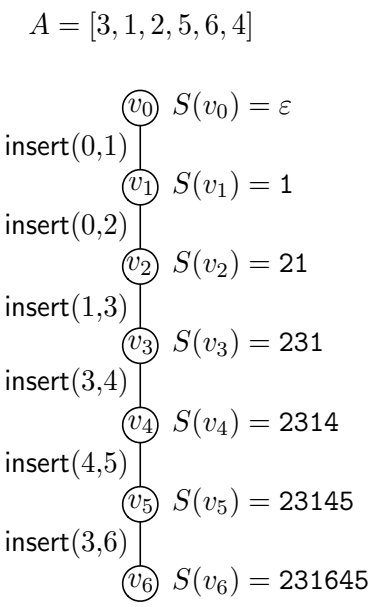

Figure 4 The corresponding random access instance for an array $A = [3, 1, 2, 5, 6, 4]$.

References

1 Jérémy Barbay, Francisco Claude, Travis Gagie, Gonzalo Navarro, and Yakov Nekrich. Efficient fully-compressed sequence representations. *Algorithmica*, 69(1):232–268, 2014.

2 Jérémy Barbay, Meng He, J Ian Munro, and S Srinivasa Rao. Succinct indexes for strings, binary relations and multi-labeled trees. In *Proc. 18th SODA*, pages 680–689, 2007.

3 Djamal Belazzougui, Patrick Hagge Cording, Simon J. Puglisi, and Yasuo Tabei. Access, rank, and select in grammar-compressed strings. In *Proc. 23rd ESA*, pages 142–154, 2015.

4 Djamal Belazzougui and Gonzalo Navarro. Optimal lower and upper bounds for representing sequences. *ACM Trans. Algorithms*, 11(4):1–21, 2015.

5 Philip Bille, Anders Roy Christiansen, Patrick Hagge Cording, Inge Li Gørtz, Frederik Rye Skjoldjensen, Hjalte Wedel Vildhøj, and Søren Vind. Dynamic relative compression, dynamic partial sums, and substring concatenation. *Algorithmica*, 80(11):3207–3224, 2018. Announced at ISAAC 2016.

6 Philip Bille, Anders Roy Christiansen, Nicola Prezza, and Frederik Rye Skjoldjensen. Succinct partial sums and Fenwick trees. In *Proc. 24th SPIRE*, pages 91–96, 2017.

7 Philip Bille, Mikko Berggren Ettienne, Inge Li Gørtz, and Hjalte Wedel Vildhøj. Time–space trade-offs for lempel–ziv compressed indexing. *Theoret. Comput. Sci.*, 713:66–77, 2018.

8 Philip Bille, Inge Li Gørtz, Gad M Landau, and Oren Weimann. Tree compression with top trees. *Inform. and Comput.*, 243:166–177, 2015.

9 Philip Bille, Gad M. Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. Random access to grammar-compressed strings and trees. *SIAM J. Comput.*, 44(3):513–539, 2015. Announced at SODA 2011.

10 M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Trans. Inform. Theory*, 51(7):2554–2576, 2005.

11 Bernard Chazelle. Filtering search: A new approach to query-answering. *SIAM J. Comput.*, 15(3):703–724, 1986.

12 BG Chern, Idoia Ochoa, Alexandros Manolakos, Albert No, Kartik Venkat, and Tsachy Weissman. Reference based genome compression. In *Proc. 12th ITW*, pages 427–431, 2012.

13 P. F. Dietz. Fully persistent arrays (extended array). In *Proc. 1st WADS*, volume 382, pages 67–74, 1989.

14 Paul F Dietz. Optimal algorithms for list indexing and subset rank. In *Proc. 1st WADS*, pages 39–46, 1989.

15 Huy Hoang Do, Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Fast relative Lempel–Ziv self-index for similar sequences. *Theoret. Comput. Sci.*, 532:14–30, 2014.

16 J. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan. Making data structures persistent. *J. Comput. System Sci.*, 38:86–124, 1989.

17 Peter M Fenwick. A new data structure for cumulative frequency tables. *Software: Pract. Exper.*, 24(3):327–336, 1994.

18 Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms*, 3(2):20, 2007.

19 Paolo Ferragina and Rossano Venturini. A simple storage scheme for strings achieving entropy bounds. *Theoret. Comput. Sci.*, 372(1):115–121, 2007.

20 Michael Fredman and Michael Saks. The cell probe complexity of dynamic data structures. In *Proc. 21st STOC*, pages 345–354, 1989.

21 Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. System Sci.*, 47(3):424–436, 1993.

22 Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. System Sci.*, 48(3):533–551, 1994.

23 Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J Puglisi. A faster grammar-based self-index. In *Proc. 6th LATA*, pages 240–251, 2012.

24 Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J Puglisi. LZ77-based self-indexing with faster pattern matching. In *Proc. 14th LATIN*, pages 731–742, 2014.

25 Travis Gagie, Paweł Gawrychowski, and Simon J Puglisi. Approximate pattern matching in lz77-compressed texts. *J. Discrete Algorithms*, 32:64–68, 2015.

26 Travis Gagie, Kalle Karhu, Gonzalo Navarro, Simon J Puglisi, and Jouni Sirén. Document listing on repetitive collections. In *Proc. 24th CPM*, pages 107–119, 2013.

27 Moses Ganardi, Artur Jez, and Markus Lohrey. Balancing straight-line programs. In *Proc. 60th FOCS*, pages 1169–1183, 2019.

28 Alexander Golynski, J Ian Munro, and S Srinivasa Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 17th SODA*, pages 368–373, 2006.

29 Alexander Golynski, Rajeev Raman, and S Srinivasa Rao. On the redundancy of succinct data structures. In *Proc. 11th SWAT*, pages 148–159, 2008.

30 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proc. 14th SODA*, pages 841–850, 2003.

31 Roberto Grossi, Rajeev Raman, Satti Srinivasa Rao, and Rossano Venturini. Dynamic compressed strings with random access. In *Proc. 40th ICALP*, pages 504–515. Springer, 2013.

32 Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Succinct data structures for searchable partial sums with optimal worst-case performance. *Theoret. Comput. Sci.*, 412(39):5176–5186, 2011.

33 Christopher Hoobin, Simon J Puglisi, and Justin Zobel. Relative Lempel-Ziv factorization for efficient storage and retrieval of web collections. *Proc. VLDB Endowment*, 5(3):265–273, 2011.

34 Allan Grønlund Jørgensen and Kasper Green Larsen. Range selection and median: Tight cell probe lower bounds and adaptive data structures. In *Proc. 22nd SODA*, pages 805–813, 2011.

35 Dominik Kempa and Nicola Prezza. At the roots of dictionary compression: String attractors. In *Proc. 50th STOC*, pages 827–840, 2018.

36 Shanika Kuruppu, Simon J Puglisi, and Justin Zobel. Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In *Proc. 17th SPIRE*, pages 201–206, 2010.

37 Shanika Kuruppu, Simon J Puglisi, and Justin Zobel. Optimized relative Lempel-Ziv compression of genomes. In *Proc. 34th ACSC*, pages 91–98, 2011.

38 Stan Y. Liao, Srinivas Devadas, and Kurt Keutzer. A text-compression-based method for code size minimization in embedded systems. *Trans. Design Autom. Electr. Syst.*, 4(1):12–38, 1999.

39 Stan Y. Liao, Srinivas Devadas, Kurt Keutzer, Steven W. K. Tjiang, and Albert Wang. Code optimization techniques in embedded DSP microprocessors. *Design Autom. Emb. Sys.*, 3(1):59–73, 1998.

40 Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *J. Comput. Biol.*, 17(3):281–308, 2010.

41 J Ian Munro and Yakov Nekrich. Compressed data structures for dynamic sequences. In *Proc. 23rd ESA*, pages 891–902. Springer, 2015.

42 Gonzalo Navarro. Indexing highly repetitive collections. In *Proc. 23rd IWOCA*, pages 274–279, 2012.

43 Gonzalo Navarro. Document listing on repetitive collections with guaranteed performance. *Theoret. Comput. Sci.*, 772:58–72, 2019.

44 Mihai Pătraşcu and Mikkel Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. In *Proc. 55th FOCS*, pages 166–175, 2014.

45 Mihai Pătraşcu and Erik D. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM J. Comput.*, 35(4):932–963, 2006. Announced at SODA 2004.

46 Rajeev Raman, Venkatesh Raman, and S Srinivasa Rao. Succinct dynamic data structures. In *Proc. 7th WADS*, pages 426–437, 2001.

47 Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoret. Comput. Sci.*, 302(1-3):211–222, 2003.

48 Kunihiko Sadakane and Roberto Grossi. Squeezing succinct data structures into entropy bounds. In *Proc. 17th SODA*, pages 1230–1239, 2006.

49 Neil Sarnak and Robert Endre Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, 1986.

50 James A. Storer and Thomas G. Szymanski. The macro model for data compression. In *Proc. 10th STOC*, pages 30–39, 1978.

51 James A Storer and Thomas G Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, 1982.

52 Robert Endre Tarjan and Uzi Vishkin. Finding biconnected componemts and computing tree functions in logarithmic parallel time. In *Proc. 25th FOCS*, pages 12–20, 1984.

53 Elad Verbin and Wei Yu. Data structure lower bounds on random access to grammar-compressed strings. In *Proc. 24th CPM*, pages 247–258, 2013.

54 Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inform. Theory*, 23(3):337–343, 1977.