A General Technique for Managing Strings in Comparison-Driven Data Structures

Gianni Franceschini and Roberto Grossi

Dipartimento di Informatica, Università di Pisa
Largo Pontecorvo 1, 56127 Pisa, Italy

Abstract. This paper presents a general technique for optimally transforming any dynamic data structure $\mathscr{D}$ that operates on atomic and indivisible keys by constant-time comparisons, into a data structure $\mathscr{D}'$ that handles unbounded-length keys whose comparison cost is not a constant.

1 Introduction

Many applications manage keys that are arbitrarily long strings, such as multi-dimensional points, multiple-precision numbers, multi-key data, URL addresses, IP addresses, XML path strings, and that are modeled either as k-dimensional keys for a given positive integer $k > 1$, or as variable-length keys. In response to the increasing variety of these applications, the keys need to be maintained in sophisticated data structures. The comparison of any two keys is more realistically modeled as taking time proportional to their length, producing an undesirable slowdown factor in the complexity of the operations supported by the data structures.

More efficient *ad hoc* data structures have been designed to tackle this drawback. A first version of lexicographic or ternary search trees [6] dates back to [7] and is alternative to tries. Each node contains the i-th symbol of a k-dimensional key along with three branching pointers (left, middle, and right) for the three possible comparison outcomes $[<, =, >]$ against that element. The dynamic balancing of ternary search trees was investigated with lexicographic D-trees [18], multidimensional B-trees [13], lexicographic globally biased trees [5], lexicographic splay trees [23], k-dimensional balanced binary search trees [11], and balanced binary search trees or kBB-trees [25]. Most of these data structures make use of sophisticated and tricky techniques to support search, insert, and delete of a key of length k in $O(k + \log n)$ time [5, 11]. Some others support also split and concatenate operations in $O(k + \log n)$ time [13, 18, 23, 25]. Moreover, other data structures allow for weighted keys (e.g., access frequencies) and the $\log n$ term in their time complexity is replaced by the logarithm of the ratio between the total weights and the weight of the key at hand [5, 18, 23, 25].

This multitude of *ad hoc* data structures stems from the lack of a general data structural transformation from indivisible (i.e., constant-time comparable) keys to strings. Many searching data structures, such as AVL-trees, red-black trees [24], (a, b)-trees [15], weight-balanced BB$[\alpha]$-trees [20], self-adjusting trees [23], and random search trees [22], etc., are currently available, with interesting combinatorial properties that make them attractive both from the theoretical and from the practical point of view. They are defined on a set of indivisible keys supporting an order relation. Searching and updating is driven by constant-time comparisons against the keys stored in them. One may wonder whether should data structuring designers reinvent the wheel in some cases or can they reuse the properties of these solutions.

A first step in reusing this body of knowledge and obtaining new data structures for managing strings has been presented theoretically in [12] and validated with experiments in [9]. It is based on the topology of the data structures by augmenting the nodes along the access paths to keys, each node with a pair of integers. By topology awareness, we mean that the designer must know the combinatorial properties and the invariants that are used to search and update the data structures, since he has to deal with all possible access paths to the same node. This depends on how the graph structure behind the data structure is maintained. While a general scheme is described for searching under this requirement, updating is discussed on an individual basis for the above reason. A random access path, for example, cannot be managed unless the possible access paths are limited in number. Also, adding an internal link may create many access paths to a given node. Related techniques, although not as general as that in [12], have been explored in [16, 21] for specific data structures being extended to manage strings.

In this paper, we go on one step ahead. We completely drop any topological knowledge of the underlying data structures and still obtain the asymptotic bounds of previous results. The goal is to show that a more general transformation is indeed possible. In particular, we present a general technique which is capable of reusing many kinds of (heterogeneous) data structures so that they can operate on strings. We just require that each such data structure, say $\mathscr{D}$, is driven by constant-time comparisons among the keys (i.e., no hashing or bit manipulation of the keys) and that the insertion of a key into $\mathscr{D}$ identifies the predecessor or the successor of that key in $\mathscr{D}$. We are then able to transform $\mathscr{D}$ into a new data structure, $\mathscr{D}'$, storing n strings as keys while preserving all the nice features of $\mathscr{D}$. Asymptotically speaking, this transformation is costless. First, the space complexity of $\mathscr{D}'$ is $\mathscr{S}(n)+O(n)$, where $\mathscr{S}(n)$ denotes the space complexity of $\mathscr{D}$ (just store the pointers to strings, not the strings themselves). Second, each operation involving $O(1)$ strings taken from $\mathscr{D}'$ requires $O\big(\mathscr{T}(n)\big)$ time, where $\mathscr{T}(n)$ denotes the time complexity of the corresponding operation originally supported in $\mathscr{D}$. Third, each operation involving a string y *not* stored in $\mathscr{D}'$ takes $O\big(\mathscr{T}(n) + |y|\big)$ time, where $|y|$ denotes the length of y.

The field of interest for our technique is especially for sub-logarithmic costs, when $\mathscr{T}(n) = o(\log n)$: either in the worst case (e.g., $\mathscr{D}$ is a finger search tree), in amortized sense (e.g., $\mathscr{D}$ is a self-adjusting tree) or with high probability (e.g., $\mathscr{D}$ is a treap), when considering access frequencies in the analysis.

Our technique exploits the many properties of one-dimensional searching, and combines in a variety of novel ways techniques from data structures and string algorithms. Formally, we manage input strings $x_1, x_2, \ldots, x_n$ of total length $M = \sum_{i=1}^{n} |x_i|$. Each string x_i is a sequence of $|x_i|$ symbols drawn from a potentially unbounded alphabet Σ, and the last symbol of x_i is a special endmarker less than any symbol in Σ. In order to compare two strings x and y, it is useful to employ the length of their longest common prefix, defined as $lcp(x, y) = \max\{\ell \geq 0 \mid x[1\,..\,\ell] = y[1\,..\,\ell]\}$ (here, $\ell = 0$ denotes empty prefixes). Given that length, we can compare x and y in constant time by simply comparing their first mismatching symbol, which is at position $1 + lcp(x, y)$ in x and y.

With this fact in mind, we can use the underlying data structure $\mathscr{D}$ as a black box. We use simple properties of strings and introduce a powerful oracle for string comparisons that extends the functionalities of the Dietz-Sleator list [10], which is able to maintain order information in a dynamic list (shortly, DS list). We call the resulting structure a DS_{lcp} list, which stores the sorted input strings in $O(n)$ space, and allows us to find the length of the longest common prefix of any two strings stored in the DS_{lcp} list, in constant time. [1] We can maintain dynamically a DS_{lcp} list in constant time per operation (see Section 2.1 for the operations thus supported) by using a simple but key idea in a restricted dynamic version of the range minima query problem [4]. Note that otherwise would not be possible to achieve constant time per operation in the fully dynamic version of this problem as we can perform sorting with it.

Using our general technique, we obtain previous theoretical bounds in an even simpler way. We also obtain new results on searching and sorting strings. For example, we can perform suffix sorting, a crucial step in text indexing and in block sorting compression based on the Burrows-Wheeler transform, in $O\big(n + \sum_{i=1}^{n} \mathscr{T}(i)\big)$ time, also for unbounded alphabet Σ. For this alphabet, this appears to be a new result; the known literature reports the time complexity of $\Theta(n \log n)$ in the worst case as it tantamounts to sorting the alphabet symbols (a linear time bound is possible in some special cases). Using our result, we can perform suffix sorting in $O\big(n \log(F/n)\big)$ time, where $0 \leq F \leq n(n-1)/2$ is the number of inversions. This new result is a simple consequence of our result, when applied to the techniques for one-dimensional keys given, for example, in [19]. Another example of use is that of storing implicitly the root-to-nodes paths in a tree as strings, so that we can support dynamic lowest common ancestor (lca) queries in constant time, where the update operations involve adding/removing leaves. In previous work, this result has been obtained with a special data structure based upon a more sophisticated solution treating also insertions that split arcs [8]. We obtain a simple method for a restricted version of the problem.

As a final remark for our technique, we do not claim that it is as amenable to implementation in a practical setting such as the technique in [9, 12]. Nevertheless, we believe that our general technique may be helpful in the theoretical setting for providing an immediate benchmark to the data structuring designer.

[1] When $\mathscr{T}(n) = \Omega(\log n)$, there are alternative techniques, e.g., using compacted tries and dynamic lowest common ancestor queries [8], as $O\big(\mathscr{T}(n) + |y|\big)$ absorbs the cost of inserting a string y into the trie, which is either $O(|y| \log |\Sigma|)$ or $O(|y| + \log n)$ in the worst case, as is the case for unbounded Σ in the comparison model.

```
1: $m \leftarrow DS_{lcp}(best.friend, x)$
2: IF $m \geq best.lcp$ THEN
3:     $m \leftarrow best.lcp$
4:     WHILE $x[m+1] = y[m+1]$ DO $m \leftarrow m+1$
5:     $best.friend \leftarrow x$
6:     $best.lcp \leftarrow m$
7: RETURN $m$
```

Fig. 1. Code for computing $lcp(x, y)$ values on the fly.

When inventing a new data structure for strings, the designer can easily realize whether it compares favorably to the known data structures, whose functionalities can be smoothly extended as a black box to strings without giving up their structural and topological properties.

2 The General Technique for Strings

We now describe our technique. The operations supported by the DS_{lcp} list are listed in Section 2.1, whose implementation is discussed later on, in Section 4. The fast computation on the fly of *lcp* values is presented in Section 2.2. The use of the latter two tools in our technique is shown in Section 2.3. We recall that, for any two strings x and y, we have $x \leq y$ in lexicographic order if and only if $x[\ell+1] < y[\ell+1]$, where $\ell = lcp(x, y)$. Here is why we center our discussion around the efficient computation of *lcp* values.

2.1 The DS_{lcp} list

The DS_{lcp} list stores a sequence of strings $x_1, x_2, \ldots, x_n$ in lexicographic order, each string is of unbounded-length and is referenced by a constant-space pointer (e.g., `char *p` in C language). A DS_{lcp} list L supports the following operations:

- Query $DS_{lcp}(x_p, x_q)$ in L, returning the value of $lcp(x_p, x_q)$, for any pair of strings x_i and x_j stored in L.
- Insert y in a position between two consecutive keys x_{k-1} and x_k in L. Requirements: $x_{k-1} \leq y \leq x_k$ holds, and $lcp(x_{k-1}, y)$ and $lcp(y, x_k)$ are known.
- Remove string x_i from its position in L.

Theorem 1. *A DS_{lcp} list L can be implemented in $O(n)$ space, so that querying for lcp values, inserting keys into L and deleting keys from L can be supported in constant time per operation, in the worst case.*

2.2 Computing *lcp* values on the fly

The DS_{lcp} list L is a valid tool to dynamically compute *lcp* values for the strings stored in L. We now examine the situation in which we have to compare a string

$y \notin L$ against an arbitrary choice of strings $x \in L$. We put ourselves in the worst situation, namely, the choice of x is unpredictable from our point of view. Even in this case, we can still compute $lcp(x, y)$ efficiently. We implicitly assume that the empty string is kept in L as the least string.

We employ two global variables, *best.friend* and *best.lcp*, which are initialized to the empty string and to 0, respectively. During the computation, they satisfy the invariant that, among all the strings in L compared so far against y, the one pointed by *best.friend* gives the maximum *lcp* value, and that value is *best.lcp*. We now have to compare y against x, following the simple algorithm shown in Fig. 1. Using L, we can compute $m = DS_{lcp}(best.friend, x)$, since both strings are in L. If $m < best.lcp$, we can infer that $lcp(x, y) = m$ and return that value. Otherwise, we may possibly extend the number of matched characters in y storing it into *best.lcp*, thus finding a new *best.friend*. It's a straightforward task to prove the correctness of the invariant (note that it works also when $x = best.friend$). Although the code can be improved by splitting the case $m \geq best.lcp$ of line 2 into two cases, it does not improve the asymptotic complexity.

Let's take an arbitrary operation that accesses some of the strings in L in arbitrary order, say, $x'_1, x'_2, \ldots, x'_g$ (these strings are not necessarily distinct and/or sorted). For a given string $y \notin L$, the total cost of computing $lcp(x'_1, y)$, $lcp(x'_2, y)$, $\ldots$, $lcp(x'_g, y)$ on the fly with the function shown in Fig. 1 can be accounted as follows. The cost of the function is constant unless we enter the body of the while loop at line 4, to match further characters while increasing the value of m. We can therefore restrict our analysis to the strings $x'_i \in L$ that cause the execution of the body of that while loop. Let's take the kth such string, and let m_k be the value of m at line 6. Note that the body of the while loop at line 4 is executed $m_k - m_{k-1}$ times (precisely, this is true since $best.lcp = m_{k-1}$, where $m_0 = 0$). Thus the cost of computing the *lcp* value for such a string is $O(1 + m_k - m_{k-1})$.

We can sum up all the costs. The strings not entering the while loop contribute each for a constant number of steps; the others contribute for $O(1+m_k - m_{k-1})$ steps. As a result, we obtain a total cost of $O(g + \sum_k (m_k - m_{k-1})) = O(g + |y|)$ time, since $m_k \geq m_{k-1}$ and $\sum_k (m_k - m_{k-1})$ is upper bounded by the length of the longest matched prefix of y, which is in turn at most $|y|$.

Lemma 1. *The computation on the fly of any sequence of g lcp values involving a given string $y \notin L$ and some strings in L can be done in $O(g + |y|)$ time.*

Note that, if y were in L, we could obtain a bound of $O(g)$ in Lemma 1. Instead, $y \notin L$, and L helps us to reduce the cost from $O(g \cdot |y|)$ to $O(g + |y|)$.

2.3 Exploiting *lcp* values in comparison-driven data structures

We now finalize the description of our general technique, leaving that of the implementation of the DS_{lcp} list L to Section 4.

Theorem 2. *Let $\mathscr{D}$ be a comparison-driven data structure such that the insertion of a key into $\mathscr{D}$ identifies the predecessor or the successor of that key in $\mathscr{D}$. Then, $\mathscr{D}$ can be transformed into a data structure $\mathscr{D}'$ for strings such that*

- *the space complexity of $\mathscr{D}'$ is $\mathscr{S}(n) + O(n)$ for storing n strings as keys (just store the references to strings, not the strings themselves), where $\mathscr{S}(n)$ denotes the space complexity of $\mathscr{D}$;*
- *each operation involving $O(1)$ strings in $\mathscr{D}'$ takes $O\big(\mathscr{T}(n)\big)$ time, where $\mathscr{T}(n)$ denotes the time complexity of the corresponding operation originally supported in $\mathscr{D}$;*
- *each operation involving a string y* not *stored in $\mathscr{D}'$ takes $O\big(\mathscr{T}(n) + |y|\big)$ time, where $|y|$ denotes the length of y.*

Proof. The new data structure $\mathscr{D}'$ is made up of the original data structure $\mathscr{D}$ along with the DS_{lcp} list L of Section 2.1, and uses the computation on the fly described in Section 2.2. The additional space is that of L, namely, $O(n)$ by Theorem 1.

For the cost of the operations, consider first the insertion of a key y into $\mathscr{D}'$. We run the insertion algorithm supported by $\mathscr{D}$ as a black box. When this algorithm requires to compare y with a string x already in $\mathscr{D}'$, we proceed as in Section 2.2. By hypothesis, the algorithm determines also the predecessor or the successor of y. In $O(|y|)$ time, we can compute (if not yet determined) their lcp values, which are needed to insert y into L. The final cost is that of Lemma 1, where $g \leq \mathscr{T}(n)$. The other operations supported by $\mathscr{D}'$ have a similar analysis. If they require comparisons that involve strings already stored in $\mathscr{D}'$, each comparison can be clearly performed in constant time by Theorem 1. Hence their cost is just $O\big(\mathscr{T}(n)\big)$ since $g \leq \mathscr{T}(n)$.

3 Some Applications

3.1 Suffix sorting

As previously mentioned, suffix sorting is very useful in compressing, with block sorting methods and Burrows-Wheeler transform, and in text indexing, with suffix arrays [17]. The problem is that of sorting lexicographically the suffixes of an input string s of length n. Let s_1, s_2, …, s_n denote the suffixes of s, where $s_i \equiv s[i\,..\,n]$ corresponds to the ith suffix in s. We show how to apply the ideas behind Theorem 2 to the suffix sorting problem. We recall that comparing two suffixes s_i and s_j takes constant time if we known their value of $lcp(s_i, s_j)$. Again, we focus our presentation on the lcp computation.

We first need to augment the DS_{lcp} list L of Theorem 2 with suffix links. Let's take a snapshot of L at the end of the suffix sorting. A suffix link $sl(s_r)$ points to s_{r+1} in L, for $1 \leq r < n$. During the intermediate steps, we insert the suffixes s_1, s_2, …, s_n into $\mathscr{D}'$, in this order.

Before inserting s_i into $\mathscr{D}'$, the pointers $sl(s_j)$ are defined for $1 \leq j < i - 1$. The current entry in L is s_{i-1}, for which we know its predecessor, x_{i-1}, and its successor, y_{i-1}, in L. Note that we cannot exploit $sl(s_{i-1})$ as s_i has still to be inserted. We also know $lcp(x_{i-1}, s_{i-1})$ and $lcp(s_{i-1}, y_{i-1})$. This invariant is trivially satisfied before inserting the first suffix, s_1 (here, $x_0 = s_0 = y_0 =$ empty string).

We use induction to describe the step for s_i. It suffices to show how to compute $lcp(z, s_i)$ on the fly, for $z \in \{s_j \mid j < i\}$, with reference to the code shown in Fig. 1. Assuming that $lcp(x_{i-1}, s_{i-1}) \geq lcp(s_{i-1}, y_{i-1})$ without loss of generality, we set $best.friend = sl(x_{i-1})$ and $best.lcp = \max\{0, best.lcp - 1\}$, before executing the sequence of calls to the function in Fig. 1 related to the insertion of s_i. When the insertion completes its task, we know the predecessor, x_i, and the successor, y_i, of s_i in L. We also know their *lcp* values. To maintain the invariant for the next step, we need to pose $sl(s_{i-1}) = s_i$.

Theorem 3. *Let $\mathscr{D}'$ be a data structure for managing strings obtained following Theorem 2. Then, all the suffixes of an input string of length n can be inserted into $\mathscr{D}'$, in space $O(n)$ and time*

$$O\left(n + \sum_{i=1}^{n} \mathscr{T}(i)\right),$$

where $\mathscr{T}(\cdot)$ denotes the time complexity of the insert operation in the original data structure $\mathscr{D}$ from which $\mathscr{D}'$ has been obtained. The suffixes can be retrieved in lexicographic order in linear time.

Theorem 3 provides an adaptive bound for input strings whose symbols have some presortedness. There are cases in which $\sum_{i=1}^{n} \mathscr{T}(i) = o(n \log n)$ for arbitrary alphabets Σ, whereas the known literature for suffix sorting reports the time complexity of $\Theta(n \log n)$ in the worst case as we are essentially sorting the alphabet symbols (a linear time bound is possible in special cases). One extreme example is an input string with all distinct characters in increasing order, for which the bound of Theorem 3 is $O(n)$. In general, we can perform suffix sorting in $O\big(n \log(F/n)\big)$ time, where $0 \leq F \leq n(n-1)/2$ is the number of inversions. We obtain a new result that reuses techniques for one-dimensional keys given, for example, in [19].

3.2 Dynamic lowest common ancestor (*lca*)

The lowest common ancestor problem for a tree is at the heart of several algorithms [4, 14]. We consider here the dynamic version in which insertions add new leaves as children to existing nodes and deletions remove leaves. The more general (and complicated) case of splitting an arc by inserting a node in the middle of the arc is treated in [8].

We maintain the tree as an Euler tour, which induces an implicit lexicographic order on the nodes. Namely, if a node is the ith child of its parent, the implicit label of the node is i. The root has label 0. (These labels are mentioned only for the purpose of presentation.) The implicit string associated with a node is the sequence of implicit labels obtained in the path from the root to that node plus an endmarker that is different for each string (also when the string is duplicated; see the discussion below on insertion). Given any two nodes, the *lcp* value of their implicit strings gives the string implicitly represented by their *lca*. We maintain the Euler tour with a DS_{lcp} list L in $O(n)$ space (see Section 2.1), where n is the number of nodes (the strings are implicit and thus do not need to be stored). We also maintain the dynamic data structure in [1] to find the level ancestor of a node in constant time.

Given any two nodes u and v, we compute $lca(u,v)$ in constant time as follows. We first find $d = lcp(s_u, s_v)$ using L, where s_u and s_v are the implicit strings associated with u and v, respectively. We then identify their ancestor at depth d using a level ancestor query.

Inserting a new leaf duplicates the implicit string s of the leaf's parent, and puts the implicit string of the leaf between the two copies of s thus produced in the Euler tour. Note that we satisfy the requirements described in Section 2.1 for the insert, as we know their lcp values. By Theorem 1, this takes $O(1)$ time. For a richer repertoire of supported operations in constant time, we refer to [8].

Theorem 4. *The dynamic lowest common ancestor problem for a tree, in which leaves are inserted or removed, can be solved in $O(1)$ time per operation in the worst case, using a DS_{lcp} list and the constant-time dynamic level ancestor.*

4 Implementation of the DS_{lcp} List

We describe how to prove Theorem 1, implementing the DS_{lcp} list L introduced in Section 2.1. We use the fact that $lcp(x_p, x_q) = \min\{lcp(x_{k-1}, x_k) \mid p < k \leq q\}$ since the strings $x_1, x_2, \ldots, x_n$ in L are in lexicographic order (here, $p < q$). In other words, storing only the lcp values between each key x_k and its predecessor in L, we can answer arbitrary lcp queries using the so-called range minima query problem [4]. The input is an unordered set of m entries (the $lcp(x_{k-1}, x_k)$s) and, for any given range $[i \mathinner{..} j]$, we want to report the minimum among the entries from i to j (where $i = p + 1$ and $j = q$ for $lcp(x_p, x_q)$).

We are interested in discussing the dynamic version of the problem. In its general form, this is equivalent to sorting. Fortunately, we can attain constant time per operation since we impose the additional constraint that the set of entries can only vary *monotonically*. Namely, an entry e changes by replacing it with two entries e_1 and e_2, such that $e_1 \geq e_2 = e$ or $e_2 \geq e_1 = e$. This constraint is not artificial, being dictated by the requirements listed in Section 2.1 when inserting string y between x_{k-1} and x_k. A moment of reflection shows that both $e_1 \equiv lcp(x_{k-1}, y)$ and $e_2 \equiv lcp(y, x_k)$ are greater than or equal to $e \equiv lcp(x_{k-1}, x_k)$ (the entry to be changed), and at least one of them equals e.

Going straight to the benefit of monotonicity in a dynamic setting, consider the problem of maintaining the prefix minima $p_1, p_2, \ldots, p_m$ for the m entries (treating suffix minima is analogous). When inserting y as above, *just two* prefix minima can change, whereas they can all change without the monotonicity. Namely, $p_y = \min\{p_{k-1}, e_1\}$ and $p_k = \min\{p_y, e\}$, assuming $e_1 \geq e_2 = e$ and letting p_y be the prefix minimum for the entry e_1 associated with y. We use this as a key observation to obtain constant-time complexity.

We focus on insertions as deletions are weak and can be treated with partial rebuilding techniques (deletions replace two consecutive entries with the smallest of the two and so do not change the range minima of the remaining entries). Note that the insertion of y can be viewed as either the insertion of entry e_1 to the left of entry e (when $e_1 \geq e_2 = e$) or the insertion of e_2 to the right of e (when $e_2 \geq e_1 = e$).

For implementing the DS_{lcp} list we adopt a two-level scheme. We introduce the upper level consisting of the *main tree* in Section 4.1, and the lower level populated of *micro trees* in Section 4.2. We sketch the method for combining the two levels in Section 4.3. The net result is a generalization of the structure of Dietz and Sleator that works for multidimensional keys, without relying on the well-known algorithm of Willard [26] to maintain order in a dense file (avoiding Willard's for the amortized case has been suggested in [3]). When treating information that can be represented with $O(\log n)$ bits, we will make use of table lookups in $O(1)$ time. The reader may verify that we also use basic ideas from previous work [2, 4, 14].

4.1 Main tree

For the basic shape of the main tree we follow the approach of [2]. The main tree has m leaves, all on the same level (identified as level 0), each leaf containing one entry of our range minima problem. The *weight* $w(v)$ a node v is *(i)* the number of its children (leaves), if v is on level 1 *(ii)* the sum of the weights of its children, if v is on a level $l > 1$. Let $b > 4$ the *branching parameter*. We maintain the following constraints on the weight of a node v on a level l.

1. If $l = 1$, $b \leq w(v) \leq 2b - 1$.
2. If $l > 1$, $w(v) < 2b^l$.
3. If $l > 1$ and v is not the root of the tree, $w(v) > \frac{1}{2}b^l$.

From the above constraints it follows that each node on a level $l > 1$ in the main tree has between $b/4$ and $4b$ children (with the exception of the root that can have a minimum of two children). From this we can easily conclude that the height of the main tree is $h = O(\log_b m) = 0(\log m)$.

When a new entry is inserted as a new leaf v in the main tree, any ancestor u of v that does not respect the weight constraint is split into two new nodes and the new child is inserted in the parent of u (unless u is the root, in that case a new root is created). That rebalancing method has an important property.

Lemma 2 ([2]). *After a split of a node u on level $l > 1$ into nodes u' and u'', at least $b^l/2$ inserts have to be performed below u' (or u'') before splitting again.*

The nodes of the main tree are augmented with two secondary structures.

The former secondary structure is devoted to *lca* queries. Each internal node u is associated with a numeric identifier $sib(u)$ representing its position among its siblings; since the maximum number of children of a node is a constant, we need only a constant number of bits, say c_b, to store each identifier. Each leaf v is associated with two vectors, $\mathscr{P}_v$ and $\mathscr{A}_v$. Let u_i be the i-th ancestor of v (starting from the root). The i-th location of $\mathscr{P}_v$ contains the identifier

$sib(u_i)$ whereas the i-th location of $\mathscr{A}_v$ contains a pointer to u_i. Note that $\mathscr{P}_v$ occupies $h \cdot c_b = O(\log m)$ bits. These auxiliary vectors are used to find the *lca* between any two leaves v', v'' of the main tree in constant time. First, we find $j = lcp(\mathscr{P}_{v'}, \mathscr{P}_{v''})$ by table lookups; then, we use the pointer in $\mathscr{A}_v[j]$ to access the node.

The latter secondary structure is devoted to maintain some range minima. Each internal node u is associated with a doubly linked list $\mathscr{E}_u$ that contains the copies of the entries in the descendant leaves of u. The order of the copies in $\mathscr{E}_u$ is identical to that in the leaves (i.e., the lexicographical order in which the strings are maintained). As previously mentioned, we maintain the prefix minima and the suffix minima in $\mathscr{E}_u$. Then, we associate with each leaf v a vector $\mathscr{C}_v$ containing pointers to all the copies of the entry in v, each copy stored in the doubly linked lists of v's ancestors.

Because of the redundancy of information, the total space occupied by the main tree is $O(m \log m)$ but now we are able to answer a general range minima query for an interval $[i\,..\,j]$ in constant time. We first find the lowest common ancestor u of the leaves v_i and v_j corresponding to the ith and the jth entries, respectively. Let u_i be the child of u leading to v_i and u_j that leading to v_j (they must exist and we can use $\mathscr{P}_{v_i}$ and $\mathscr{P}_{v_j}$ for this task). We access the copies of the entries of i and j in $\mathscr{E}_{u_i}$ and $\mathscr{E}_{u_j}$, respectively, using $\mathscr{C}_{v_i}$ and $\mathscr{C}_{v_j}$. We then take the suffix minimum anchored in i for $\mathscr{E}_{u_i}$, and the prefix minimum anchored in j for $\mathscr{E}_{u_j}$. We also take the minima in the siblings between u_i and u_j (excluded). The minimum among these $O(1)$ minima is then the answer to our query for interval $[i\,..\,j]$.

Lemma 3. *The main tree for m entries occupies $O(m \log m)$ space, and support range minima queries in $O(1)$ time and monotone updates in $O(\log m)$ time.*

It remains to see how the tree can be updated. We are going to give a "big picture" of the techniques used, leaving the details of the most complicated aspects to the full version of the paper. We already said that deletions can be treated lazily with usual partial rebuilding techniques. We follow the same approach for treating the growth of the height h of the main tree and the subsequent variations of its two secondary structures, $\mathscr{P}$, $\mathscr{A}$, $\mathscr{E}$, and $\mathscr{C}$. From now on, let's assume w.l.o.g. that the insertions do not increase the height of the main tree.

When a new string is inserted, we know by hypothesis a pointer to its predecessor (or successor) in the lexicographical order and the pointer to the leaf v of the main tree that receives a new sibling v' and contains the entry (*lcp* value) to be changed. The creation and initialization of the vectors associated with the new leaf v' can be obviously done in $O(\log m)$ time. Then we must propagate the insertion of the new entry in v' to its ancestors. Let u be one of these ancestors. We insert the entry into its position in $\mathscr{E}_u$, using $\mathscr{C}_v$, which is correctly set (and useful for setting $\mathscr{C}_{v'}$). As emphasized at the beginning of Section 4, the monotonicity guarantees that the only prefix minima changing are constant in number and near to the new entry (an analogous situation holds for the suffix minima). As long as we do not need to split an ancestor, we can therefore perform this update in constant time per ancestor.

If an ancestor u at level l needs to split in two new nodesu' and u'' so as to maintain the invariants on the weights, we need to recalculate $O(|\mathscr{E}_u|) = O(b^l)$ values of prefix and suffix minima. By Lemma 2 we can immediately conclude that the time needed to split u is $O(1)$ in amortized sense. A lazy approach to the construction of the lists $\mathscr{E}_{u'}$ and $\mathscr{E}_{u''}$ will lead to the desired worst case constant time complexity for the splitting of an internal node. This construction is fairly technical (e.g., see [2]) and we will detail it in the full paper.

4.2 Micro trees for indirection

We employ micro trees for providing a level of indirection to reduce space and update time in the main tree of Section 4.1. Each micro tree satisfies the invariants 1–3 of the main tree, except that all the nodes contain $O(\log n)$ entries, with a fan out of $O(\log n)$. We guarantee that a micro tree stores $\Theta(\log^2 n)$ entries in two levels (its height is $h = 2$). We fill its nodes by starting from a doubly linked list of its $\Theta(\log^2 n)$ entries. We partition it into sublists of size $O(\log n)$, which are the leaves. Then, we take the first and the last entry in each sublist, and copy these two entries in a new sublist of size $O(\log n)$, so as to form the root. It's not difficult to maintain the weighted median entry of each sublist in $O(1)$ time per operation. This is useful to split a sublist (node) into two sublists (nodes) of equal size in $O(1)$ time (we can incrementally scan them by $O(1)$ entries at a time and find the new weighted median before they split again).

We have a secondary structure in the nodes of the micro trees, to locally support range minima, split and insert in $O(1)$ time. Each sublist is associated with a Cartesian tree [4, 14]. The root is the minimum entry and its left (right) subtree recursively represents the entries to the left (right). The base case corresponds to the empty set which is represented by the null pointer. The observation in [4] is that the range minimum from entry i to entry j is given by the entry represented by $lca(i, j)$ in the Cartesian tree. We encode all this information in $O(\log n)$ bits so that it can be manipulated in $O(1)$ time with table lookups. Note that inserting entries monotonically guarantees that these entries are inserted as leaves, so we know the position of insertion in constant time. Again, we will detail it in the full paper.

4.3 Implementing the operations

Our high-level scheme is similar to that in [10]. The main tree has $m = O(n/\log^2 n)$ leaves. Each leaf is associated with a distinct micro tree, so that the concatenation of the micro trees gives the order kept in the DS_{lcp} list. Each micro tree contributes to main tree with its leftmost and rightmost entries (actually, the range minima of its two extremal entries). A micro tree is ready to split, when the number of its keys is at least 2/3 of the maximum allowed. The ready micro trees are kept in a queue sorted by size. We take the largest such tree, split it in $O(1)$ time and insert two new entries in the main tree. However, we perform incrementally and lazily the $O(\log n)$ steps for the insertion (Lemma 3) of these two entries. At any time only one update is pending in the main tree by an argument similar to that in [10].

References

1. S. Alstrup, J. Holm. Improved algorithms for finding level ancestors in dynamic trees. *ICALP*, 73–84, 2000.
2. L. Arge, J. S. Vitter. Optimal external memory interval management. *SIAM Journal on Computing*, 32:1488–1508, 2003.
3. M. A. Bender, R. Cole, E. M. Demaine, M. Farach-Colton, J. Zito. Two simplified algorithms for maintaining order in a list. In *ESA*, 2002.
4. M. A. Bender and M. Farach-Colton. The LCA problem revisited. *LATIN*, 88–94, 2000.
5. S.W. Bent, D.D. Sleator and R.E. Tarjan. Biased search trees. *SIAM Journal on Computing* 14 (1985), 545–568.
6. J.L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *SODA* (1997), pages 360–369.
7. H.A. Clampett. Randomized binary searching with the tree structures. *Communications of the ACM* 7 (1964), 163–165.
8. R. Cole and R. Hariharan. Dynamic LCA queries on trees. *SODA*, 235–244, 1999.
9. P. Crescenzi, R. Grossi, and G.F. Italiano. Search data structures for skewed strings. *WEA*, 2003.
10. P.F. Dietz and D.D. Sleator. Two algorithms for maintaining order in a list. *STOC*, 365–372, 1987.
11. T.F. Gonzalez. The on-line d-dimensional dictionary problem. *SODA*, 376–385, 1992.
12. R Grossi and G. F. Italiano. Efficient techniques for maintaining multidimensional keys in linked data structures (extended abstract). *ICALP*, 372–383, 1999.
13. R.H. Gueting and H.-P. Kriegel. Multidimensional B-tree: An efficient dynamic file structure for exact match queries. *10th GI Annual Conference*, 375–388, 1980.
14. D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal of Computing*, 13:338–355, 1984.
15. S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica* 17 (1982), 157–184.
16. R.W. Irving and L. Love. The suffix binary search tree and suffix AVL tree. In *Journal of Discrete Algorithms*, 387–408, 2003.
17. U. Manber and E.W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing* 22 (1993), 935–948.
18. K. Mehlhorn. Dynamic binary search. *SIAM J. on Computing* 8 (1979), 175–198.
19. K. Mehlhorn. *Data structures and algorithms: 1. Searching and sorting*, 1984.
20. J. Nievergelt and E.M. Reingold. Binary search trees of bounded balance. *SIAM Journal on Computing* 2 (1973), 33–43.
21. S. Roura. Digital access to comparison-based tree data structures and algorithms. *Journal of Algorithms*, 40:1–23, 2001.
22. R. Seidel and C.R. Aragon. Randomized search trees. *Algorithmica*, 1996, 464–497.
23. D. D. Sleator and R.E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM* 32 (1985), 652–686.
24. R.E. Tarjan. *Data structures and network algorithms*, SIAM (1983).
25. V. K. Vaishnavi. On k-dimensional balanced binary trees. *JCSS* 52 (1996), 328–348.
26. D.E. Willard. A density control algorithm for doing insertions and deletions in a sequentially ordered file in good worst-case time. *Informat. and Comput.*, 1992.