A Simple Mergeable Dictionary*

Adam Karczmarz

Insitute of Informatics, University of Warsaw, Poland
a.karczmarz@mimuw.edu.pl

Abstract

A mergeable dictionary is a data structure storing a dynamic subset S of a totally ordered set $\mathcal{U}$ and supporting predecessor searches in S. Apart from insertions and deletions to S, we can both merge two arbitrarily interleaved dictionaries and split a given dictionary around some pivot $x \in \mathcal{U}$. We present an implementation of a mergeable dictionary matching the optimal amortized logarithmic bounds of Iacono and Özkan [11]. However, our solution is significantly simpler. The proposed data structure can also be generalized to the case when the universe $\mathcal{U}$ is dynamic or infinite, thus addressing one issue of [11].

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, E.1 Data Structures

Keywords and phrases dictionary, mergeable, data structure, merge, split

Digital Object Identifier 10.4230/LIPIcs.SWAT.2016.7

1 Introduction

Let $\mathcal{U}$ be some totally ordered set. An *ordered dictionary* is a data structure maintaining a set $S \subseteq \mathcal{U}$ and supporting the following operations:

- $S \leftarrow$ Make-Set(): create an empty set S.
- Insert(S, x): add an element $x \in \mathcal{U}$ to the set S.
- Delete(S, x): remove an element $x \in \mathcal{U}$ from the set S.
- $y \leftarrow$ Search(S, x): find the largest $y \in S$ such that $y \leq x$ (if such y exists).

Typically, such dictionaries also allow traversing the stored sets in order in linear time.

We call a data structure a *mergeable dictionary* if it supports two additional operations:

- $C \leftarrow$ Merge(A, B): create a set $C = A \cup B$. The sets A and B are destroyed.
- $(A, B) \leftarrow$ Split(C, x): create two sets $A = \{y \in C : y \leq x\}$ and $B = C \setminus A$, where $x \in \mathcal{U}$. The set C is destroyed.

Note that the operation Merge does not pose any conditions on its arguments. It should not be confused with the commonly used operation Join(A, B) which merges its arguments under the assumption that all the elements of A are no larger than the smallest element of B.

The ordered dictionary problem is well understood and various optimal solutions have been developed, including balanced binary search trees such as AVL trees or red-black trees [9], and skip-lists [14]. Each of these data structures performs the operations Insert, Delete, Search on a set S in $O(\log |S|)$ worst-case time. Most ordered dictionaries can be easily extended to support the operations Join and Split within the same time bounds. Clearly, it is not possible to achieve $o(|A| + |B|)$ worst-case bound for the Merge(A, B) operation, as that would lead to a $o(n \log n)$ comparison-based sorting algorithm. Nevertheless, it is interesting to study the amortized upper bounds of the mergeable dictionary operations.

* Supported by the grant NCN2014/13/B/ST6/01811 of the Polish Science Center.

Iacono and Özkan [11] developed the only data structure to date which provably supports *all* the mergeable dictionary operations in amortized logarithmic time. It is worth noting that their definition of a mergeable dictionary is a bit different: they define it to be a data structure maintaining a partition of a finite universe $\mathcal{U}$ of size n. The set of operations they support is Merge, Split, Search and Find, where Find(x) returns the unique element of the partition containing $x \in \mathcal{U}$. We find it more appropriate to call a data structure supporting such an interface *an ordered union-split-find* data structure instead. This small difference in the definition does not influence the core of the problem. The amortized lower bound of $\Omega(\log n)$ for at least one of the operations Merge, Split and Find is an easy consequence of the lower bounds for *partial sums* and *dynamic connectivity* [15].[1]

However, the data structure of Iacono and Özkan has two drawbacks. First, both the methods they used and the analysis are quite involved. Specifically, in order to achieve the goal, they used a highly non-trivial potential function for analysis, extended the *biased skip list* of [2] to support various finger-search-related operations. They also developed an element weighting scheme that allows Merge to be performed in time proportional to the decrease of the potential. Second, their weighting scheme depends heavily on the differences of ranks [2] of individual elements in $\mathcal{U}$ and as a result it is not clear how to generalize the data structure to work with potentially infinite universes, such as $\mathbb{R}$, in an online fashion. The subtlety of handling such universes lies in the fact that one can always insert a new element between consecutive elements of a stored set.

In this paper we show a very simple data structure that addresses the former issue in the case of a finite universe $\mathcal{U}$. We then generalize our approach and obtain a slightly more involved data structure supporting infinite/dynamic universes.

Techniques. We map the universe $\mathcal{U}$ into a set of $O(\log |\mathcal{U}|)$-bit labels and implement the mergeable dictionary S as a *compressed trie* with leaves corresponding to the elements of S. This resembles the approach used by Willard [18] to obtain an efficient dynamic predecessor search data structure called the *x-fast trie*. However, as we aim at performing the operations in amortized $O(\log |\mathcal{U}|)$ time, the additional components that make up the x-fast-trie are unnecessary. The tight structure of tries allows us to use a fine-grained potential function for analyzing the amortized cost of mergeable dictionary operations. As a result, both the implementation and the analysis are surprisingly simple. In order to obtain linear space, the paths consisting of trie nodes with a single child are replaced with edges labeled with bit strings. As we work in the word-RAM model, each such label can be stored in $O(1)$ machine words.

In order to allow dynamic (or potentially infinite) universes, we look at the used tries from a somewhat different perspective: each trie can be seen as a subtree of a single tree T representing the entire universe. Our method is to maintain such a tree T representing the part of the universe that contains all the elements of the stored sets. We implement T with a *weight-balanced B-tree* of [1] and represent the individual sets as compressed subtrees of T. This in turn enables us to control the behavior of our potential function when inserting previously unseen elements of $\mathcal{U}$.

Related Work. Ordered dictionaries supporting arbitrary merges but no splits have also been studied, although somewhat implicitly. Brown and Tarjan [4] showed how to merge two

[1] See also [13]. For a detailed reduction in the case of the mergerable dictionary operations, see Section 2.2.
[2] The rank of x in $\mathcal{U}$ is defined as the size of the set $\{y \in \mathcal{U} : y \leq x\}$.

AVL trees of sizes n and m, $n \leq m$, in $O(n \log (m/n))$ worst-case time, which they further proved to be optimal. They also showed that using their merging method, any sequence of merges on a set of n singleton sets can be performed in $O(n \log n)$ time. Hence, assuming no Split operations, each of the operations Insert, Delete, Merge can be performed in amortized $O(\log n)$ time.

An alternative method to handle the case of no splits, called *segment merging*, follows as an easy application of *finger search trees* [10]. A finger search tree is capable of joining two ordered dictionaries of sizes n and m in $O(\log \min(n, m))$ time, as well as splitting an ordered dictionary into parts of sizes n, m in $O(\log \min(n, m))$ time. In order to merge two arbitrarily interleaved ordered dictionaries A and B, where $|A| \leq |B|$, we can partition the set $A \cup B$ into a minimal number of *segments* $\{C_1, \ldots, C_l\}$ such that for each i we have either $C_i \subseteq A$ or $C_i \subseteq B$ and $\max\{C_i\} \leq \min\{C_{i+1}\}$. The finger search tree allows to sequentially extract the segments C_i from either A or B in $O(\log |C_i|)$ time. The segments are then joined in $O\left(\sum_i^l \log |C_i|\right)$ time. As $l \leq |A|$, from the concavity of a logarithmic function it follows that this merging algorithm runs in $O\left(|A| \log \frac{|B|}{|A|}\right)$ time, which is no worse than the algorithm of Brown and Tarjan.

The ordered dictionaries supporting both arbitrary merges and splits have been first considered explicitly by Iacono and Özkan [11]. However, as they point out, the need for a data structure supporting a similar set of operations emerged in several prior works, e.g., the *union-split-find* problem [13], the first non-trivial algorithm for pattern matching in a LZ77-compressed text [6], and the data structure for *mergeable trees* [8]. In particular, Farach and Thorup [6] used a potential function argument to prove (somewhat implicitly) that when using the segment merging strategy, any sequence of Merge and Split operations performed on a collection of subsets with n distinct elements has amortized $O(\log n)$ segments per merge. [3] Hence, segment merging can be used to obtain a mergeable dictionary with $O(\log^2 n)$ amortized bounds, even if one uses an ordinary balanced binary search tree in place of a finger search tree.

On the other hand, Lai [13] proved that if we store individual sets as finger search trees and use the segment merging strategy as discussed above, there exist a sequence of merges and splits that leads to $\Omega(\log^2 n)$ amortized time per Merge. This implies that even if we use an optimal merging algorithm, splits may cause the merges to run asymptotically slower.

As we later show, an optimal solution to the ordered union-split-find problem can be easily obtained by extending our simple data structure for a finite universe. However, in the case of mergeable trees and the compressed pattern matching algorithm of Farach and Thorup, an optimal mergeable dictionary does not immediately lead to a better solution. The mergeable trees [8] generalize mergeable dictionaries in a way analogous to how dynamic trees [16] generalize dynamic paths. Thus, employing the main idea of [16], i.e., decomposing a tree into a set of paths and representing each path with a mergeable dictionary, would lead to amortized $O(\log^2 n)$ time per Merge, a bound already achieved by the data structure of Georgiadis et al. [8]. Obtaining a more efficient data structure for mergeable trees would probably require developing some kind of biased version of a mergeable dictionary.

In the algorithm of Farach and Thorup, a somewhat more powerful variant of a mergeable dictionary is needed. For $\mathcal{U} = \{0, \ldots, N\}$ one also needs to support efficient shifting of all the elements of a set $S \subseteq \mathcal{U}$ by a constant. Even though the data structure of Iacono and Özkan

[3] In fact, the very same potential function was used by Iacono and Özkan [11] to analyze their mergeable dictionary data structure.

can be easily augmented to support such shifts, in our data structure, the representation of a set might dramatically change after such a shift. Nevertheless, the algorithm of Farach and Thorup has another bottleneck and it is not clear how to remove it, even equipped with a mergeable dictionary supporting efficient shifts. It is worth noting that a more efficient solution to LZ77-compressed pattern matching was developed recently, using very different methods [7].

Organization of the Paper. In Section 2 we develop a simple solution for finite universes. We generalize the used methods to obtain a data structure for infinite universes in Section 3. In Section 4 we make some concluding remarks and discuss a few further interesting questions concerning the mergeable dictionaries.

2 A Data Structure for a Finite Universe

In this section we assume that $|\mathcal{U}| = n$ and that the entire universe (along with the order of the elements) is known beforehand: in particular, $\mathcal{U}$ is allowed to be preprocessed during the initialization. We treat n as a measure of the problem size and consequently assume that we operate in the word-RAM model, where arithmetic and bitwise operations on $\lceil \log_2 n \rceil$-bit integers are performed in constant time.

Representation. Let D be the smallest integer such that $2^D \geq n$. Each $x \in \mathcal{U}$ is assigned a bit string $\mathrm{bits}(x)$ of length D such that for $y \in \mathcal{U}$, $x \leq y$, $\mathrm{bits}(x)$ is lexicographically not greater than $\mathrm{bits}(y)$.

Recall that a trie T storing a set of strings W is a rooted tree with single-character *labels* on edges. T has a unique node v_p for each distinct prefix p of some of the strings of W. If sz is a prefix of some word of W and z is a character, then $c_z(v_s) = v_{sz}$ is a child of v_s and the edge $v_s - v_{sz}$ is labeled z. If sz is not a prefix of any word of W, we set $c_z(v_s) = \textbf{nil}$. If some node $v \in T$ corresponds to a prefix p, then we call $p = \ell(v)$ a *label* of the node v. Note that $\ell(v)$ is the string composed of the subsequent characters on the root-to-v path in T. A subtree of T rooted at v, denoted by T_v, can also be seen as a trie, but storing strings (in fact, some suffixes of the words in W) that are $|\ell(v)|$ characters shorter. If $W = \emptyset$, we set the corresponding trie to an empty trie, also denoted by **nil**.

Each set $S \subseteq \mathcal{U}$ is represented as a trie $\mathcal{T}(S)$ storing the strings $\mathcal{B}(S) = \{\mathrm{bits}(s) : s \in S\}$. Note that all the stored strings are of the same length and thus the leaves of $\mathcal{T}(S)$ are at the same depth and correspond to individual elements of $\mathcal{B}(S)$. The tries we use are binary, i.e., each node $v \in \mathcal{T}(S)$ has at most two children $c_0(v)$ and $c_1(v)$. We sometimes call $c_0(v)$ $(c_1(v))$ the left (right respectively) child of v and we call the subtrees $\mathcal{T}(S)_{c_0(v)}$, $\mathcal{T}(S)_{c_1(v)}$ the left and right subtrees of v, correspondingly. Each leaf v stores the value $q(v) \in \mathcal{U}$ such that $\mathrm{bits}(q(v)) = \ell(v)$. See Figure 1 for an example.

Implementing the Operations. We now show how to implement the operations. The operation MAKE-SET returns **nil**.

To insert an element x into S, we descend down the tree $\mathcal{T}(S)$ to the deepest node v corresponding to a prefix of $\mathrm{bits}(x)$ (if $\mathcal{T}(S) = \textbf{nil}$, we first create the root node). We then create $D - |\ell(v)|$ new nodes so that the created leaf has label $\mathrm{bits}(x)$. The operation DELETE(S, x) is basically a reverse of INSERT(S, x): we locate the leaf v with label $\mathrm{bits}(x)$ and sequentially remove its ancestors until we reach a node w with label being a prefix of some string of $\mathcal{B}(S \setminus \{x\})$. Both DELETE and INSERT take $O(D) = O(\log n)$ time.

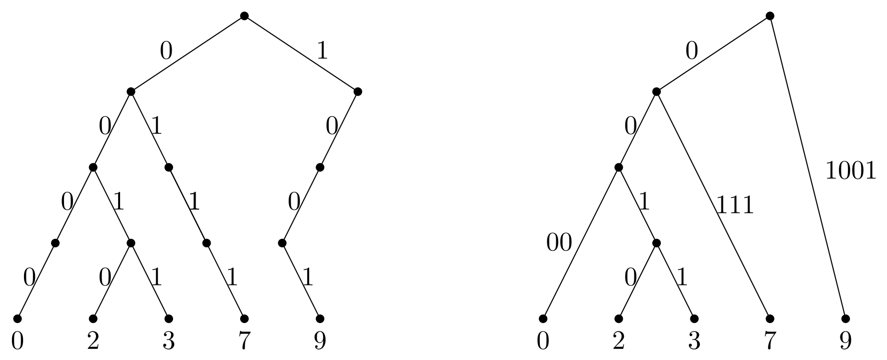

Figure 1 In the left, we have the representation $\mathcal{T}(S)$ (left) of the set $S = \{0, 2, 3, 7, 9\}$. In this example $\mathcal{U} = \{0, \ldots, 15\}$ and thus $D = 4$. We assume that for each $u \in \mathcal{U}$, bits(u) is the 4-bit binary representation of u. In the right, a compressed version $\mathcal{T}^*(S)$ of $\mathcal{T}(S)$ is depicted.

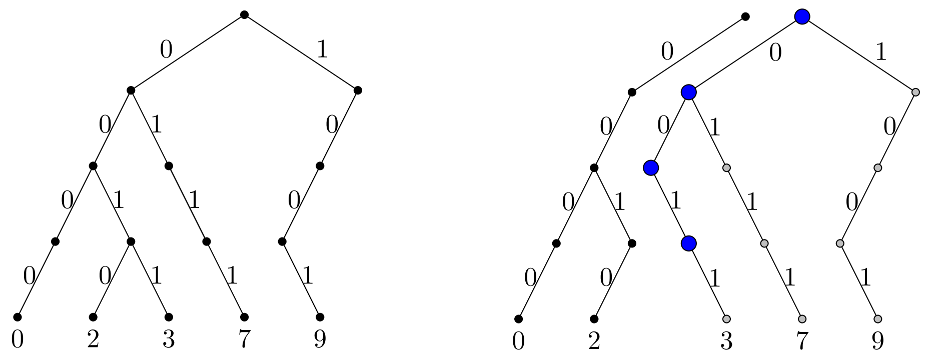

Figure 2 The effect of the call $(A, B) \leftarrow$ Split$(S, 2)$, where $S = \{0, 2, 3, 7, 9\}$. The larger blue nodes denote the only nodes that had to be copied.

To perform Search(S, x), we again descend to the deepest node v such that $\ell(v)$ is a prefix of bits(x). If v is a leaf, then $x \in S$ and we return x. Otherwise, we climb up the tree until we reach a node w such that $c_0(w) \neq$ **nil** and $\ell(w)1$ is a prefix of bits(x). If no such w exists, we return **nil**. Otherwise, we descend to the rightmost leaf w_R in the subtree $\mathcal{T}(S)_{c_0(w)}$ and return the corresponding element $q(w_R)$. One can easily perform these steps in $O(D) = O(\log n)$ worst-case time.

To split the set S around a pivot x, we need to construct two tries $\mathcal{T}^-$ and $\mathcal{T}^+$ such that $\mathcal{T}^-$ stores the strings $\mathcal{B}^- = \{s \in \mathcal{B}(S) : s \leq \text{bits}(x)\}$ and $\mathcal{T}^+$ stores the strings $\mathcal{B}^+ = \mathcal{B}(S) \setminus \mathcal{B}^-$. Both $\mathcal{T}^-$ and $\mathcal{T}^+$ can be obtained by removing some subtrees of $\mathcal{T}(S)$. If $\mathcal{B}^- = \emptyset$, then $\mathcal{T}^- =$ **nil** and $\mathcal{T}^+ = \mathcal{T}(S)$. The case when $\mathcal{B}^+ = \emptyset$ is analogous. Now, let v_l be the leaf of $\mathcal{T}(S)$ such that $\ell(v_l) = \max\{\mathcal{B}^-\}$ and let r be the leaf such that $\ell(v_r) = \min\{\mathcal{B}^+\}$. $\mathcal{T}^-$ ($\mathcal{T}^+$) is exactly the part of $\mathcal{T}$ weakly to the left (to the right, resp.) of the path from the root to v_l (v_r resp.). These paths have at most D common nodes in $\mathcal{T}(S)$ – let us call the set of common nodes C. In order to obtain $\mathcal{T}^-$, we remove all the right subtrees of nodes in C, whereas to construct $\mathcal{T}^+$, a copy of each node of C is made with the left subtree removed (Figure 2). Therefore, the operation Split can be implemented in $O(\log n)$ worst-case time.

The implementation of Merge(A, B) is very simple. We use a recursive function `merge` returning a union of two (possibly empty) tries $\mathcal{T}_1$, $\mathcal{T}_2$. Unless some of the tries $\mathcal{T}_1$, $\mathcal{T}_2$ are non-empty, the tries are required to have equal heights in the interval $[0, D]$. `merge` uses parts of $\mathcal{T}_1$ and $\mathcal{T}_2$ to assemble a trie storing exactly the strings that are stored in $\mathcal{T}_1$ or in

$\mathcal{T}_2$. For $i = 1, 2$, denote by $\mathcal{T}_i^L$ and $\mathcal{T}_i^R$ the left and right subtrees of the root node $\texttt{root}(\mathcal{T}_i)$ of $\mathcal{T}_i$. We have

$$\texttt{merge}(\mathcal{T}_1, \textbf{nil}) = \mathcal{T}_1, \tag{1}$$

$$\texttt{merge}(\textbf{nil}, \mathcal{T}_2) = \mathcal{T}_2, \tag{2}$$

$$\texttt{merge}(\mathcal{T}_1, \mathcal{T}_2) = \texttt{trie}(\texttt{root}(\mathcal{T}_1), \texttt{merge}(\mathcal{T}_1^L, \mathcal{T}_2^L), \texttt{merge}(\mathcal{T}_1^R, \mathcal{T}_2^R)). \tag{3}$$

Here, we use assume that the call $\texttt{trie}(v, \mathcal{T}^L, \mathcal{T}^R)$ creates a trie rooted at v with the left and right subtrees $\mathcal{T}^L$ and $\mathcal{T}^R$ respectively, without copying the subtrees. Clearly, after we call $\texttt{merge}(\mathcal{T}(A), \mathcal{T}(B))$, in each recursive step $\texttt{merge}(\mathcal{T}_1, \mathcal{T}_2)$ such that $\mathcal{T}_1 \neq \textbf{nil}$ and $\mathcal{T}_2 \neq \textbf{nil}$, the labels $\ell(\texttt{root}(\mathcal{T}_1))$ in $\mathcal{T}(A)$ and $\ell(\texttt{root}(\mathcal{T}_2))$ in $\mathcal{T}(B)$ are equal. The correctness of this trie merging procedure can be proved in a bottom-up manner with the following simple structural induction argument. The correctness in cases (1) and (2) is trivial. Consider the case (3) and let h be the height of both $\mathcal{T}_1$ and $\mathcal{T}_2$. Then, either one of the tries $\mathcal{T}_1^L$ and $\mathcal{T}_2^L$ is empty, or both $\mathcal{T}_1^L$ and $\mathcal{T}_2^L$ have height $h - 1$. Thus, by the inductive assumption, $\texttt{merge}(\mathcal{T}_1^L, \mathcal{T}_2^L)$ returns the union of $\mathcal{T}_1^L$ and $\mathcal{T}_2^L$. Symmetrically, $\texttt{merge}(\mathcal{T}_1^R, \mathcal{T}_2^R)$ returns the union of $\mathcal{T}_1^R$ and $\mathcal{T}_2^R$. In the final step, the unions of respective subtries are made the new children of $\texttt{root}(\mathcal{T}_1)$.

Note that each time the case (3) arises, the node $\texttt{root}(\mathcal{T}_2)$ is destroyed.

The Amortized Cost of the Operations. Let $\mathcal{S} = \{S_1, S_2, \ldots, \}$ be the collection of subsets of $\mathcal{U}$ maintained by our data structure. We define the potential $\phi(\mathcal{S})$ to be the sum of sizes of the tries representing individual sets, i.e., $\phi(\mathcal{S}) = \sum_{S \in \mathcal{S}} |\mathcal{T}(S)|$. It is clear that each operation Make-Set, Insert and Split increases $\phi(\mathcal{S})$ by at most $D = O(\log n)$. The operations Delete and Merge can only decrease the potential. We now show that the worst-case running time of the operation $C \leftarrow \text{Merge}(A, B)$ is $O(|\mathcal{T}(A)| + |\mathcal{T}(B)| - |\mathcal{T}(A \cup B)| + 1)$, i.e., it is proportional to the decrease of the potential. Indeed, consider the call $\texttt{merge}(\mathcal{T}_1, \mathcal{T}_2)$ which is not the topmost call $\texttt{merge}(\mathcal{T}(A), \mathcal{T}(B))$. If $\mathcal{T}_1 \neq \textbf{nil}$ and $\mathcal{T}_2 \neq \textbf{nil}$, we can charge the cost of this call (not including the recursive calls) to the destroyed root of $\mathcal{T}_2$. Otherwise, the parent invocation $\texttt{merge}(*, *)$ was of type (3) and thus we can charge this call to the destroyed parent of $\mathcal{T}_2$. Consequently, for each destroyed node, at most 3 calls to $\texttt{merge}$ are charged to that node. The total number of destroyed nodes after calling $\texttt{merge}(\mathcal{T}(A), \mathcal{T}(B))$ is $|\mathcal{T}(A)| + |\mathcal{T}(B)| - |\mathcal{T}(A \cup B)|$.

The amortized cost of an operation is defined as its actual cost plus the increase of the potential. Hence, both amortized and worst-case costs of the operations Insert, Delete and Split on $\mathcal{S}$ are $O(\log n)$, whereas the amortized cost of Merge is $O(1)$.

▶ **Theorem 1.** *Let $|\mathcal{U}| = n$. There exists a data structure supporting* Insert, Delete *and* Split *in $O(\log n)$ amortized and worst-case time. The operation* Merge *takes $O(1)$ amortized time. The operation* Search *can be performed in $O(\log n)$ worst-case time.*

The Ordered Union-Split-Find Data Structure. One can easily extend our approach to implement the ordered union-split-find data structure. Assume that the collection $\mathcal{S}$ forms a partition of $\mathcal{U}$, i.e., the elements of $\mathcal{S}$ are disjoint and $\bigcup \mathcal{S} = \mathcal{U}$. For each $u \in \mathcal{U}$ we store a pointer to the leaf of a unique trie $\mathcal{T}(S)$ such that $u \in S$. Additionally, each trie node is accompanied with a parent pointer.

When performing a Split operation, we update the parent pointers of all the newly created (copied) nodes and their children. During Merge, parent pointers are updated each time a node is assigned new children (case (3) of the $\texttt{merge}$ procedure). Insert and Delete can also be easily extended to update the appropriate parent pointers. The maintenance of parent pointers does not influence the asymptotic worst-case and amortized time bounds of the operations.

Answering a FIND(u) query boils down to climbing up the appropriate trie using the parent pointers and returning the root of $\mathcal{T}(S)$, where $u \in S$.

2.1 Obtaining Linear Space

The above construction might incur $\Omega(\log n)$-space overhead per each stored element, e.g., if every set of the collection is a singleton. This can be easily avoided by dissolving all the non-root non-leaf trie nodes having a single child. Now, each edge can be labeled with at most D bits (stored in a single word), whereas the total length of the labels on any root-to-leaf path remains D. As this results in all the nodes having either 0 or 2 children, a compressed trie with t leaves has now at most $2t - 1$ nodes in total. Thus, any set S can be stored in $O(|S|)$ machine words. The compressed version of $\mathcal{T}(S)$ obtained this way is denoted by $\mathcal{T}^*(S)$. See Figure 1 for an example.

All the discussed operations can be implemented by introducing a layer of abstraction over $\mathcal{T}^*(S)$, so that we are allowed to operate on $\mathcal{T}(S)$ instead. Each time we access a node $v \in \mathcal{T}^*(S)$, we can "decompress" its outgoing edges by creating at most two additional nodes c_0, c_1 and make them the children of v, so that the labels of the edges (v, c_0) and (v, c_1) have single-bit labels. All nodes of $\mathcal{T}(S)$ "touched" by an operation are processed bottom-up after the operation completes and the non-root nodes of $\mathcal{T}(S)$ with a single child are dissolved back.

2.2 Lower Bound

For completeness, we prove the following lemma, which establishes the optimality of our data structure, as far as the cost of the most expensive operation is concerned.

▶ **Lemma 2.** *Let $|\mathcal{U}| = \Omega(n^2)$. At least one of the mergeable dictionary operations* SPLIT*,* MERGE *and* SEARCH *requires $\Omega(\log n^2) = \Omega(\log n)$ time.*

Proof. Let $\mathcal{U} = \{(x, y) : x \in \{0, \dots, n\}, y \in \{1, \dots, n\}\}$ and suppose the order of $\mathcal{U}$ is such that $(x_1, y_1) \le (x_2, y_2)$ if and only if $x_1 < x_2$ or $x_1 = x_2 \wedge y_1 \le y_2$.

Pătraşcu and Demaine [15] considered the following dynamic permutation composition problem. Let $\pi_1, \dots, \pi_n$ be the permutations of the set $\{1, \dots, n\}$. Initially $\pi_i = \mathbf{id}$ for all i. We are to support two operations:

- UPDATE(i, π'): set $\pi_i \leftarrow \pi'$,
- VERIFY(i, π'): check if $\pi_i \circ \pi_{i-1} \circ \ldots \circ \pi_1 = \pi'$.

▶ **Lemma 3** ([15]). *Any data structure requires $\Omega(n^2 \log n)$ expected time to support a sequence of n* UPDATE *operations and n* VERIFY *operations.*

We show how to reduce this problem to maintaining a certain partition of $\mathcal{U}$. In our reduction we maintain n sets $S_1, \dots, S_n$ so that after each UPDATE operation, for each $j = 1, 2, \dots, n$ we have

$$S_j = \{(0, j), (1, \pi_1(j)), (2, \pi_2(\pi_1(j))), \dots, (n, \pi_n(\dots(\pi_1(j))))\}. \tag{4}$$

Clearly, $S_i \subseteq \mathcal{U}$. Note that for each $k \in [0, n]$ and $j \in [1, n]$ we can find $\pi_k(\dots(\pi_1(j)))$ with a single SEARCH($S_j, (k, n)$) query. Thus, VERIFY(i, π') can be implemented with n SEARCH operations: for each $j = 1, \dots, n$ we check whether SEARCH($S_j, (i, n)$) $= (i, \pi'(j))$.

In order to implement $\textsc{Update}(i, \pi')$, we first execute $(A_j, B_j) \leftarrow \textsc{Split}(S_j, (i-1, n))$ for each $j = 1, 2, \ldots, n$. Note that $A_j \neq \emptyset$ and $B_j \neq \emptyset$. The last element $(i-1, a_j)$ of each A_j can be found with a single $\textsc{Search}$ operation. Similarly, the first element (i, b_j) of each B_j can be found with a single $\textsc{Search}$. The values a_j are distinct and so are the values b_j. The last step is to create each set S_j by merging A_j with a unique B_k satisfying $b_k = \pi'(a_j)$. It is easy to see that $S_j = \textsc{Merge}(A_j, B_k)$ satisfies (4) with $\pi_i = \pi'$.

To conclude, n $\textsc{Update}$ and $\textsc{Verify}$ can be implemented with $O(n^2)$ $\textsc{Search}$, $\textsc{Split}$ and $\textsc{Merge}$ operations on a mergeable dictionary. Being able to execute each of these mergeable dictionary in $o(\log n)$ amortized time would contradict Lemma 3. ◀

3 Handling Dynamic and Infinite Universes

Overview. In the previous section we have only supported subsets of a finite universe $\mathcal{U}$. The critical idea was that we could assign a $O(\log |\mathcal{U}|)$-bit label $\text{bits}(x)$ to each $x \in \mathcal{U}$ so that $x \leq y$ implied $\text{bits}(x) \leq \text{bits}(y)$. This allowed us to store the sets in trees of small depth and predictable structure, which was consistent among the representations of different sets. If the universe can grow or is infinite, e.g. $\mathcal{U} = \mathbb{R}$, it is not clear how to assign such labels beforehand, during the initialization.

In this section we aim at achieving amortized $O(\log N)$ bounds for all mergeable dictionary operations on the collection $\mathcal{S} = \{S_1, S_2, \ldots, \}$, where $N = \sum_{S \in \mathcal{S}} |S|$. At any time, N is no more than the number of $\textsc{Insert}$ operations performed.

Imagine a perfect binary tree $\bar{T}$ with 2^B leaves such that each edge to the left child is labeled with 0 and each edge to the right child is labeled with 1. The (uncompressed) tries used in the previous section can be seen as subtrees of $\bar{T}$. More formally, $\mathcal{T}(S)$ can be obtained from $\bar{T}$ by removing all the subtrees $\bar{T}_v$ of $\bar{T}$ such that $\bar{T}_v$ does not contain any leaf corresponding to an element of S.

Our strategy is to maintain a similar "global" tree T, so that the representations of individual sets constitute subtrees of T. We incrementally store all the elements of $\bigcup \mathcal{S}$ in the leaves of a *weight-balanced B-tree* T [1]. As opposed to $\bar{T}$, T is not binary. However, it still allows us to keep all the elements as leaves at the same depth of order $O(\log N)$ and add new elements in logarithmic time. One crucial property of a weight-balanced B-tree allows us to still represent the sets $S \in \mathcal{S}$ as compressed subtrees $\mathcal{T}(S)$ of T, even though T undergoes updates. The potential function ϕ we use to analyze the amortized performance of the operations is exactly the same as previously, i.e., $\phi(\mathcal{S}) = \sum_{S \in \mathcal{S}} |\mathcal{T}(S)|$.

The weight-balanced trees have been previously used in the context of the *monotonic list labeling* problem, which typically asks to maintain a totally ordered set Q and $O(\log |Q|)$-bit labels of the elements of Q subject to insertions of a new element y to Q between two existing elements $x < z$, $x, z \in Q$. Several optimal data structures exist for this problem (e.g. [3, 5, 12]): each supports inserting a new element in $O(\log |Q|)$ amortized time and guarantees that such insertion incurs amortized logarithmic number of relabels of existing elements in Q. In particular, Kopelowitz [12] used the weight-balanced B-tree to obtain optimal worst-case bounds for this problem. However, it is not clear how to use a monotonic list labeling data structure as a black-box in our case. Instead of keeping the number of relabels small, we rather need to keep the *potential increase* per insertion small.

We again assume that we work in the word-RAM model, so that the operations on $O(\log N)$-bit integers take $O(1)$ time and the space is measured in the number of words.

The Weight-Balanced B-tree. A weight-balanced B-tree T with a (constant) branching parameter $a \geq 4$ stores its elements in leaves. For an internal node $v \in T$, we define its weight $w(v)$ to be the number of leaves among the descendants of v. The following are the key invariants that define a weight-balanced B-tree:

1. All the leaves of T are at the same depth.
2. Let *height* of a node $v \in T$ be the number of edges on the path from v to any leaf. An internal node v of height h has weight less than $2a^h$.
3. Except for the root, an internal node of height h has weight greater than $\frac{1}{2}a^h$.

▶ **Lemma 4** ([1]). *Assume T is a weight-balanced B-tree with branching parameter a.*

- *All internal nodes of T have at most $4a$ children.*
- *Except for the root, all internal nodes of T have at least $a/4$ children.*
- *If T contains n elements, then the height of T is $O(\log_a n)$.*

For each internal node v and its two children v_1, v_2 such that v_1 is to the left of v_2, the elements in the subtree of v_1 are no larger than any of the elements in the subtree of v_2. Each internal node stores the minimum and maximum elements stored in its subtree. This information allows us to drive the searches down the tree.

To insert an element e into T, we first descend down T to find an appropriate position for the new leaf corresponding to e. The insertion of a new leaf may result in some nodes getting out of balance. Let $v \in T$ be the deepest node such that $w(v) = 2a^h$ at that point, where h is the height of v. As each child of v has weight less than $2a^{h-1}$, one can split the children of v into two groups of consecutive children C_-, C_+ so that the total weight of nodes in any group is in the interval $(a^h - 2a^{h-1}, a^h + 2a^{h-1})$. We have $a^h - 2a^{h-1} = a^h(1 - 2/a) \geq \frac{1}{2}a^h$ and similarly $a^h + 2a^{h-1} \leq \frac{3}{2}a^h$. v is split into two nodes v_- and v_+ so that the elements of C_- become the children of v_- and the elements of C_+ become the children of v_+. We have $w(v_-), w(v_+) \in (\frac{1}{2}a^h, \frac{3}{2}a^h)$, so both v_- and v_+ satisfy the balance constraints. If v is not the root before the split, nodes v_-, v_+ are made the children of the parent of v in place of v. Otherwise, a new root with children v_-, v_+ is created. The process is repeated until all the nodes are balanced and thus the insertion takes $O(\log n)$ time, where n is the number of elements stored in T.

To delete an element from a weight-balanced B-tree, we mark the corresponding leaf as deleted, which takes $O(\log n)$ time. Once more than a half of the stored elements are marked as deleted, the entire tree is rebuilt (the elements marked as deleted are skipped) in $O(n)$ time. This can be charged to the deletions that left the marked leaves. Thus, the amortized time complexity of a deletion is $O(\log n)$ as well.

The main advantage of a weight-balanced B-tree is the fact that for any newly created node v of height h, at least $\Omega(a^h)$ leaves have to be inserted into the subtree of v to cause the split of v. Therefore, when the node v is split, we can afford to spend $O(a^h)$ time for instance for traversing all the leaf descendants of v or updating some secondary data structure that accompanies v. This work can be charged to $\Omega(a^h)$ insertions into the subtree of v that take place between the creation of v and its split. The total amortized time spent on the "additional maintenance" per insertion is thus proportional to the depth of T, i.e., $O(\log n)$.

Labeling the Tree T. Each time an operation $\textsc{Insert}(S, u)$ (for $u \notin S$) is issued, u is stored as a leaf at an appropriate position of the weight-balanced B-tree T. We stress that there is a separate leaf for each (u, S) pair, where $u \in S$, i.e., multiple leaves may correspond to a single $u \in S$. Such a design decision is explained later on (see Remark 1).

We introduce the labels $\ell(v)$ of the vertices of T such that for any $v_1, v_2 \in T$, where v_1 is an ancestor of v_2, the path $v_1 \to v_2$ in T (i.e., the indices of children of subsequent nodes to be entered when following the path $v_1 \to v_2$ in T) can be computed based only on $\ell(v_1)$ and $\ell(v_2)$. As the trees $\mathcal{T}(S)$ are stored in a compressed way, the labels will help navigate $\mathcal{T}(S)$ while performing the operations on S. We now define the labels $\ell(v)$ formally.

Let H be the height of T. The label $\ell(v)$ of a node v of height h consists of H blocks of $\lceil \log_2 (4a+1) \rceil = O(1)$ bits. Clearly, $\ell(v)$ can be stored in a constant number of machine words. We number the blocks with integers $0, \ldots, H-1$ starting at the block containing the least significant bits. Let $v_H \to \ldots \to v_h = v$ be the root-to-v path in T. We define $z(v_i)$ to be the (0-based) position of v_i among the children of v_{i+1} in the left-to-right order. At any time (even immediately before the split) v_{i+1} has at most $4a+1$ children. Thus, $\lceil \log_2 (4a+1) \rceil$ bits suffice to store $z(v_i)$. For $i \in [h, H-1]$, we define the the bits of the i-th block of $\ell(v)$ to contain exactly the value $z(v_i)$. The blocks $h-1, \ldots, 0$ of $\ell(v)$ are filled with zeros. Note that the label $\ell(v)$ can be computed in $O(1)$ time based on the label of its parent in T using standard bitwise operations.

Storing the Individual Sets. Let $S \in \mathcal{S}$. Denote by $L(S)$ the set of leaves of T that correspond to the elements of S. Recall that each $\mathcal{T}(S)$ is in fact the tree T with subtrees containing no leaves of $L(S)$ removed. Again, in the compressed version $\mathcal{T}^*(S)$ we only keep the nodes v of $\mathcal{T}(S)$ such that either v is the root of $\mathcal{T}(S)$, $v \in L(S)$, or for at least two children c_1, c_2 of v, the subtree rooted at c_i $(i = 1, 2)$ contains at least one leaf of $L(S)$. An example tree T along with the compressed and uncompressed representation of a set $S \in \mathcal{S}$ is presented in Figure 3.

Each node v of $\mathcal{T}^*(S)$ is a copy of the corresponding node of T and v stores a pointer to the original node of T. Every node of T also maintains a list of its copies used in the representations $\mathcal{T}^*(S)$ of the sets of $S \in \mathcal{S}$. The pointers between $\mathcal{T}^*(S)$ and T along with the labels $\ell(*)$ allow to temporarily decompress the relevant parts of $\mathcal{T}^*(S)$ when performing the operations Insert, Delete, Merge and Split, analogously as in Section 2.1.

Differences in the Implementation of Operations. In comparison to the data structure of Section 2, the implementations of operations Insert, Delete, Split do not generally change. We basically replace values bits$(*)$ with labels $\ell(v)$. Each operation Insert(S, x) first inserts a leaf into T and thus the new element is given a label before we modify $\mathcal{T}^*(S)$.

When the operation Search(S, x) is performed, we first find in $O(\log N)$ worst-case time a leaf in T that corresponds to a maximum value $y \in \bigcup \mathcal{S}$ such that $y \le x$. Note that Search(S, y) computes the same value as Search(S, x), but now y corresponds to some leaf of T and it has a label, so we can proceed analogously as in Section 2.

Handling the Splits of the Nodes of T. Suppose a new leaf is added to T at an appropriate position among the children of some height-1 node v. The labels of all the children of the node v might have to be recomputed.

The insertion may also cause the splits of some internal nodes, as described previously. Let v be an internal non-root node of height h that is split into two nodes v_-, v_+. Denote by p the parent of v. After the split, both the values $z(*)$ of the children of p and the values $z(*)$ of the children of v_-, v_+ may change. This implies that for each v of the $O(a^h)$ nodes of the subtree rooted at p, the contents of at most two blocks (namely, the blocks h and $h-1$) of $\ell(v)$ may change. As discussed above, we can afford going through all these nodes without sacrificing our amortized $O(\log n)$ insertion bound.

The split also requires to repair some of the representations $\mathcal{T}^*(S)$. First assume that S is such that there is no copy of v included in $\mathcal{T}^*(S)$. Then, either there is no copy of v in

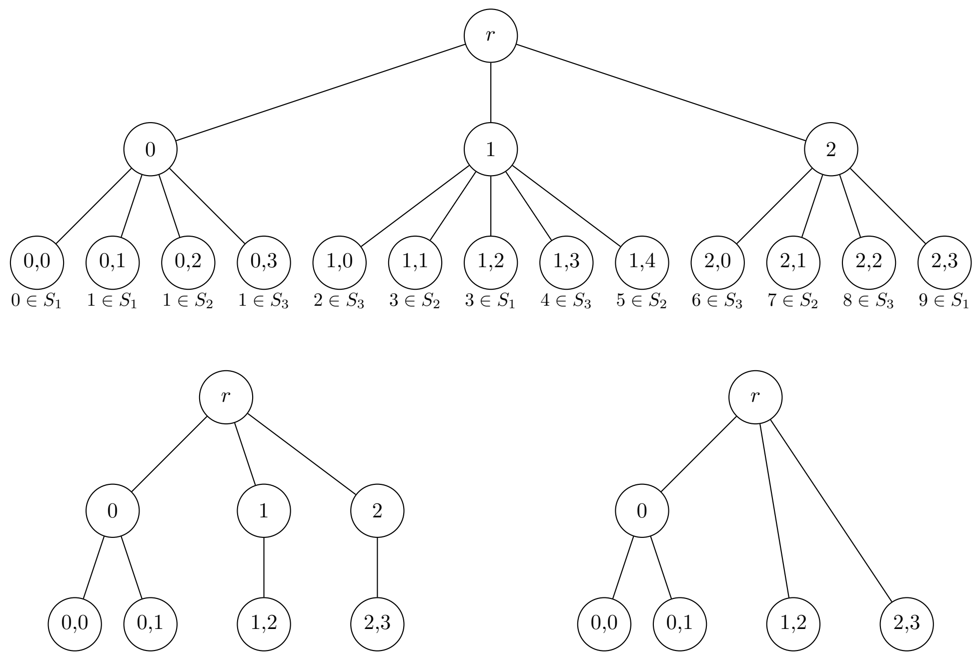

Figure 3 Let $\mathcal{S} = \{S_1, S_2, S_3\}$, where $S_1 = \{0, 1, 3, 9\}$, $S_2 = \{1, 3, 5, 7\}$ and $S_3 = \{1, 2, 4, 6, 8\}$. A weight-balanced B-tree T (with branching factor $a = 4$) that could arise when constructing $\mathcal{S}$ is depicted at the top. The values in the nodes are their labels $\ell(*)$. The trie $\mathcal{T}(S_1)$ can be seen in the bottom left. The compressed version $\mathcal{T}^*(S_1)$ is illustrated in the bottom-right.

$\mathcal{T}(S)$ and thus $\mathcal{T}(S)$ contains no leaves of the subtree of T rooted at v, and we are done, or a copy v_S is a node of $\mathcal{T}(S)$. Then, v_S has a single child c in $\mathcal{T}(S)$ and therefore, after the split v_S should be replaced with a copy of either v_- or v_+. However, as v_- or v_+ would have been dissolved in $\mathcal{T}^*(S)$, we actually do not need to update $\mathcal{T}^*(S)$ at all. Moreover, in this case the size $|\mathcal{T}(S)|$ does not change and neither does the potential ϕ.

Let us now suppose that a copy v_S of v is a node of $\mathcal{T}^*(S)$ and denote by q the parent of v_S in $\mathcal{T}^*(S)$. If all the children of v_S in $\mathcal{T}(S)$ are contained in the subtree of v_- of T after the split, it suffices to replace v in $\mathcal{T}^*(S)$ with a copy of v_- and update the pointers between $\mathcal{T}^*(S)$ and T. The case when all the children of v in $\mathcal{T}(S)$ are contained in the subtree of v_+ of T is similar. Both this cases required $O(1)$ time to process, but ϕ does not change. The last case is when some two children c_-, c_+ of v in $\mathcal{T}(S)$ are contained in the subtrees of v_- and v_+, respectively. Then, copies of both v_- and v_+ have to be introduced in $\mathcal{T}(S)$ in place of v. Thus, the potential ϕ increases by 1 in this case. As far as the compact representation $\mathcal{T}^*(S)$ is concerned, a copy of p has to be included in $\mathcal{T}^*(S)$, if it is not already there. The copies of v_- and v_+ are created in $\mathcal{T}^*(S)$ only if they would not be dissolved afterwards. We skip the description of the case when v is the root, as it is analogous.

We conclude that it takes $O(1)$ time to repair $\mathcal{T}^*(S)$ in any case and the potential ϕ increases by at most 1 per repair. The number of repairs incurred by the split of v is not more than the number of leaves of the subtree rooted in v, i.e., $O(a^h)$, as for each representation of S that actually needs to be repaired, $\mathcal{T}^*(S)$ has to contain (a copy of) some leaf of the subtree T_v. Finally, note that a single leaf of T_v has a copy in at most one representation $\mathcal{T}^*(S)$.

Thus, the repairs made during the maintenance of the tree T increase the potential by amortized $O(\log N)$ per INSERT operation.

▶ Remark 1. Imagine the tree T was allowed to contain only a single leaf for each element $u \in \bigcup \mathcal{S}$. Suppose that for each $S_i \in \mathcal{S} = \{S_1, \ldots, S_m\}$, $S_i = \{u\}$, for some $u \in \mathcal{U}$. Each split of an ancestor of the leaf corresponding to u in T would cause a repair of m set representations. Let $x_0 \in \mathcal{U}$ be such that $x_0 > u$. Now suppose the adversary sequentially performs $\textsc{Insert}(S_i, x_i)$, where $u < x_i < x_{i-1}$ for $i = 1, \ldots, m$. $\Omega(m)$ of such operations would lead to a split of some ancestor of the leaf u, and the total running time of these sequence could be as much as $\Omega(m^2)$.

The weight-balanced B-tree does only guarantees that the total size of split subtrees after m insertions is $O(m \log m)$. However, as the above example shows, the total number of times when some particular leaf is contained in a subtree undergoing a split might be $\Omega(m^2)$. That is why we decided to store duplicate leaves per single value $u \in \mathcal{U}$, if u is a frequent element in the stored sets.

The Amortized Analysis of the Operations. Each of the operations INSERT, DELETE, SPLIT runs in amortized $O(\log N)$ time, as discussed above. Also it is clear that the (amortized) potential increase per each of this operations is $O(\log N)$.

The operation MERGE is implemented almost identically as in Section 2. We only need to make sure that the modified recursive procedure `merge` is always fed two copies of the same node $v \in T$ as arguments. As each node has $O(1)$ children, we can charge a constant amount of work to the nodes of $\mathcal{T}(*)$ destroyed during the merging process. Consequently, MERGE runs in time proportional to the decrease of the potential ϕ.

▶ **Theorem 5.** *There exists a data structure supporting all the mergeable dictionary operations on a collection $\mathcal{S}$ of subsets of $\mathcal{U}$ in amortized $O(\log N)$ time, where $N = \sum_{S \in \mathcal{S}} |S|$.*

4 Conclusions and Open Problems

In this paper we developed a simpler solution for the mergeable dictionary problem. We also addressed the issue of supporting dynamic/infinite universes raised in [11].

We can see two interesting further questions about mergeable dictionaries. First, in the finite universe case, the amortized cost of all the operations was logarithmic in the size of the universe. On the other hand, for the infinite case, we only managed to obtain amortized $O\left(\log \sum_{S \in \mathcal{S}} |S|\right) = O\left(\log |\mathcal{S}| + \log |\bigcup \mathcal{S}|\right)$ bounds. We can think of the size of the "used universe" to be $|\bigcup \mathcal{S}|$. Thus, in the infinite universe case, our time bounds are also logarithmic in the number of stored sets, which might be of order much larger than $|\bigcup \mathcal{S}|$. It would be interesting to know if one could remove this dependence.

Second, our solution for infinite universes involves maintaining a "common infrastructure" T in order to limit the potential growth. Is there a way to implement a mergeable dictionary in a dynamic/infinite universe regime without any common infrastructure, so that the representation of a set does not depend on the shapes of other stored sets? In particular, is the splay tree [17] a mergeable dictionary with such a property?

Acknowledgments. We thank the anonymous reviewers for their helpful comments.

References

1 Lars Arge and Jeffrey Scott Vitter. Optimal external memory interval management. *SIAM J. Comput.*, 32(6):1488–1508, 2003. doi:10.1137/S009753970240481X.

2 Amitabha Bagchi, Adam L. Buchsbaum, and Michael T. Goodrich. Biased skip lists. *Algorithmica*, 42(1):31–48, 2005. `doi:10.1007/s00453-004-1138-6`.

3 Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *Algorithms – ESA 2002, 10th Annual European Symposium, Rome, Italy, September 17-21, 2002, Proceedings*, pages 152–164, 2002. `doi:10.1007/3-540-45749-6_17`.

4 Mark R. Brown and Robert Endre Tarjan. A fast merging algorithm. *J. ACM*, 26(2):211–226, 1979. `doi:10.1145/322123.322127`.

5 Paul F. Dietz and Daniel Dominic Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 365–372, 1987. `doi:10.1145/28395.28434`.

6 Martin Farach and Mikkel Thorup. String matching in lempel-ziv compressed strings. *Algorithmica*, 20(4):388–404, 1998. `doi:10.1007/PL00009202`.

7 Paweł Gawrychowski. Pattern matching in lempel-ziv compressed strings: Fast, simple, and deterministic. In *Algorithms – ESA 2011 – 19th Annual European Symposium, Saarbrücken, Germany, September 5-9, 2011. Proceedings*, pages 421–432, 2011. `doi:10.1007/978-3-642-23719-5_36`.

8 Loukas Georgiadis, Haim Kaplan, Nira Shafrir, Robert Endre Tarjan, and Renato Fonseca F. Werneck. Data structures for mergeable trees. *ACM Transactions on Algorithms*, 7(2):14, 2011. `doi:10.1145/1921659.1921660`.

9 Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978*, pages 8–21, 1978. `doi:10.1109/SFCS.1978.3`.

10 Scott Huddleston and Kurt Mehlhorn. A new data structure for representing sorted lists. *Acta Inf.*, 17:157–184, 1982. `doi:10.1007/BF00288968`.

11 John Iacono and Özgür Özkan. Mergeable dictionaries. In *Proceedings of the 37th International Colloquium Conference on Automata, Languages and Programming*, ICALP'10, pages 164–175, Berlin, Heidelberg, 2010. Springer-Verlag. `doi:10.1007/978-3-642-14165-2_15`.

12 Tsvi Kopelowitz. On-line indexing for general alphabets via predecessor queries on subsets of an ordered list. In *53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012, New Brunswick, NJ, USA, October 20-23, 2012*, pages 283–292, 2012. `doi:10.1109/FOCS.2012.79`.

13 Katherine Jane Lai. Complexity of union-split-find problems. Master's thesis, Massachusetts Institute of Technology, 2008. URL: `http://erikdemaine.org/theses/klai.pdf`.

14 William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990. `doi:10.1145/78973.78977`.

15 Mihai Pătraşcu and Erik D. Demaine. Lower bounds for dynamic connectivity. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*, pages 546–553, 2004. `doi:10.1145/1007352.1007435`.

16 Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, June 1983. `doi:10.1016/0022-0000(83)90006-5`.

17 Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary trees. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983, Boston, Massachusetts, USA*, pages 235–245, 1983. `doi:10.1145/800061.808752`.

18 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space theta(n). *Inf. Process. Lett.*, 17(2):81–84, 1983. `doi:10.1016/0020-0190(83)90075-3`.