Mergeable Dictionaries

John Iacono Özgür Özkan

February 23, 2010

Abstract

A data structure is presented for the Mergeable Dictionary abstract data type, which supports the following operations on a collection of disjoint sets of totally ordered data: Predecessor-Search, Split and Merge. While Predecessor-Search and Split work in the normal way, the novel operation is Merge. While in a typical mergeable dictionary (e.g. 2-4 Trees), the Merge operation can only be performed on sets that span disjoint intervals in keyspace, the structure here has no such limitation, and permits the merging of arbitrarily interleaved sets. Tarjan and Brown present a data structure [4] which can handle arbitrary Merge operations in $\mathcal{O}(\log n)$ amortized time per operation if the set of operations is restricted to exclude the Split operation. In the presence of Split operations, the amortized time complexity of their structure becomes $\Omega(n)$. A data structure which supports both Split and Merge operations in $\mathcal{O}(\log^2 n)$ amortized time per operation was given by Farach and Thorup [6]. In contrast, our data structure supports all operations, including Split and Merge, in $O(\log n)$ amortized time, thus showing that interleaved Merge operations can be supported at no additional cost vis-à-vis disjoint Merge operations.

Dagstuhl Seminar Proceedings 10091
Data Structures
http://drops.dagstuhl.de/opus/volltexte/2010/2685

1 Introduction

Consider the following operations on a data structure which maintains a dynamic collection $\mathcal{S}$ of disjoint sets $\{S_1, S_2, \ldots\}$ which partition some totally ordered universal set $\mathcal{U}$:

- $S \leftarrow \textsc{Find}(x)$: Returns the set $S \in \mathcal{S}$ that contains x.
- $p \leftarrow \textsc{Search}(S, x)$: Returns the largest element in S that is at most x.
- $(A, B) \leftarrow \textsc{Split}(S, x)$: Splits S into two sets $A = \{y \in S \,|\, y \leq x\}$ and $B = \{y \in S \,|\, y > x\}$. S is removed from $\mathcal{S}$ while A and B are inserted.
- $C \leftarrow \textsc{Merge}(A, B)$: Creates $C = A \cup B$. C is inserted into $\mathcal{S}$ while A and B are removed.

We call a data structure that supports these operations a *Mergeable Dictionary*. In this paper we present a data structure, which implements these operations in amortized time $\mathcal{O}(\log n)$, where n is the total number of items in $\mathcal{U}$. What makes the concept of a Mergeable Dictionary interesting is that the Merge operation does not require that the two sets being merged occupy disjoint intervals in keyspace. As we discuss in full detail in Section 1.2, a data structure for merging arbitrarily interleaved sets has appeared independently in the context of Union-Split-Find, Mergeable Trees and string matching in Lempel-Ziv compressed text. In all three cases, a $o(\log^2 n)$ bound on mergeable dictionary operations could not be achieved. We present a data structure that is able to break through this bound though use of a novel weighting scheme applied to an extended version of the Biased Skip List data structure [2]. Another alternative would be to extend and use Biased Search Trees [3] but we believe extending Biased Skip Lists will be easier, at least in terms of presentation.

We first present a high-level description of the core data structure of the previous work, and show at a high level the method and motivation we use to improve the runtime. Given this description, we then can discuss in some detail the three aforementioned works. Finally, we present the full details of our result.

1.1 High-Level Description

The basic idea of the structure is simple. As a first attempt we show how to achieve $\mathcal{O}(\log^2 n)$ time, which we outline here and fully present in Section 2. Store each set using an existing dictionary that supports Search, Split and Join[1] in $\mathcal{O}(\log n)$ time (e.g. 2-4 trees). Thus, the only operation that requires a non-wrapper implementation is Merge. One first idea would be to implement Merge in linear time as in *Merge-Sort*, but this performs poorly, as one would expect. A more intelligent idea is to use a sequence of searches to determine how to partition the two sets into sets of *segments* that span maximal disjoint intervals. Then, use a sequence of Splits to split each set into the segments and a sequence Join operations to piece together the segments in sorted order. As the number of segments between two sets being merged could be $\Theta(n)$, the worst-case runtime of such an implementation is $\mathcal{O}(n \log n)$, even worse than the $\mathcal{O}(n)$ of a brute-force merge. However, it is impossible to perform many Merges with a high number of segments, and an amortized analysis bears this out; there are only $\mathcal{O}(\log n)$ amortized segments per Merge. Thus, since each segment can be processed in $\mathcal{O}(\log n)$ time, the total amortized cost per Merge operation is $\mathcal{O}(\log^2 n)$.

[1] Join merges two sets but requires that the sets span disjoint intervals in keyspace

In [9], it was shown that there are sequences of operations that have $\Theta(\log n)$ amortized segments per MERGE. This, combined with the worst-case lower bound of $\Omega(\log n)$ for the dictionary operations needed to process each segment seemingly gives a strong argument for a $\Omega(\log^2 n)$ lower bound, which was formally conjectured by Lai. It would appear that any effort to circumvent this impediment would require abandoning storing each set in sorted order. We show this is not necessary, as a weighting scheme allows us finesse the balance between the cost of processing each segment, and the number of segments to be processed; we, in essence, prevent the worst-case of these two needed events from happening simultaneously. Our scheme, combined with an extended version of Biased Skip Lists, allows us to speed up the processing of each segment to $\mathcal{O}(1)$ when there are many of them, yet gracefully degrades to the information-theoretically mandated $\Theta(\log n)$ worst-case time when there are only a constant number of segments. The details, however, are numerous. We show in Section 3.3 how to augment Biased Skip Lists to support weighted finger versions of operations that we need. Given this, in Section 4 a full description of our structure is presented, and in Section 5 the runtime analysis is proved.

1.2 Relationship to existing work

The underlying problem addressed here has come up independently three times in the past, in the context of Union-Split-Find, Mergeable Trees and string matching in Lempel-Ziv compressed text. All three of these results, which were initially done independently of each other, bump up against the same $\mathcal{O}(\log^2 n)$ issue with merging, and all have some variant of the $\mathcal{O}(\log^2 n)$ structure outlined above at their core. While the intricacy of the latter two precludes us claiming here to reduce the squared logarithmic terms in their runtimes, we believe that we have overcome the fundamental obstacle towards this improvement.

1.2.1 Searching in Lempel-Ziv

In the paper of Farach and Thorup, an algorithm is presented for string matching in a Lempel-Ziv compressed string [6]. They show how to search for a string of length p in a compressed string of length n that was compressed by a factor of f in time $\mathcal{O}(p + n \log^2 f)$. The algorithm is complex, but at its heart needs a data structure that can hold a set S of at most n integers in the range $1 \ldots \mathcal{U}$, and can perform an operation that shifts all values in some interval in S by some specified integer amount, so long as all values remain the range $1 \ldots \mathcal{U}$. They show how to do this in time $\mathcal{O}(\log \mathcal{U} \log N)$ using a variant of the $\mathcal{O}(\log^2 n)$ structure described above. This variant allows a whole set to be shifted, and thus shifting an interval can be done with two splits and two merges in addition to a shift. The potential function they use to bound the number of segments is the same one as in our presentation of the $\mathcal{O}(\log^2 n)$ structure. A simple extension of our structure can speed up the needed shifts by a $\log n$ factor. However, since they do not completely use the aforementioned shifting data structure as a black box (there is an "unwinding" of some work performed there, among other subtleties), we leave the improvement of the runtime to $\mathcal{O}(p+n \log f)$ only as a conjectured application of our result.

1.2.2 Mergable Trees

In the paper of Georgiadis, Tarjan, Werneck [8] and the follow-up tech report which adds the authors Kaplan and Shafrir [7], the problem of *Mergeable Trees* is studied. This paper is concerned

with maintaining a dynamic collection of heaps, subject to operations which constitute the *dynamic tree* ADT: parent, root, nca, insert, link, cut, delete, and one additional operation, MERGE, which makes things interesting. In the MERGE operation, two nodes are specified, and the two root-to-node paths, which are each in sorted order due to the heap property, are merged. The idea of a *Mergeable Tree* ADT originated in an algorithm to compute the structure of a 2-maniford in R^3 (The original paper has a algorithm which is shown in [1] to take time $\Omega(\sqrt{n})$ per operataion) This ADT is a generalization of our Mergeable Dictionary, as if the heaps are restricted to be paths, both structures have identical functionality. They too obtain a $\mathcal{O}(\log n)$ amortized bound on the number of segments, using the same potential function as in our $\mathcal{O}(\log^2 n)$ presentation, albeit using link-cut trees [12] as the underlying $\mathcal{O}(\log n)$ structure due to their need to have the operations be performed on a tree paths in a heap rather than a totally ordered set, thus obtaining a $\mathcal{O}(\log^2 n)$ amortized bound on their operations. We conjecture that using the weighting scheme presented in this paper will allow the development of a Mergeable Tree with $\mathcal{O}(\log n)$ runtime for all operations. This would require the development of a weighted variant of link-cut trees that support weighted finger searches. In effect, link-cut trees extend $(2, 4)$ trees to allow operations on a tree topology, while biased skip lists allow sophisticated weighting operations. We would need a combination of both in order to achieve $\mathcal{O}(\log n)$ amortized time Mergable Trees. While we conjecture such a combination is possible, the details to be worked out are numerous.

1.2.3 Union-Split-Find

In the MIT thesis of Lai, supervised by Demaine, the problem of *Union-Split-Find* is studied [9]. This is proposed as a variant of the classic *Union-Find* data structure of [13]. Our Mergeable Dictionary ADT implements FIND as SEARCH which is a stronger operation than FIND (e.g. FIND can be implemented by returning the maxima of a set). They propose the $\mathcal{O}(\log^2 n)$ structure outlined above, and show that its amortized performance is $\Omega(\log^2 n)$. They conjecture (correctly) that there is a potential function that gives $\mathcal{O}(\log^2 n)$ runtime, but do not discover it, instead listing several potential functions which do not work. With thanks to Mihai Pătraşcu, they show that there is a lower bound of $\Omega(\log n)$ for this problem which follows from dynamic connectivity lower bounds [11]. This lower bound indicates our use of the stronger SEARCH instead of the weaker FIND comes at no additional asymptotic amortized cost. They conjecture (incorrectly) that this problem has a lower bound of $\Omega(\log^2 n)$ per operation; our $\mathcal{O}(\log n)$ result refutes this. Our results directly provide an amortized optimal $\mathcal{O}(\log n)$ time solution to their problem, while the best upper bound they could prove was $\mathcal{O}(n)$.

1.3 Future Work

This paper opens up several avenues for future work. First, as previously stated we believe that our approach can remove a $\log n$ factor from the runtimes of the Lempel-Ziv searching algorithm and of Mergeable Trees. Both of these results will require integrating our result into each of these different and complicated results. But, since in both cases the same potential function is used as in the $\mathcal{O}(\log^2 n)$ structure presentation below, we believe that our data structure provides the fundamental breakthrough needed to improve these results.

Secondly, for simplicity we do not consider the dynamic case: our data structure always stores a collection of sets that partitions the same totally ordered set. Extending our result to allow insertion and deletion will require adding additional complexity to our weighting scheme. Finally,

we note that the idea that arbitrarily interleaved dictionaries can be merged in $\mathcal{O}(\log n)$ amortized time is probably a surprising one, which at first glance probably appears to to impossible (which is supported by Lai's inability to get a $o(n)$ solution). While previous results had elements of the Mergeable Dictionary ADT, and a $\mathcal{O}(\log^2 n)$ solution to it, here is the first clear abstraction of it as a pure dictionary problem. The fact that the three results above were initially discovered independently of each other also speaks to the "buried" nature of the fundamental problem in the previous work. We hope that this result finds applications which were not considered because of the seeming impossibility at first glance of a $o(n)$ solution.

2 A Simple Heuristic for Merge and the $\mathcal{O}(\log^2 n)$ Amortized Bound

As mentioned previously, the main difficulty in designing a data structure for our problem with $o(n)$ worst-case time complexity lies in being able to perform the MERGE operation fast. This is confirmed by a lower bound of $\Omega(n)$ on the worst-case time complexity of merging two arbitrary sets [5]. We will describe a heuristic for the MERGE operation presented in [5] and used in previous work [7, 9], and show that the use of this heuristic yields $o(n)$ amortized bounds as a warm up.

Consider the MERGE(A, B) operation and a maximal subset in either set A or B such that all the elements of the other set are less than or greater than each element of the subset. We call this maximal subset a segment. We can view the MERGE(A, B) operation as gluing the appropriate segments of set A and B. Consider, for instance, the *Merge* algorithm of *Merge-Sort*, which could be used to implement the MERGE operation. The *Merge* algorithm linearly scans each segment until it locates its maximum element. The *segment merging heuristic* is based on that idea that there are more efficient methods of locating the maximum element of a set than a linear scan.

Next, we make the notion of segments slightly more precise, describe the heuristic, and describe the potential function which yields an upper bound of amortized $\mathcal{O}(\log^2 n)$ time.

2.1 The Segment Merging Heuristic

Define a *segment* of the MERGE(A, B) operation to be a maximal subset S of either set A or set B such that no element in $(A \cup B) \setminus S$ lies in the interval $[\min(S), \max(S)]$.

Each set in the collection is stored as a balanced search tree (i.e. 2-4 tree) with level links. The FIND, SEARCH, and SPLIT operations are implemented[2] in a standard way to run in $\mathcal{O}(\log n)$ worst-case time. The MERGE(A, B) operation is performed as follows:

We first locate the minimum and maximum element of each segment of the MERGE(A, B) operation using the SEARCH operation and the level links, then extract all these segments using the SPLIT operation, and finally we merge all the segments in the obvious way using a standard JOIN operation. Therefore since each operation takes $\mathcal{O}(\log n)$ worst-case time, the total running time is $\mathcal{O}(T \cdot \log n)$ where T is the number of segments.

We now analyze all the operations using the potential method [14], with respect to two parameters: n, the size of the universe, and m, the total number of all operations.

Let D_i represent the data structure after operation i, where D_0 is the initial data structure. Operation i has a cost of c_i and transforms D_{i-1} into D_i. We have a potential function $\Phi : \{D_i\} \to \mathbb{R}$ such that $\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i. The amortized cost of operation i, $\hat{c_i}$, with respect

[2] See [9] for a detailed description of this implementation.

to Φ is defined as $\hat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1})$. The total amortized cost of m operations will be

$$\sum_{i=1}^{m} \hat{c_i} = \sum_{i=1}^{m} (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^{m} c_i + \Phi(D_n) - \Phi(D_0) \geq \sum_{i=1}^{m} c_i$$

since $\Phi(D_n) \geq 0$ and $\Phi(D_0) = 0$. Thus, the amortized cost will give us an upper bound on the worst-case cost.

Next, we describe a potential function which yields an amortized bound of $\mathcal{O}(\log^2 n)$ on the running time. This potential function was essentially used in [6, 7] which are the only instances where a $o(n)$ solution has been presented.

2.2 The Potential Function

We need to define some terminology before describing the potential function. Let $\text{pos}_S(x)$ be the position of x in set S, or more formally $pos_S(x) = |\{y \in S \mid y \leq x\}|$. Then $g_S(k)$, the size of the k^{th} gap of set S, is the difference of positions between the element of position k and $k+1$ of set S in universe U. In other words, $g_S(k) = pos_U(x) - pos_U(y)$ where $pos_S(x) = k$ and $pos_S(y) = k+1$. For the boundary cases, let $g_S(0) = g_S(|S|) = 1$.

Recall that D_i is the data structure containing our dynamic collection of disjoint sets, $\mathcal{S}^{(i)} = \{S_1^{(i)}, S_2^{(i)}, \ldots\}$ after the i^{th} operation. Finally, let $\varphi(S) = \sum_{j=1}^{|S|-1} \log g_S(j)$. Then we define the potential after the i^{th} operation as follows:

$$\Phi(D_i) = \kappa_a \cdot \sum_{S \in \mathcal{S}^{(i)}} \varphi(S) \log n$$

where κ_a is a positive constant to be determined later.

Note that since the collection of sets consists of the n singleton sets, the data structure initially has 0 potential ($\Phi(D_0) = 0$). Furthermore, because any gap has size at least 1, the data structure always has non-negative potential ($\Phi(D_i) \geq 0,\ \forall i \geq 0$).

2.3 The Amortized $\mathcal{O}(\log^2 n)$ Bound

The FIND, SEARCH, and SPLIT operations have worst-case $\mathcal{O}(\log n)$ running times. The first two of these operations do not change the structure and therefore do not affect the potential. Observe that the SPLIT operation can only decrease the potential. Thus, the amortized cost of all three operations is $\mathcal{O}(\log n)$.

Now, suppose the i^{th} operation is MERGE(A, B) where A and B are sets in D_{i-1}. Assume w.l.o.g. that the minimum element in $A \cup B$ is an element of A. Let $I(A, B) = \{A_1, B_1, A_2, B_2, \ldots\}$ be the set of segments of operation MERGE(A, B), where $\max(A_i) < \min(A_j)$ and $\max(B_i) < \min(B_j)$ for $i < j$, and $\max(A_i) < \min(B_i) < \max(B_i) < \min(A_{i+1})$ for all i. As previously noted, the worst-case cost of the MERGE operation is $\mathcal{O}(|I(A, B)| \cdot \log n)$. Let a_i be the size of the gap between the maximum element of A_i and the minimum element of A_{i+1}, or more formally let $a_i = g_{A_i \cup A_{i+1}}(|A_i|)$. Define b_i similarly. Now, let

$$a_i' = g_{A_i \cup B_i}(|A_i|)$$
$$a_i'' = g_{B_i \cup A_{i+1}}(|B_i|)$$

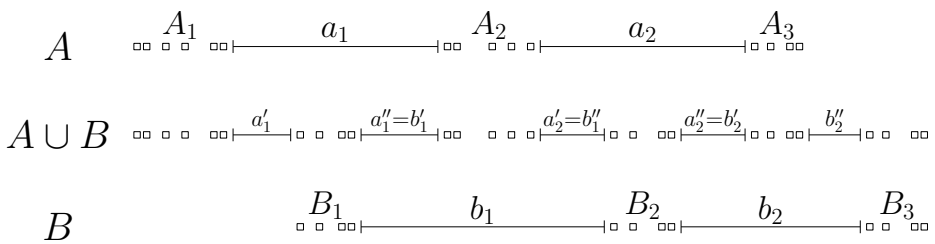

Figure 1: Gaps a_i, b_i, a'_i, b''_i, b'_i, and b''_i are defined with respect to the MERGE(A, B) operation.

and

$$b'_i = g_{B_i \cup A_{i+1}}(|B_i|)$$
$$b''_i = g_{A_{i+1} \cup B_{i+1}}(|A_{i+1}|)$$

Note that $a''_i = b'_i$ and $a'_i = b''_{i-1}$ (see Figure 1). During the analysis we will take into account whether $|I(A, B)|$ is odd or even. Let $\sigma = |I(A, B)| \bmod 2$, and $R = \lfloor (|I(A, B)| - 2)/2 \rfloor$. We are now ready to bound the amortized cost of the MERGE operation. We have

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

$$\begin{aligned}\Phi(D_{i-1}) = &\sum_{S \in \mathcal{S} \setminus \{A,B\}} \varphi(S)\kappa_a \log n \\ &+ \left(\sum_i \varphi(A_i) + \sum_i \varphi(B_i) \right) \kappa_a \log n \\ &+ \left(\sum_{i=1}^{R} (\log a_i + \log b_i) + \sigma \log a_{R+1} \right) \kappa_a \log n\end{aligned}$$

and

$$\begin{aligned}\Phi(D_i) = &\sum_{S \in \mathcal{S} \setminus \{A,B\}} \varphi(S)\kappa_a \log n \\ &+ \left(\sum_i \varphi(A_i) + \sum_i \varphi(B_i) + \log a'_1 \right) \kappa_a \log n \\ &+ \frac{1}{2} \left(\sum_{i=1}^{R} \log a''_i + \log b'_i + \log b''_i + \log a'_{i+1} \right) \kappa_a \log n \\ &+ \frac{1}{2} \left(\log a''_{R+1} + \log b'_{R+1} \right) \sigma \kappa_a \log n.\end{aligned}$$

This gives us

$$\Phi(D_i) - \Phi(D_{i-1}) = \left(\frac{1}{2} \left(\sum_{i=1}^{R} \log a''_i b'_i b''_i a'_{i+1} - 2 \log a_i b_i \right) + \log a'_1 \right) \kappa_a \log n$$

$$
\begin{aligned}
&+ \left(\frac{1}{2}\left(\log a''_{R+1} + \log b'_{R+1}\right) - \log a_{R+1}\right)\sigma\kappa_a \log n \\
&= \left(\frac{1}{2}\left(\sum_{i=1}^{R} \log a''_i b'_i b''_i a'_i - 2\log a_i b_i\right) + \frac{1}{2}\left(\log a'_1 + \log a'_{R+1}\right)\right)\kappa_a \log n \\
&+ \left(\frac{1}{2}\left(\log a''_{R+1} + \log b'_{R+1}\right) - \log a_{R+1}\right)\sigma\kappa_a \log n
\end{aligned}
$$

We have $a''_i, a'_i < a_i \leq n$ and similarly $b''_i, b'_i < b_i \leq n$. Also note that since $a'_i + a''_i \leq a_i$, we have $\log a'_i + \log a''_i \leq \log a'_i + \log(a_i - a'_i) \leq \log a_i/2 + \log a_i/2$. Similarly, $\log b'_i + \log b''_i \leq \log b_i/2 + \log b_i/2$.

$$
\begin{aligned}
\Phi(D_i) - \Phi(D_{i-1}) &\leq \frac{1}{2}\left(\sum_{i=1}^{R} \log a''_i b'_i b''_i a'_i - 2\log a_i b_i\right)\kappa_a \log n + \mathcal{O}(\log^2 n) \\
&= \frac{1}{2}\left(\sum_{i=1}^{R} \log \frac{a'_i \cdot a''_i \cdot b'_i \cdot b''_i}{a_i \cdot a_i \cdot b_i \cdot b_i}\right)\kappa_a \log n + \mathcal{O}(\log^2 n) \\
&\leq \frac{1}{2}\left(\sum_{i=1}^{R} \log \frac{a_i/2 \cdot a_i/2 \cdot b_i/2 \cdot b_i/2}{a_i \cdot a_i \cdot b_i \cdot b_i}\right)\kappa_a \log n + \mathcal{O}(\log^2 n) \\
&= \frac{1}{2}\left(\sum_{i=1}^{R} \log \frac{1}{16}\right)\kappa_a \log n + \mathcal{O}(\log^2 n) \\
&= -\kappa_b \cdot I(A,B)\kappa_a \log n + \mathcal{O}(\log^2 n).
\end{aligned}
$$

Recall that the worst-case cost of the MERGE operation, c_i, is $\mathcal{O}(|I(A,B)| \log n)$. Let κ_c be a constant such that $c_i \leq \kappa_c \cdot |I(A,B)| \log n$. Then the above bound yields

$$
\begin{aligned}
\hat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&\leq c_i - \kappa_b \cdot I(A,B)\kappa_a \log n + \mathcal{O}(\log^2 n) \\
&\leq \kappa_c \cdot I(A,B) \log n - \kappa_b \cdot I(A,B)\kappa_a \log n + \mathcal{O}(\log^2 n) \\
&= \mathcal{O}(\log^2 n) \qquad\qquad (\text{set } \kappa_a = \kappa_c/\kappa_b).
\end{aligned}
$$

Thus, the amortized cost of the MERGE operation is $\mathcal{O}(\log^2 n)$. Combined with the arguments before, this gives us the following theorem.

Theorem 1. *The Mergeable Dictionary problem can be solved such that a sequence of m* FIND, SEARCH, MERGE, *and* SPLIT *operations can be executed in $\mathcal{O}(m \log^2 n)$ worst-case time.*

In order to obtain a data structure with an amortized running time of $o(\log^2 n)$ per operation, we certainly need a new potential function. To see this, observe the case when we have two singleton sets A and B with elements $x \in A$ and $y \in B$ such that $\log(pos_U(x) - pos_U(y)) = \Omega(\log n)$. If we use the potential function defined in Section 2.2, the potential increase alone as a result of a MERGE(A,B) operation is $\Omega(\log^2 n)$. We want to eliminate the extra $\log n$ factor in the potential function but this implies we need to be able to join segments in amortized $\mathcal{O}(1)$ time.

We ultimately want a data structure such that each operation except MERGE can be performed in worst-case $\mathcal{O}(\log n)$ time, and MERGE can be performed in worst-case

$$\mathcal{O}\left(\log n + \sum_i F(A_i) + \sum_j F(B_j)\right)$$

time where A_i and B_j are segments involved in the operation, $F(A_i)$ and $F(B_j)$ denote the time it takes to process segments A_i and B_j respectively, and $\sum_i F(A_i) + \sum_j F(B_j)$ is no more than the decrease in potential.

In the next section, we describe biased skip lists, the underlying data structure we will be using in our data structure.

3 Biased Skip Lists with Extended Operations

Biased skip lists are a variant of skip lists [10] which we assume the reader is familiar with. Biased skip lists as described in [2] are missing some operations which will be vital in the implementation of our data structure. Therefore, in order to be able to design a highly tuned MERGE operation, we will extend biased skip lists.

First, we describe essential biased skip list details. The reader is referred to [2] for further details on biased skip lists.

3.1 Biased Skip Lists

We will first cover basic definitions followed by the three key invariants of the structure.

Definitions A biased skip list (BSL) S stores an ordered set X where each element $x \in X$ corresponds to a node[3] $x \in S$ with *weight* $w(x)$, which is user-defined, and integral *height* $h(x)$, which is initially computed from the weight of the node. For our purposes, we will assume that the weights are bounded from below by 1 and bounded from above by a polynomial in n.

Each node $x \in S$ is represented by an array of length $h(x) + 1$ called the *tower* of node x. The *level-j predecessor*, $L_j(x)$, of x is the largest node k in S such that $k < x$ and $h(k) \geq j$. The *level-j successor*, $R_j(x)$, is defined symmetrically. The j^{th} element of the tower of node x, contains pointers to the j^{th} elements of towers of node $L_j(x)$ and node $R_j(x)$ with the exception of towers of adjacent nodes where pointers between any pair of adjacent nodes x and y on level $\min(h(x), h(y)) - 1$ are nil and the pointers below this level are undefined. Node levels progress from top to bottom. Two distinct elements x and y are called *consecutive* if and only if they linked together in S; or equivalently if and only if for all $x < z < y$, $h(z) < \min(h(x), h(y))$. A *plateau* is a maximal set of consecutive nodes of the same height. The *rank* of a node x is defined as $r(x) = \lfloor \log_a w(x) \rfloor$ where a is a constant to be specified later. For our purposes, we will set $a = 2$.

Additionally, let $\text{pred}_X(x)$ be the predecessor of x in set X, and let $\text{succ}_X(x)$ be the successor of x in set X. Let $H(X) = \max_{x \in X} h(x)$. Let $S[\hookleftarrow j] = \{x \in S \mid x \leq j\}$ and $S[j\rightarrow] = \{x \in S \mid x > j\}$. Let $W(S) = \sum_{x \in S} w(x)$. Also let $W_{[i,j]}(S) = \sum_{x \in S; i \leq x \leq j} w(x)$.

For convenience, we imagine sentinel nodes $-\infty$ and $+\infty$ of height $H(S)$ at the beginning and end of biased skip list S. These sentinels are not actually stored or maintained.

[3]We will use the terms "element", "node", "key", and "item" interchangeably; the context clarifies any ambiguity.

The *left profile* of x in a biased skip list S is defined as $\{L_j(x) \mid h(L_j(x)) = j\}$. Similarly the *right profile* of x in a biased skip list S is defined as $\{R_j(x) \mid h(R_j(x)) = j\}$. The *profile* of node x in a biased skip list S is the union of its left profile and right profile.

The *left cover* of a biased skip list S is defined as $\{\min(S)\} \cup \{R_j(\min(S)) \mid h(R_j(\min(S))) = j,\ j > h(\min(S))\}$. Similarly, the *right cover* of a biased skip list S is defined as $\{\max(S)\} \cup \{L_j(\max(S)) \mid h(L_j(\max(S))) = j,\ j > h(\max(S))\}$. The *cover* of biased skip list S is the union of its left cover and right cover.

Invariants The three invariants of biased skip lists are listed below. Note that a and b can be suitable constants satisfying the definition of (a, b)-biased skip lists. For our purposes it is sufficient to set $a = 2, b = 6$.

Definition 2. *For any a and b such that $1 < a \leq \lfloor \frac{b}{3} \rfloor$, an (a, b)-biased skip list is a biased skip list with the following properties:*

(**I0**) *Each item x has height $h(x) \geq r(x)$.*

(**I1**) *There are never more than b consecutive items of any height.*

(**I2**) *For each node x and for all i such that $r(x) < i \leq h(x)$, there are at least a nodes of height $i - 1$ between x and any consecutive node of height at least i.*

In the remainder of the paper, we will refer to $(2, 6)$-biased skip lists simply as biased skip lists.

3.2 Operations

We now describe the original biased skip list operations we will be using in our data structure.

- $p \leftarrow$ BSL-Search(S, i): Performs a standard search in biased skip list S using search key i. This operation runs in worst-case $\mathcal{O}(\log n)$ time.
- $p \leftarrow$ BSL-FSearch(i, j): Starting from a given finger[4] to a node i in some biased skip list S perform a predecessor search in S using j as the search key. This operation runs in

$$\mathcal{O}\left(1 + \log \frac{W_{[i,\mathrm{succ}_S(j)]}(S)}{\min(w(i), w(\mathrm{pred}_S(j)), w(\mathrm{succ}_S(j)))}\right)$$

  worst case time.
- $(A, B) \leftarrow$ BSL-Split(S, i): Splits the biased skip list S at i into two biased skip lists A and B storing sets $\{x \in S \mid x \leq i\}$ and $\{x \in S \mid x > i\}$ respectively, and returns an ordered pair of handles to A and B. This operation runs in worst-case $\mathcal{O}(\log n)$ time.
- BSL-Rew(S, i, w): Changes the weight of node $i \in S$ to w. This operation runs in worst-case $\mathcal{O}(\log n)$ time.

[4]This operation is originally described with three arguments in [2] as $FingerSearch(X, i, j)$. However, note that X is a redundant argument here as we already have a pointer to an element of X, namely, to i.

3.3 Extended Operations

Biased skip lists support finger searches, however we need to extend biased skip lists to also support a finger split, a finger join, and a finger reweight operation.

Finger Split Given a pointer to node $f \in S$, BSL-FSplit(f) splits biased skip list S into two biased skip lists, A and B, storing sets $\{x \in S \mid x \leq f\}$ and $\{x \in S \mid x > f\}$ respectively, and returns an ordered pair of handles to A and B.

$(A, B) \leftarrow$ BSL-FSplit(f):

1. Disconnect the pointers between the each node in the right profile of node f and the left profile of node $f' = \text{succ}_S(f)$, effectively splitting S and forming A and B. More precisely, disconnect pointers between the j^{th} level of node $R_j(f)$ and $L_j(f')$ where $j \geq \min(h(f), h(f'))$. Pointers below this level are already null.

2. Restore (**I2**) in A. We will process the nodes of the right cover of A after Step 1 in the order of increasing height. Denote the current node being processed by u. Let h' be the height of the node which was most recently processed.

   (a) If $u = f$, then set $h' = h(u)$ and demote the height of u to $r(u)$.

   (b) If (**I2**) is not violated at u, then stop if $h(u) > \min(H(S[\hookleftarrow f]), H(S[f\rightarrow]))$, and set $h' = h(u)$ and iterate with the next node, $L_{h(u)+1}(A)$, otherwise[5].

   (c) If (**I2**) is violated at node u, then demote the height of u to $\max(r(u), h')$. Set h' to the height of u before the demotion.

   - If a demotion causes an (**I1**) violation at the new height of u by creating a plateau of $b' > b$ nodes, then promote the height of the median of these nodes by 1, and iterate at the level above to check for percolating (**I1**) violations.
   - Once all the (**I1**) violations are fixed, iterate with the next node, $L_{h'+1}(A)$, to fix the next potential (**I2**) violation.

   Restore (**I2**) in B essentially symmetrically.

Correctness All the pointers in S connecting any node in A and any node in B are precisely those described in Step 1. Therefore Step 1 splits the nodes of A and B correctly.

We need to ensure that all the invariants are preserved. When we perform demotions in Step 2 we make sure that we do not demote the height of any node lower than its rank. Therefore (**I0**) is preserved. Note that removing predecessors or successors cannot cause an (**I1**) violation.

Observe that in A, (**I2**) can only be violated at the nodes in the right cover of A after Step 1, once per level. A symmetric argument holds for (**I2**) violations in B. When we demote a node u this fixes the (**I2**) violation at that node because the height of the node is either demoted to $r(u)$ which by definition implies the violation is fixed, or to h'. If the height of node u is demoted to h', this implies that there was another node u' with height h', consecutive to u, before the BSL-FSplit operation. The only way we change the height of any node between node u and node u'

[5]Note that $L_{h(u)+1}(A)$ can be computed in $\mathcal{O}(b) = \mathcal{O}(1)$ time due to (**I1**); and $\min(H(S[\hookleftarrow f]), H(S[f\rightarrow]))$ can be computed during Step 1.

during BSL-FSplit(f) is an (**I1**) promotion, which, cannot cause an (**I2**) violation. Note that there cannot be an (**I2**) violation at these nodes since they are not in the right cover of A after Step 1. We assume there were no (**I2**) violations before the BSL-FSplit operation. Then it holds that there can be no (**I2**) violations at node u at any level less than or equal to h'. Therefore, demoting u to level h' fixes any (**I2**) violations at this node.

Any of the demotions may cause an (**I1**) violation which are also fixed by Step 2. Because we promote the median, this cannot cause an (**I2**) violation since $\lfloor b'/2 \rfloor \geq \lfloor b'/3 \rfloor \geq a$. Also note that the (**I1**) promotions caused by the demotion of a node never percolate higher than the height of the node before the demotion. Thus, nodes on the right cover of A after Step 1 that have not yet been processed during Step 2 are not removed from the right cover due to an (**I1**) promotion.

Observe that if a node u in the right cover of A after Step 1 has height greater than $\min(H(S[\hookleftarrow f]), H(S[f\rightarrow]))$ and there is no (**I2**) violation at this node, then no other node of greater height in A can have an (**I2**) violation. Symmetric arguments again apply for B. Thus, by induction, iterating Step 2 fixes all (**I2**) violations.

Finger Join Given pointers to ℓ and r, the maximum and minimum nodes of two distinct biased skip lists A and B respectively, BSL-FJoin(ℓ, r) returns a new biased skip list C containing all elements of A and B assuming $\ell < r$. A and B are destroyed in the process.

$S \leftarrow$ BSL-FJoin(ℓ, r)

1. Connect pointers between each node in the right profile of node ℓ and the left profile of node r, effectively joining A and B, and forming C. More precisely, create pointers between the j^{th} level of node $R_j(\ell)$ and $L_j(r)$ where $j \geq \min(h(\ell), h(r))$. Pointers below this level need to already be null.

2. Restore (**I1**) in C. For $j = \max(h(\ell), h(r))$ up through $\min(H(A), H(B))$, if there is an (**I1**) violation at level j caused by a plateau of $b' > b$ nodes, then promote the height of the median of these nodes by 1, and iterate at the next level. For $j = \min(H(A), H(B)) + 1$ up through $\max(H(A), H(B))$, if there is no (**I1**) violation at level j, then stop[6]. Otherwise promote the median node of the plateau of $b' > b$ nodes causing the violation to restore (**I1**) and iterate at the next level.

Correctness Joining A and B only affects the right profile of ℓ and the left profile of r. Therefore, Step 1 connects the two biased skip lists A and B and forms C correctly. Joining two biased skip lists cannot create any (**I2**) violations assuming there were no such violations before the operation; and fixing (**I1**) violations cannot create (**I2**) violations since $\lfloor b'/2 \rfloor \geq \lfloor b'/3 \rfloor \geq a$. Therefore, (**I2**) is preserved. Note that after Step 1, any level in the range $[\max(h(\ell), h(r)), \max(H(A), H(B))]$ could have at most $2b + 1$ (b from A, b from B, and 1 due to a promotion from the level below) consecutive nodes of the same height; which is fixed in Step 2. By induction, iterating Step 2 fixes all (**I1**) violations.

Finger Reweight Given a pointer to a node $f \in S$, changes its weight to w while preserving invariants (**I0**), (**I1**), and (**I2**) of the biased skip list containing f.

[6] $\min(H(A), H(B))$ can be computed during Step 1.

BSL-FRew(f, w) :

1. Let $r'(f)$ be the new rank of f. If $r'(f) = r(f)$, then stop.
2. If $r'(f) > r(f)$ and $h(f) \geq r'(f)$, then stop.
3. If $r'(f) > r(f)$ and $h(f) < r'(f)$, then promote the height of f to $r'(f)$. Restore (**I2**) as in Step 2 of BSL-FSplit but start from the first node in the left profile of f that has height greater than $h(f)$; and symmetrically from the first node in the right profile of f that has height greater than $h(f)$. Then, restore (**I1**) as in Step 2 of BSL-FJoin but start from level $r'(f)$.
4. If $r'(f) < r(f)$, then demote the height of f to $r'(f)$. Restore (**I1**) as in Step 2 of BSL-FJoin but starting from $r'(f)$. Then, restore (**I2**) as in Step 2 of BSL-FSplit, but starting from the first node in the left profile of f that has height greater than $h(f)$; and symmetrically from the first node in the right profile of f that has height greater than $h(f)$.

Correctness If the rank of node f does not change, there are no structural changes to the biased skip list and therefore Step 1 is correct.

If the rank of f increases but it is still less than its height, then (**I0**) is preserved. By not changing the height of the node we ensure that (**I1**) and (**I2**) are preserved as well. Therefore, Step 2 is also correct.

If the rank of a node f becomes greater than its height, then (**I0**) is violated and we promote f to its new rank to fix the violation. Observe that this promotion can cause (**I2**) to be violated at the nodes in the left profile of f that have height greater than the old height of f; and symmetrically at the nodes in the right profile of f that have height greater than the old height of f, once per level. Step 3, by the correctness of Step 2 of BSL-FSplit, fixes all (**I2**) violations. The promotion can also cause (**I1**) to be violated at level $r'(f)$. Step 3, by the correctness of Step 2 of BSL-FJoin, fixes all (**I1**) violations. Therefore, Step 3 is correct.

If the rank of a node f decreases, we demote f to its new rank so (**I2**) cannot be violated at this node. Observe that this demotion can cause (**I1**) to be violated at level $r'(f)$. Step 4, by the correctness of Step 2 of BSL-FJoin, fixes all (**I1**) violations. The demotion can also cause (**I2**) to be violated at the nodes in the left profile of f that have height greater than the old height of f; and symmetrically at the nodes in the right profile of f that have height greater than the old height of f, once per level. Step 4, by the correctness of Step 2 of BSL-FSplit, fixes all (**I2**) violations. Therefore, Step 4 is correct.

Since steps 1-4 exhaust all possible scenarios, all the invariants are preserved and the BSL-FRew operation is correct.

Before moving on, we need to analyze the time complexity of these new operations.

3.4 The Analysis of Extended Operations

We now analyze the time complexity of the extended operations described above using the potential method.

The extended operations, BSL-FSplit, BSL-FJoin, and BSL-FRew, are executed on a set of biased skip lists. Let L_k be the set of biased skip lists after the k^{th} operation, where L_0 is the initial set of biased skip lists we are given.

Let $\mathcal{P}_k$ be the set of plateaus which contain an element in the cover of some biased skip list in L_k. For a plateau p, let $|p|$ be the number of nodes contained in plateau p. We define the following sets of plateaus. Let $S^k_{[x,y]} = \{p \in \mathcal{P}_k \,|\, x \leq |p| \leq y\}$ and $S^k_{(x,y)} = \{p \in \mathcal{P}_k \,|\, x < |p| < y\}$. We can now define a potential function as follows.

$$\Phi_e(L_k) = \kappa_g(8|S^k_{(0,a)}| + 4|S^k_{[a,a]}| + |S^k_{[b,b]}| + 3|S^k_{(b,2b)}| + 5|S^k_{[2b,2b+1]}|).$$

Observe that $\Phi_e(L_k) \geq 0$ for any k. For each extended operation, we will use this potential function to prove upper bounds on the amortized time complexity of the operation as well as worst-case time complexity of a sequence of operations.

Lemma 3. *The* $(A, B) \leftarrow$ BSL-FSplit(f) *operation, where* $f \in S$*, has an amortized time complexity of*

$$\mathcal{O}(\min(H(A'), H(B')) - \min(r(\max(A')), r(\min(B'))) + 1)$$

where $A' = S[\leftarrow f]$ *and* $B' = S[f \rightarrow]$.

Proof. Let BSL-FSplit(f) be the k^{th} operation. Step 1 takes $\mathcal{O}(\min(H(A'), H(B')) - \min(h(\max(A')), h(\min(B'))) + 1)$ time. Step 2 takes constant time at each level from level $\min(r(\max(A')), r(\min(B')))$ to level $\min(H(A'), H(B'))$. We will show that the time spent by Step 2 on levels greater than $\min(H(A'), H(B'))$ is essentially negligible by showing that it equals the decrease in potential at these levels.

We now analyze the contribution of Step 1 and Step 2 to the potential change, $\Phi_e(L_k) - \Phi_e(L_{k-1})$.

Step 1 Due to the split, the plateaus on the cover of S becomes plateaus on the cover of A and B. Additionally, the plateaus on levels less than or equal to $\min(H(A'), H(B'))$ on the right cover of A and left cover of B are added to $\mathcal{P}_k$. Therefore, the contribution of Step 1 to the potential change is at most $\mathcal{O}(\min(H(A'), H(B')) - \min(h(\max(A')), h(\min(B'))) + 1)$.

Step 2 We now look at the contribution of Step 2 to the potential change, $\Phi_e(L_k) - \Phi_e(L_{k-1})$. Note that at any level less than or equal to $\min(H(A'), H(B'))$, the potential increase can be at most a constant. Therefore, the maximum contribution of Step 2 to the potential change in these levels is $\mathcal{O}(\min(H(A'), H(B')) - \min(r(\max(A')), r(\min(B'))))$. Next, we bound the contribution of Step 2 to the potential change in levels greater than $\min(H(A'), H(B'))$.

Demotions Consider a demotion operation to restore (**I2**) at some node x. This demotion could cause a change in potential in two ways.

First, the plateau p' which was causing the (**I2**) violation could have had less than a nodes, and now has more.

Note that since we demote node x to the level of p', there must be a plateau of at least a nodes (due to (**I2**)) on the other side of node x. Therefore, p' will have at least $a + 1$ nodes. Also, the plateau on the other side of node x cannot have more than b nodes. Therefore, p' will have at most $b + a$ nodes. This implies that p' can only have between $a + 1$ and $b + a$ nodes. If p' has between $a + 1$ and $b - 1$ nodes, the contribution of this part to the potential change is $-8\kappa_g$. If p' has b nodes, the contribution of this part to the potential change is $-8\kappa_g + \kappa_g = -7\kappa_g$. If p' has between

$b+1$ and $b+a$ nodes, the contribution of this part to the potential change is $-8\kappa_g + 3\kappa_g = -5\kappa_g$. Therefore, the maximum contribution of this part to the potential change is $-5\kappa_g$.

Note that p' might not exist (have zero nodes). In this case, the maximum contribution of this part to the potential change is $3\kappa_g$. However, this case is only possible if p' has height less than or equal to $\min(H(A'), H(B'))$.

Second, the demotion of x could cause the plateau p'', which x was a part of before the demotion, to have less nodes. If p'' had $2b$ nodes or more and now has between $b+1$ and $2b-1$ nodes, the contribution of this part to the potential change is $-5\kappa_g + 3\kappa_g = -2\kappa_g$. If p'' had between $b+1$ and $2b-1$ nodes and now has b nodes, the contribution of this part to the potential change is $-3\kappa_g + \kappa_g = -2\kappa_g$. If p'' had b nodes and now has between $a+1$ and $b-1$ nodes, the contribution of this part to the potential change is $-\kappa_g$. If p'' had between $a+1$ and $b-1$ nodes and now has a nodes, the contribution of this part to the potential change is $4\kappa_g$. If p'' had a nodes and now has between 1 and $a-1$ nodes, the contribution of this part to the potential change is $-4\kappa_g + 8\kappa_g = 4\kappa_g$. Additionally, if p'' had between 1 and $a-1$ nodes and now has zero nodes, the contribution of this part to the potential change is $-8\kappa_g$.

Therefore, combining the first and second part, the maximum contribution of a demotion to the potential change is $7\kappa_g$ on levels less than or equal to $\min(H(A'), H(B'))$, and $-\kappa_g$ on levels greater.

Note that the demotion of x could cause a high number of plateaus which did not have any elements in the cover of their biased skip list to now have an element in the cover and thus enter $\mathcal{P}_k$. In order for this case to occur, the height of x must be demoted at least two levels. By the description of Step 2, this implies there were no nodes of height $h(x)-1$ in the biased skip list containing x after Step 1. Since S has no (**I2**) violations before Step 1, this implies there must be nodes of height $h(x)-1$ in the other biased skip list. Therefore, this case is only possible in levels less than or equal to $\min(H(A'), H(B'))$.

Promotions Consider a promotion operation at a node x to restore an (**I1**) violation on the plateau p' node x is on. This promotion could cause a change in potential in two ways.

First, the plateau p' could have had between $b+1$ and $2b+1$ nodes, and now has less. If p' had between $b+1$ and $2b-1$ nodes, then the promotion of node x splits p' into two plateaus. Only one of these plateaus remain in the cover of the biased skip list unless nodes of p' were at the highest level of the biased skip list. The plateau remaining in the cover must have size greater than a and less than b. Therefore, p' will now have between $a+1$ and $b-1$ nodes, and the contribution of this part to the potential change is $-3\kappa_g$. In the special case of both plateaus remaining in the cover, then both of them will have between $a+1$ and $b-1$ nodes, and the contribution of this part to the potential change is $-3\kappa_g$. If p' has $2b$ or more nodes, then the promotion of x splits p' into two plateaus. Only one of these plateaus remain in the cover of the biased skip list unless nodes of p' were at the highest level of the biased skip list. The plateau remaining in the cover must have size either $b-1$ or b. If it has $b-1$ nodes, the contribution of this part to the potential change is $-5\kappa_g$. If it has b nodes, the contribution of this part to the potential change is $-5\kappa_g + \kappa_g = -4\kappa_g$. In the special case of both plateaus remaining in the cover, then either one of them has $b-1$ nodes, and the other one has b nodes, and the contribution of this part to the potential change is $-5\kappa_g + \kappa_g = -4\kappa_g$; or both of them have b nodes, and the contribution of this part to the potential change is $-5\kappa_g + 2\kappa_g = -3\kappa_g$.

Second, the promotion of x could cause the plateau p'', which x becomes a part of after the

promotion, to have more nodes. If p'' had more than $2b-1$ nodes, note that it must have had at most $2b$ nodes. Promotion of x increases the size of p'' to $2b+1$. Thus, it does not change its set and the contribution of this part to the potential change is 0. Note that, in general, if the promotion does not change the set of p'', then the contribution of this part to the potential change is 0. If p'' had $2b-1$ nodes and now has $2b$ nodes, the contribution of this part to the potential change is $-3\kappa_g + 5\kappa_g = 2\kappa_g$. If p'' had b nodes and now has $b+1$ nodes, the contribution of this part to the potential change is $-\kappa_g + 3\kappa_g = 2\kappa_g$. If p'' had $b-1$ nodes and now has b nodes, the contribution of this part to the potential change is κ_g. If p'' had a nodes and now has $a+1$ nodes, the contribution of this part to the potential change is $-4\kappa_g$. If p'' had $a-1$ nodes and now has a nodes, the contribution of this part to the potential change is $-8\kappa_g + 4\kappa_g = -4\kappa_g$. Additionally, if p'' had zero nodes and now has 1 node, the contribution of this part to the potential change is $8\kappa_g$. However, this can only happen if an earlier demotion caused the only node of p'' to be demoted. Therefore, p'' first has 1 node; then a demotion causes p'' to have 0 nodes; and then a promotion associated with that demotion causes p'' to have 1 node again. The effect on the potential change is zero.

Therefore, combining the first and second part, the maximum contribution of a promotion to the potential change is $-\kappa_g$.

Assume the worst-case running time of any single demotion or promotion operation is bounded by a constant κ_h. Let $\hat{c_k}$ and c_k respectively be the amortized and worst-case running time of BSL-FSplit(f) and let v_k be the number of violations that are restored during Step 2 above level $\min(H(A'), H(B'))$. Then we have $c_k \leq \mathcal{O}(\min(H(A'), H(B')) - \min(r(\max(A')), r(\min(B'))) + 1) + \kappa_h \cdot v_k$. Combining the bounds on the maximum contribution of Step 1 and Step 2 to the potential change and setting $\kappa_g = \kappa_h$, we have $\Phi_e(L_k) - \Phi_e(L_{k-1}) \leq -v_k\kappa_h + \mathcal{O}(\min(H(A'), H(B')) - \min(r(\max(A')), r(\min(B'))))$. This yields

$$\begin{aligned}
\hat{c_k} &= c_k + \Phi_e(L_k) - \Phi_e(L_{k-1}) \\
&\leq \kappa_h v_k + \Phi_e(L_k) - \Phi_e(L_{k-1}) + \mathcal{O}(\min(H(A'), H(B')) - \min(r(\max(A')), r(\min(B'))) + 1) \\
&\leq \kappa_h v_k - \kappa_h v_k + \mathcal{O}(\min(H(A'), H(B')) - \min(r(\max(A')), r(\min(B'))) + 1) \\
&= \mathcal{O}(\min(H(A'), H(B')) - \min(r(\max(A')), r(\min(B'))) + 1).
\end{aligned}$$

This completes the proof. □

Lemma 4. *Given a set $L_0 = \{\Lambda_1, \Lambda_2, \ldots, \Lambda_m\}$ of biased skip lists, a sequence of $(S_k, T_k) \leftarrow$ BSL-FSplit(f_k) operations for $1 \leq k \leq t$ where $f_k \in U_k$ and $U_k \in L_{k-1}$ can be executed in worst-case time*

$$\mathcal{O}\left(\sum_{k=1}^{t}(\min(H(S'_k), H(T'_k)) - \min(r(\max(S'_k)), r(\min(T'_k))) + 1)\right) +$$

$$\mathcal{O}\left(\sum_{i=1}^{m}(H(\Lambda_i) - \min(h(\min(\Lambda_i)), h(\max(\Lambda_i))) + 1)\right)$$

where $S'_k = U_k[\leftarrow f_k]$ and $T'_k = U_k[f_k \rightarrow]$.

Proof. Let $\hat{c_k}$ and c_k respectively be the amortized and worst-case running time of BSL-FSplit(f_k). Note that $\Phi_e(L_0) = \mathcal{O}(|\mathcal{P}_0|) = \mathcal{O}\left(\sum_{i=1}^{m}(H(\Lambda_i) - \min(h(\min(\Lambda_i)), h(\max(\Lambda_i))) + 1)\right)$. Then we

have

$$\begin{aligned}
\sum_{k=1}^{t} \hat{c_k} &= \sum_{k=1}^{t} (c_k + \Phi_e(L_k) - \Phi_e(L_{k-1})) \\
\sum_{k=1}^{t} \hat{c_k} &= \sum_{k=1}^{t} c_k + \Phi_e(L_t) - \Phi_e(L_0) \\
\sum_{k=1}^{t} c_k &= \sum_{k=1}^{t} \hat{c_k} - \Phi_e(L_t) + \Phi_e(L_0) \\
\sum_{k=1}^{t} c_k &\leq \sum_{k=1}^{t} \hat{c_k} + \Phi_e(L_0) \\
&= \mathcal{O}\left(\sum_{k=1}^{t} (\min(H(S'_k), H(T'_k)) - \min(r(\max(S'_k)), r(\min(T'_k))) + 1)\right) \qquad \text{By Lemma 3} \\
&\quad + \Phi_e(L_0) \\
&= \mathcal{O}\left(\sum_{k=1}^{t} (\min(H(S'_k), H(T'_k)) - \min(r(\max(S'_k)), r(\min(T'_k))) + 1)\right) \\
&\quad + \mathcal{O}\left(\sum_{i=1}^{m} (H(\Lambda_i) - \min(h(\min(\Lambda_i)), h(\max(\Lambda_i))) + 1)\right).
\end{aligned}$$

□

Lemma 5. *The $S \leftarrow$ BSL-FJoin(ℓ, r) operation, where $\ell \in A$, $r \in B$, has an amortized time complexity of*

$$\mathcal{O}\left(\min(H(A), H(B)) - \min(h(\max(A)), h(\min(B))) + 1\right).$$

Proof. Let BSL-FJoin(ℓ, r) be the k^{th} operation. Step 1 takes $\mathcal{O}(\min(H(A), H(B)) - \min(h(\max(A)), h(\min(B))) + 1)$ time. Step 2 takes constant time at each level from level $\min(h(\max(A)), h(\min(B)))$ to level $\min(H(A), H(B))$. We will show that the time spent by Step 2 on levels greater than $\min(H(A), H(B))$ is essentially negligible by showing that it equals the decrease in potential at these levels.

We now analyze the contribution of Step 1 and Step 2 to the potential change, $\Phi_e(L_k) - \Phi_e(L_{k-1})$.

Step 1 Due to the join, the plateaus on levels greater than $\min(H(A), H(B))$ as well as the plateaus on left cover of A and right cover of B becomes plateaus on the cover of S. The plateaus on levels less than or equals to $\min(H(A), H(B))$ on the right cover of A and on the left cover of B are not on the cover of S and thus are not added to $\mathcal{P}_k$.

The only way Step 1 can increase the potential is if all the plateaus on the covers of A and B becomes plateaus on the cover of S and the plateaus containing nodes $\max(A)$ and $\min(B)$ merge and form a larger plateau with more weight with respect to the potential function. Therefore, the contribution of Step 1 to the potential change is at most $3\kappa_g$.

Step 2 Note that Step 2 only involves promotions. Any promotion on level less than or equal to $\min(H(A), H(B))$ will increase the potential by at most a constant and total contribution to the potential change is bounded by $\mathcal{O}(\min(H(A), H(B)) - \min(h(\max(A)), h(\min(B))) + 1)$. We will focus on promotions which occur on higher levels.

Consider a promotion operation at a node x to restore an (**I1**) violation on the plateau p' node x is on where p' is on a level greater than $\min(H(A), H(B))$. The promotion could cause a change in potential in two ways.

First, the plateau p' could have had $b+1$ nodes, then the promotion of node x splits p' into two plateaus with between $a+1$ and $b-1$ nodes each. Then, the contribution of this part to the potential change is $-3\kappa_g$.

Second, the promotion of x could cause the plateau p'', which x becomes a part of after the promotion, to have more nodes. Note that due to (**I1**), p'' could not have had more than b nodes. If p'' had b nodes and now has $b+1$ nodes, the contribution of this part to the potential change is $-\kappa_g + 3\kappa_g = 2\kappa_g$. If p'' had less than b nodes and now has at most b nodes, the potential could increase by at most a constant, but the promotion will not cause an (**I1**) violation at p'', and Step 2 will terminate.

Therefore, combining the first and second part, the maximum contribution of a promotion to the potential change is $-\kappa_g$ except possibly for the last promotion.

Assume the worst-case running time of any single promotion operation is bounded by a constant κ_j. Let $\hat{c_k}$ and c_k respectively be the amortized and worst-case running time of BSL-FJoin(ℓ, r) and let v_k be the number of violations that are restored during Step 2 above level $\min(H(A), H(B))$. Then we have $c_k \leq \mathcal{O}(\min(H(A), H(B)) - \min(h(\max(A)), h(\min(B))) + 1) + \kappa_j \cdot v_k$. Combining the bounds on the maximum contribution of Step 1 and Step 2 to the potential change and setting $\kappa_g = \kappa_j$, we have $\Phi_e(L_k) - \Phi_e(L_{k-1}) \leq -v_k\kappa_j + \mathcal{O}(\min(H(A), H(B)) - \min(h(\max(A)), h(\min(B))) + 1)$. This yields

$$\begin{aligned}
\hat{c_k} &= c_k + \Phi_e(L_k) - \Phi_e(L_{k-1}) \\
&\leq \kappa_j v_k + \Phi_e(L_k) - \Phi_e(L_{k-1}) + \mathcal{O}(\min(H(A), H(B)) - \min(h(\max(A)), h(\min(B))) + 1) \\
&\leq \kappa_j v_k - \kappa_j v_k + \mathcal{O}(\min(H(A), H(B)) - \min(h(\max(A)), h(\min(B))) + 1) \\
&= \mathcal{O}(\min(H(A), H(B)) - \min(h(\max(A)), h(\min(B))) + 1).
\end{aligned}$$

This completes the proof. □

Lemma 6. *Given a set $L_0 = \{\Lambda_1, \Lambda_2, \ldots, \Lambda_m\}$ of biased skip lists, a sequence of $U_k \leftarrow$ BSL-FJoin(ℓ_k, r_k) operations for $1 \leq k \leq t$ where $\ell_k \in S_k$, $r_k \in T_k$ and $S_k, T_k \in L_{k-1}$ can be executed in worst-case time*

$$\mathcal{O}\left(\sum_{k=1}^{t}(\min(H(S_k), H(T_k)) - \min(h(\max(S_k)), h(\min(T_k))) + 1)\right) + \mathcal{O}\left(\sum_{i=1}^{m}(H(\Lambda_i) - \min(h(\min(\Lambda_i)), h(\max(\Lambda_i))) + 1)\right).$$

Proof. Let $\hat{c_k}$ and c_k respectively be the amortized and worst-case running time of BSL-FJoin(ℓ_k, r_k). Note that $\Phi_e(L_0) = \mathcal{O}(|\mathcal{P}_0|) = \mathcal{O}\left(\sum_{i=1}^{m}(H(\Lambda_i) - \min(h(\min(\Lambda_i)), h(\max(\Lambda_i))) + 1)\right)$. Then we have

$$\sum_{k=1}^{t} \hat{c_k} = \sum_{k=1}^{t} (c_k + \Phi_e(L_k) - \Phi_e(L_{k-1}))$$
$$\sum_{k=1}^{t} \hat{c_k} = \sum_{k=1}^{t} c_k + \Phi_e(L_t) - \Phi_e(L_0)$$
$$\sum_{k=1}^{t} c_k = \sum_{k=1}^{t} \hat{c_k} - \Phi_e(L_t) + \Phi_e(L_0)$$
$$\sum_{k=1}^{t} c_k \leq \sum_{k=1}^{t} \hat{c_k} + \Phi_e(L_0)$$
$$= \mathcal{O}\left(\sum_{k=1}^{t} (\min(H(S_k), H(T_k)) - \min(h(\max(S_k)), h(\min(T_k))) + 1)\right) \quad \text{By Lemma 5}$$
$$+ \Phi_e(L_0)$$
$$= \mathcal{O}\left(\sum_{k=1}^{t} (\min(H(S_k), H(T_k)) - \min(h(\max(S_k)), h(\min(T_k))) + 1)\right)$$
$$+ \mathcal{O}\left(\sum_{i=1}^{m} (H(\Lambda_i) - \min(h(\min(\Lambda_i)), h(\max(\Lambda_i))) + 1)\right).$$

□

Lemma 7. *The* BSL-FRew(f, w) *operation, where* $f \in S$*, has a worst-case and amortized time complexity of*

$$\mathcal{O}(\max(H(S), r'(f)) - \min(h(f), r'(f)) + 1).$$

Proof. BSL-FRew(f, w) only spends constant time at each level between $\min(h(f), r'(f))$ and $\max(h(f), r'(f))$ for promoting or demoting f; and at most constant time at each level between $\min(h(f), r'(f))$ and $\max(H(S), r'(f))$ for restoring invariants. Therefore, the worst-case complexity of BSL-FRew(f, w) is $\mathcal{O}(\max(H(S), r'(f)) - \min(h(f), r'(f)) + 1)$.

Let BSL-FRew(f, w) be the k^{th} operation. Note that no plateaus that are on levels less than $\min(h(f), r'(f))$ are affected by the operation. Since the potential increase associated with each level is bounded by a constant, we have $\Phi_e(L_k) - \Phi_e(L_{k-1}) = \mathcal{O}(\max(H(S), r'(f)) - \min(h(f), r'(f)))$. Thus, the lemma follows. □

Lemma 8. *The* BSL-FSearch*,* BSL-FSplit*,* BSL-FJoin*, and* BSL-FRew *operations all have a worst-case time complexity of* $\mathcal{O}(\log n)$*.*

Proof. Let W' be the sum of the weights of all elements in the sets involved in any of these four operations. By our previous assumptions, W' is bounded from above by a polynomial in n. Since these versions of the operations are more efficient than the non-finger versions which take $\mathcal{O}(\log W')$ time in the worst case, these operations have a worst-case time complexity of $\mathcal{O}(\log n)$. □

We now present our data structure, the Mergeable Dictionary, which improves the worst-case bound in Theorem 1 by a factor of $\log n$, matching the lower bound of [11].

4 Our Data Structure: The Mergeable Dictionary

The Mergeable Dictionary stores each set in the collection $\mathcal{S}$ as a biased skip list. The weight of each node in each biased skip list is determined by $\mathcal{S}$. When the collection of sets is modified, for instance via a MERGE operation, in order to reflect this change in the data structure, besides splitting and joining biased skip lists, we need to ensure the weights of the affected nodes are properly updated and biased skip list invariants (**I0**), (**I1**), and (**I2**) are preserved. For simplicity we assume that D_0 is the collection of singleton sets and D_i, for all i, partitions the universe $\mathcal{U}$. This lets us precompute, for each node x, $\text{pos}_{\mathcal{U}}(x)$, the global position of x.

For the MERGE algorithm, we will use the same basic approach outlined in Section 2, the segment merging heuristic, which works by extracting the segments from each set and then gluing them together to form the union of the two sets.

As previously mentioned, while we do not have control over the number of segments that need to be processed which has been shown to have an amortized lower bound of $\Omega(\log n)$ per operation, we need to process each segment faster. In order to do so, we depart from balanced search trees and instead use Biased Skip Lists [2] with the extended operations we introduced in Section 3.

Before we discuss the implementation of each operation in detail, we need to describe the weighting scheme.

4.1 Weighting Scheme

Let the weight of a node x, $w(x)$, be the sum of the sizes of its adjacent gaps. In other words, if $pos_S(x) = k$ for some node $x \in S$, then we have

$$w(x) = g_S(k-1) + g_S(k).$$

Recall that $g_S(0) = g_S(|S|) = 1$. Observe that this implies for any set S, $W(S) \leq 2n$.

4.2 The Find, Search, and Split Operations

The FIND(x) operation can simply return the maximum element of the set which contains x by invoking BSL-FSEARCH$(i, +\infty)$. The SEARCH(X, i) operation can be performed by simply invoking BSL-SEARCH(X, i). The SPLIT(X, i) operation can be performed by simply invoking BSL-SPLIT(X, i) and running BSL-FREW on one node in each of the resulting biased skip lists to restore the weights.

4.3 The Merge Operation

We will use the Mergeable Dictionary in Figure 2 to illustrate the MERGE operation. The MERGE(A, B) operation can be viewed as having four essential phases: finding the segments, extracting the segments, updating the weights, and gluing the segments. A more detailed description follows.

Phase I: Finding the segments Assume $\min(A) < \min(B)$ w.l.o.g. Let $z = \lceil |I(A,B)|/2 \rceil$ and $v = \lfloor |I(A,B)|/2 \rfloor$. Recall that $I(A,B) = \{A_1, B_1, A_2, B_2, \ldots\}$ is the set of segments associated with the MERGE(A, B) operation where A_i and B_i are the i^{th} segment of A and B respectively. We have $\min(A_1) = \min(A)$ and $\min(B_1) = \min(B)$. Given $\min(A_i)$ and $\min(B_i)$, we find $\max(A_i)$ by invoking BSL-FSEARCH$(\min(A_i), \min(B_i))$. Similarly, given $\min(B_i)$ and $\min(A_{i+1})$ we find

$\max(B_i)$ by invoking BSL-FSEARCH$(\min(B_i), \min(A_{i+1}))$. Lastly, given $\max(A_i)$ and $\max(B_i)$, observe that $\min(A_{i+1}) = \text{succ}_A(\max(A_i))$ and $\min(B_{i+1}) = \text{succ}_B(\max(B_i))$. Note that the succ() operation is performed in constant time in a biased skip list using the lowest successor link of a node. At the end of this phase, all the segments are found (see Figure 3). Specifically, we have computed for all i and j $(\min(A_i), \max(A_i))$ and $(\min(B_j), \max(B_j))$.

Phase II: Extracting the segments Since we know where the minimum and maximum node of each segment is from the previous phase, we can extract all the segments easily in order by invoking BSL-FSPLIT$(\max(A_i))$ for $1 \leq i < z$, and BSL-FSPLIT$(\max(B_j))$ for $1 \leq j < v$ (see Figure 4).

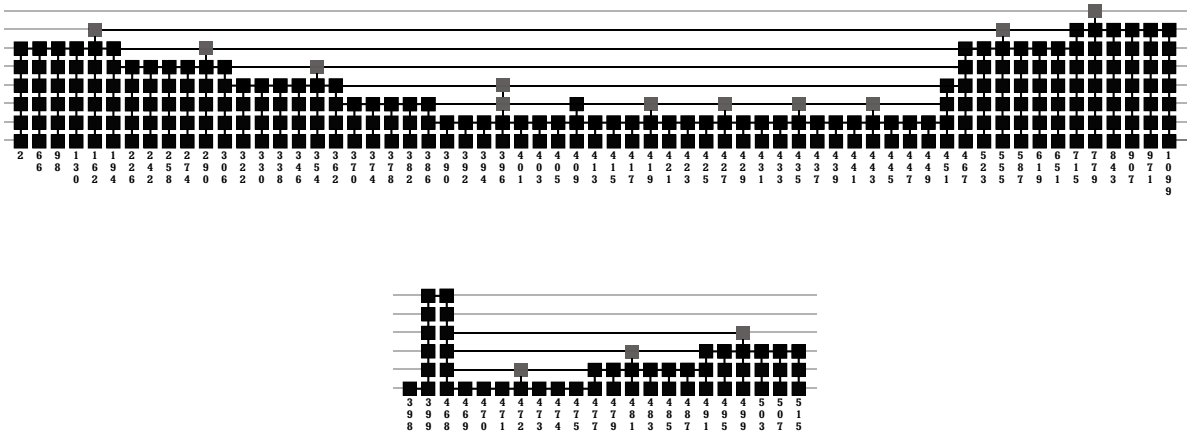

Figure 2: We have two biased skip lists A (top) and B (bottom). The tower of each node is represented by a set of vertically stacked squares which represent individual levels of a node. The predecessor/successor pointers are also shown. The lightly shaded levels exist due to promotions. The position of each node x, $\text{pos}_{\mathcal{U}}(x)$, is indicated at the bottom of the tower of each node.

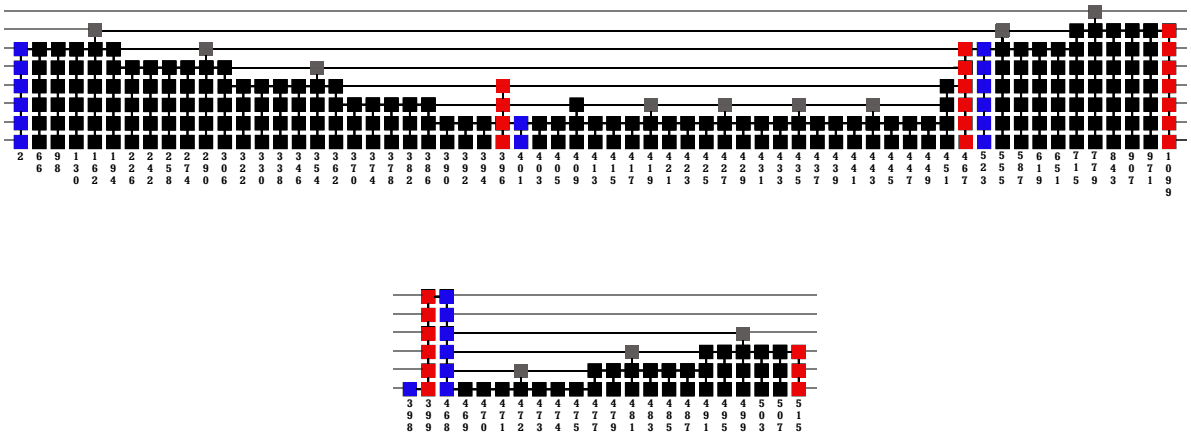

Figure 3: At the end of Phase I, we have all the minimum and maximum nodes of the segments of the MERGE(A, B) operation. The minimum nodes are colored blue, the maximum nodes are colored red.

Phase III: Updating Weights Next, we need to update the weights of the affected nodes. Let the new weight of item x be $w'(x)$. Then

$$\begin{aligned}
&\text{For } 2 \le i \le z, \text{ let}\\
&\qquad w'(\min(A_i)) = w(\min(A_i)) + pos_U(\max(A_{i-1})) - pos_U(\max(B_{i-1})),\\
&\text{For } 2 \le i \le v, \text{ let}\\
&\qquad w'(\min(B_i)) = w(\min(B_i)) + pos_U(\max(B_{i-1})) - pos_U(\max(A_i)),\\
&\text{For } 1 \le i < z, \text{ let}\\
&\qquad w'(\max(A_i)) = w(\max(A_i)) + pos_U(\min(B_i)) - pos_U(\min(A_{i+1})),\\
&\text{For } 1 \le i < v, \text{ let}\\
&\qquad w'(\max(B_i)) = w(\max(B_i)) + pos_U(\min(A_{i+1})) - pos_U(\min(B_{i+1})).
\end{aligned}$$

We also have

$$w'(\min(B_1)) = w(\min(B_1)) - 1 + pos_U(\min(B_1)) - pos_U(\max(A_1)).$$

If $|I(A,B)|$ is even, we have

$$w'(\max(A_z)) = w(\max(A_z)) - 1 + pos_U(\min(B_z)) - pos_U(\max(A_z)).$$

If $|I(A,B)|$ is odd, we have

$$w'(\max(B_v)) = w(\max(B_v)) - 1 + pos_U(\min(A_z)) - pos_U(\max(B_v)).$$

We can perform these weight updates (see Figure 5) by invoking

- BSL-FRew$(\min(A_i), w'(\min(A_i)))$ for $2 \le i \le z$,
- BSL-FRew$(\max(A_i), w'(\max(A_i)))$ for $1 \le i \le v$,
- BSL-FRew$(\min(B_j), w'(\min(B_j)))$ for $1 \le j \le v$,
- BSL-FRew$(\max(B_j), w'(\max(B_j)))$ for $1 \le j < z$.

Phase IV: Gluing the segments Since we assumed w.l.o.g. that $\min(A) < \min(B)$, the correct order of the segments is $(A_1, B_1, A_2, B_2, \ldots)$ by construction. We can glue all the segments by invoking BSL-FJoin$(\max(A_i), \min(B_i))$ for $1 \le i \le v$ and BSL-FJoin$(\max(B_i), \min(A_{i+1}))$ for $1 \le i < z$ (see Figure 6).

This concludes the presentation of our data structure. In the next section we will analyze the running time of each of the 4 operations.

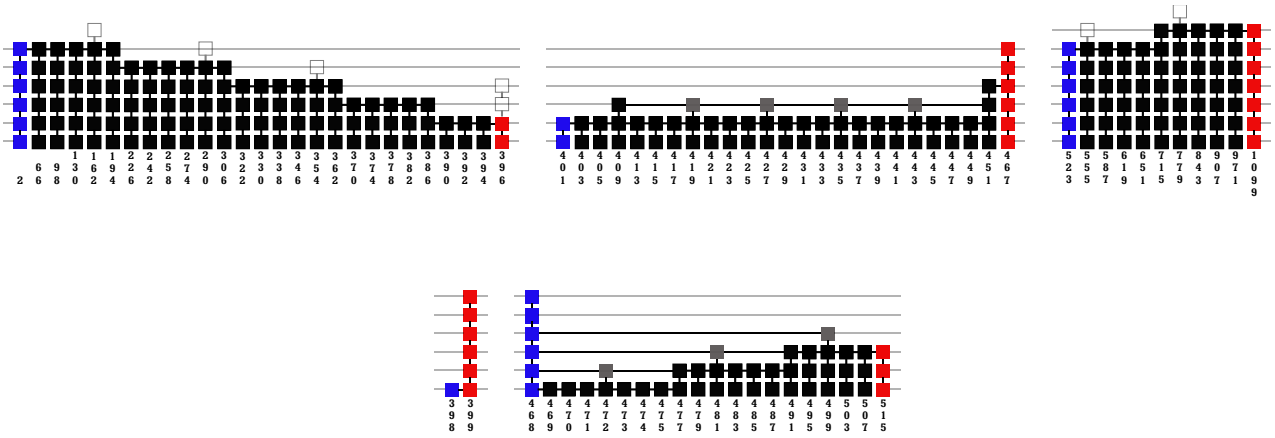

Figure 4: At the end of Phase II, all the segments are extracted. These extractions cause invariants (**I1**) and (**I2**) to be violated, which are then restored by BSL-FSplit. Hollow levels denote demotions.

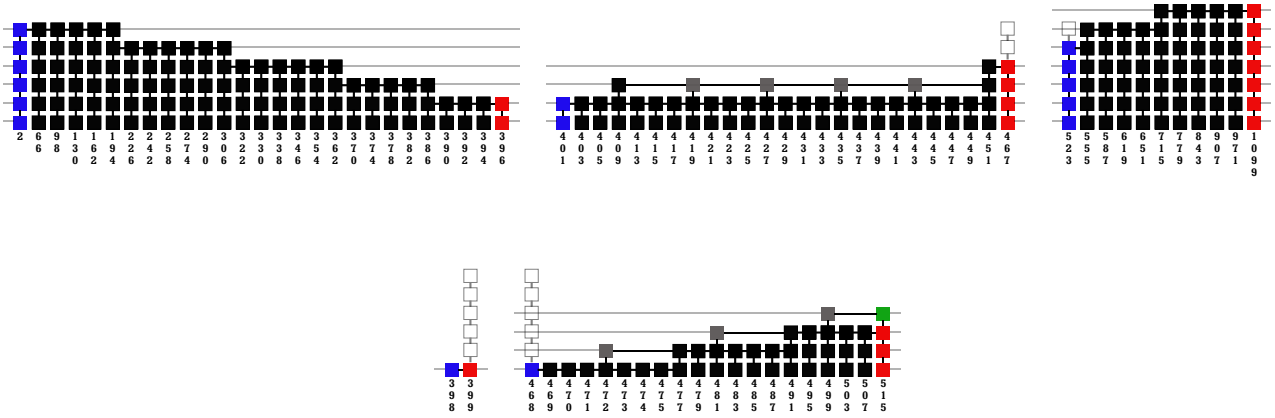

Figure 5: The weights of the minimum and maximum nodes of each segment are affected by the Merge(A, B) operation. Accordingly, in Phase III, we update the weights of these nodes. This update causes (**I0**) violations. The BSL-FRew operation first restores (**I0**). Restoring (**I0**) causes (**I1**) and (**I2**) violations which are then again restored by BSL-FRew. The hollow levels denote demotions and green levels denote promotions.

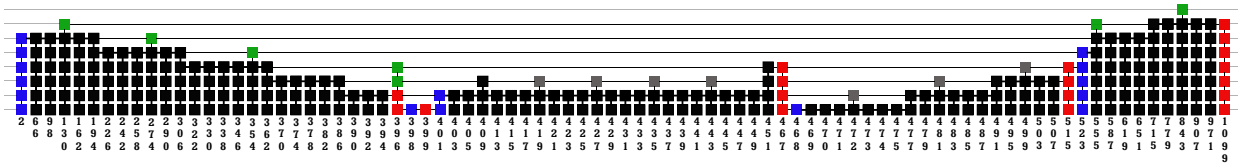

Figure 6: In Phase IV, we join all the segments to obtain the union of A and B. These joins cause invariants (**I1**) and (**I2**) to be violated, which are then restored by BSL-FJoin. The green levels denote promotions.

5 Analysis of the Mergeable Dictionary

Before we can analyze the amortized time complexity of the Mergeable Dictionary operations, we need a new potential function which we will present in Section 5.1. We then prove that all the operations except MERGE have a worst-case and amortized time complexity of $\mathcal{O}(\log n)$ in Section 5.2. Lastly, we will show that the MERGE operation has an amortized time complexity of $\mathcal{O}(\log n)$ in Section 5.3.

5.1 The New Potential Function

Let D_i be the data structure containing our dynamic collection of disjoint sets, $\mathcal{S}^{(i)} = \{S_1^{(i)}, S_2^{(i)}, \ldots\}$ after the i^{th} operation. Let

$$\varphi(S) = \sum_{x \in S} (\log g_S(pos_S(x) - 1) + \log g_S(pos_S(x))).$$

Then we define the potential after the i^{th} operation as follows.

$$\Phi(D_i) = \kappa_d \cdot \sum_j \varphi(S_j^{(i)})$$

where κ_d is a constant to be determined later.

Note that the main difference between this function and the one in Section 2.2 is the elimination of the $\log n$ term.

5.2 The Analysis of the Find, Search, and Split Operations

We now show that all the operations except MERGE have a worst-case time complexity of $\mathcal{O}(\log n)$, and they do not cause a substantial increase in the potential which yields that their amortized time complexity is also $\mathcal{O}(\log n)$.

Theorem 9. *The worst-case and amortized time complexity of the* FIND(x) *operation, the* SEARCH(S, x) *operation, and the* SPLIT(S, x) *operation is* $\mathcal{O}(\log n)$.

Proof. The worst-case time complexity of the BSL-FSEARCH operation invoked by the FIND and SEARCH operations is $\mathcal{O}(\log n)$ by Lemma 8. Recall that since these operations do not change the structure, the potential remains the same. Therefore, worst-case and amortized time complexity of FIND and SEARCH is $\mathcal{O}(\log n)$. The worst-case time-complexity of the BSL-SPLIT and BSL-REW operations invoked by the SPLIT operation is $\mathcal{O}(\log n)$ by Lemma 8. Observe that SPLIT can only decrease the potential. Therefore, the worst-case and amortized time complexity of SPLIT is $\mathcal{O}(\log n)$. □

5.3 The Analysis of the Merge Operation

All we have left to do is show that the amortized time complexity of the MERGE operation is $\mathcal{O}(\log n)$. In order to do this, first we will show that the worst-case time complexity of the MERGE(A, B) operation is $\mathcal{O}\left(\log n + \sum_i F(A_i) + \sum_j F(B_j)\right)$. We define $F(A_i)$ and $F(B_j)$ next.

Definition 10. *Consider the* MERGE(A, B) *operation. Recall that* $w'(x)$ *is the new weight of node* x *after the* MERGE(A, B) *operation. Also recall that* $z = \lceil |I(A,B)|/2 \rceil$ *and* $v = \lfloor |I(A,B)|/2 \rfloor$. *Then, for* $1 < i < z$, *let*

$$F(A_i) = \log \frac{w(\max(A_{i-1})) + w(\min(A_{i+1})) + \sum_{x \in A_i} w(x)}{\min(w'(\max(B_{i-1})), w'(\min(A_i)), w'(\max(A_i)), w'(\min(B_i)))}$$

and for $1 < j < v$, *let*

$$F(B_j) = \log \frac{w(\max(B_{j-1})) + w(\min(B_{j+1})) + \sum_{x \in B_j} w(x)}{\min(w'(\max(A_j)), w'(\min(B_j)), w'(\max(B_j)), w'(\min(A_{j+1})))}.$$

For the boundary cases, let $F(A_1) = F(B_1) = F(A_z) = F(B_v) = \log n$.

5.3.1 The Worst-Case Time Complexity

We need to bound the worst-case time complexity of each phase of the MERGE(A, B) operation.

Lemma 11. *Phase I of the* MERGE *operation has a worst-case time complexity of*

$$\mathcal{O}\left(\log n + \sum_{i=2}^{z-1} F(A_i) + \sum_{j=2}^{v-1} F(B_j)\right).$$

Proof. During Phase I, BSL-FSEARCH$(\min(A_i), t)$ is invoked for $1 \le i < z$ where $\text{pred}_A(t) = \max(A_i)$ and $\text{succ}_A(t) = \min(A_{i+1})$; and BSL-FSEARCH$(\min(B_j), s)$ is invoked for $1 \le j < v$ where $\text{pred}_B(s) = \max(B_j)$ and $\text{succ}_B(s) = \min(B_{j+1})$. Observe that $w'(\min(B_i)) < w(\min(A_{i+1}))$ and $w'(\min(B_i)) < w(\max(A_i))$. Therefore, by the definition of $F(A_i)$ and $F(B_j)$, and Lemma 8, the worst-case time complexity of Phase I is $\mathcal{O}\left(\log n + \sum_{i=2}^{z-1} F(A_i) + \sum_{j=2}^{v-1} F(B_j)\right)$. □

We will need the following lemma to bound the worst-case time complexity of Phase II-IV.

Lemma 12. *Given a biased skip list* S *and any node* $f \in S$, *recall that* $S[\hookleftarrow f] = \{x \in S \mid x \le f\}$. *Then,*

$$H(S[\hookleftarrow f]) \le \log W(S[\hookleftarrow f]).$$

Proof. Let $R = \max_{x \in S[\hookleftarrow f]} r(x)$ and $N_r(t) = \{x \in S[\hookleftarrow f] \mid r(x) = t\}$. Also, let $N_h(t) = \{x \in S[\hookleftarrow f] \mid h(x) \ge t,\ r(x) \le t\}$. Then we have

$$\begin{aligned} W(S[\hookleftarrow f]) &\ge \sum_{i=2}^{R} 2^i N_r(i) + \sum_{\substack{x \in S[\hookleftarrow f], \\ r(x)=1}} w(x) \\ &\ge \sum_{i=2}^{R} 2^i N_r(i) + 2N_h(1). \end{aligned}$$

Due to (**I2**), we have

$$N_h(i) \ge 2(N_h(i+1) - N_r(i+1))$$

$$
\begin{aligned}
&\geq 2^{R-1} N_h(R) - \sum_{i=2}^{R} 2^{i-1} N_r(i) \\
&\geq 2^{H(S[\hookleftarrow f])-1} N_h(H(S[\hookleftarrow f])) - \sum_{i=2}^{R} 2^{i-1} N_r(i) && N_r(t) = 0 \text{ for } t > R \\
&\geq 2^{H(S[\hookleftarrow f])-1} - \sum_{i=2}^{R} 2^{i-1} N_r(i)
\end{aligned}
$$

which yields

$$
\begin{aligned}
W(S[\hookleftarrow f]) &\geq \sum_{i=2}^{R} 2^i N_r(i) + 2N_h(1) \\
&\geq \sum_{i=2}^{R} 2^i N_r(i) + 2\left(2^{H(S[\hookleftarrow f])-1} - \sum_{i=2}^{R} 2^{i-1} N_r(i)\right) \\
&\geq 2^{H(S[\hookleftarrow f])} \\
\log W(S[\hookleftarrow f]) &\geq H(S[\hookleftarrow f]).
\end{aligned}
$$

□

Lemma 13. *Phase II of the* MERGE *operation has a worst-case time complexity of*

$$
\mathcal{O}\left(\log n + \sum_{i=2}^{z-1} F(A_i) + \sum_{j=2}^{v-1} F(B_j)\right).
$$

Proof. During Phase II, BSL-FSPLIT($\max(A_i)$) is invoked for $1 \leq i < z$, and BSL-FSPLIT($\max(B_j)$) is invoked for $1 \leq j < v$. Combining Lemma 4 and Lemma 12 yields that the worst-case time complexity of Phase II is

$$
\mathcal{O}\left(\sum_{i=1}^{z-1} (\log W(A_i) - \min(r(\min(A_i)), r(\max(A_i)), r(\min(A_{i+1}))) + 1)\right) +
$$

$$
\mathcal{O}\left(\sum_{j=1}^{v-1} (\log W(B_j) - \min(r(\min(B_j)), r(\max(B_j)), r(\min(B_{j+1}))) + 1)\right)
$$

Observe that $w'(\min(B_i)) < w(A_{i+1})$ and $w'(\min(A_{j+1})) < w(B_{j+1})$. Then, by Lemma 8 and the definitions of $F(A_i)$ and $F(B_j)$, the worst-case complexity of Phase II is

$$
\mathcal{O}\left(\log n + \sum_{i=2}^{z-1} F(A_i) + \sum_{j=2}^{v-1} F(B_j)\right).
$$

□

Lemma 14. *Phase III of the* MERGE *operation has a worst-case time complexity of*

$$\mathcal{O}\left(\log n + \sum_{i=2}^{z-1} F(A_i) + \sum_{j=2}^{v-1} F(B_j)\right).$$

Proof. During Phase III, we invoke

- BSL-FREW$(\min(A_i), w'(\min(A_i)))$ for $2 \leq i \leq z$,
- BSL-FREW$(\max(A_i), w'(\max(A_i)))$ for $1 \leq i \leq v$,
- BSL-FREW$(\min(B_j), w'(\min(B_j)))$ for $1 \leq j \leq v$,
- BSL-FREW$(\max(B_j), w'(\max(B_j)))$ for $1 \leq j < z$.

Let $t_1 = \min(B)$. If $|I(A, B)|$ is even, let $t_2 = \max(A)$, otherwise let $t_2 = \max(B)$. Observe that $r'(x) \leq r(x) \leq h(x)$ if and only if $x \notin \{t_1, t_2\}$. Then, by Lemma 7 and Lemma 12, BSL-FREW$(x, w'(x))$ has a worst-case time complexity of

- $\mathcal{O}(\log W(A_i) - r'(\min(A_i)) + 1)$ for all $x \in \{\min(A_i) \,|\, 1 \leq i \leq z\}$,
- $\mathcal{O}(\log W(A_i) - r'(\max(A_i)) + 1)$ for all $x \in \{\max(A_i) \,|\, 1 \leq i \leq z\} \setminus \{t_2\}$,
- $\mathcal{O}(\log W(B_j) - r'(\min(B_j)) + 1)$ for all $x \in \{\min(B_j) \,|\, 1 \leq j \leq v\} \setminus \{t_1\}$,
- $\mathcal{O}(\log W(B_j) - r'(\max(B_j)) + 1)$ for all $x \in \{\max(B_j) \,|\, 1 \leq j \leq v\} \setminus \{t_2\}$,

and a worst-case time complexity of $\mathcal{O}(\log n)$ for $x \in \{t_1, t_2\}$.

Therefore, by Lemma 8 and the definitions of $F(A_i)$ and $F(B_j)$, the worst-case time complexity of Phase III is $\mathcal{O}\left(\log n + \sum_{i=2}^{z-1} F(A_i) + \sum_{j=2}^{v-1} F(B_j)\right)$.

□

Lemma 15. *Phase IV of the* MERGE *operation has a worst-case time complexity of*

$$\mathcal{O}\left(\log n + \sum_{i=2}^{z-1} F(A_i) + \sum_{j=2}^{v-1} F(B_j)\right).$$

Proof. During Phase IV, we invoke BSL-FJOIN$(\max(A_i), \min(B_i))$ for $1 \leq i \leq v$ and we invoke BSL-FJOIN$(\max(B_i), \min(A_{i+1}))$ for $1 \leq i < z$. Combining Lemma 6 and Lemma 12 yields that the worst-case time complexity of Phase IV is

$$\mathcal{O}\left(\sum_{i=2}^{z} (\log W(A_i) - \min(r(\max(B_{i-1})), r(\min(A_i)), r(\max(A_i))) + 1)\right) +$$

$$\mathcal{O}\left(\sum_{j=1}^{v} (\log W(B_j) - \min(r(\max(A_j)), r(\min(B_j)), r(\max(B_j))) + 1)\right)$$

which by Lemma 8 and the definitions of $F(A_i)$ and $F(B_j)$ yields

$$\mathcal{O}\left(\log n + \sum_{i=2}^{z-1} F(A_i) + \sum_{j=2}^{v-1} F(B_j)\right).$$

□

The next theorem bounds the worst-case time complexity of the MERGE(A, B) operation.

Theorem 16. *The* MERGE(A, B) *operation has a worst-case time complexity of*

$$\mathcal{O}\left(\log n + \sum_{i=2}^{z-1} F(A_i) + \sum_{j=2}^{v-1} F(B_j)\right).$$

Proof. The worst-case time complexity of the MERGE(A, B) operation is determined by the time it spends on each of the four phases. Therefore, by lemmas 11, 13, 14, and 15, the theorem follows. □

5.3.2 Amortized Time Complexity

Before we can show that the amortized time complexity of the MERGE(A, B) operation is $\mathcal{O}(\log n)$, we will need to prove 3 lemmas. Let us first define the potential loss associated with a gap. Recall the definitions of gaps a_i, a'_i, a''_i and similarly b_j, b'_j, b''_j first defined in Section 2.3.

Definition 17. *We define pl(a_i) and pl(b_j), the potential loss associated with gap a_i and the potential loss associated with gap b_i respectively, for $1 \leq i < z$ and $1 \leq j < v$, as follows:*

$$pl(a_i) = 2\log a_i - \log a'_i - \log a''_i \tag{1}$$
$$pl(b_j) = 2\log b_j - \log b'_j - \log b''_j \tag{2}$$

Assume w.l.o.g. that $\min(A) < \min(B)$. *Then we also let* $pl(a_0) = 0$ *and*

$$pl(b_0) = -\log a'_1. \tag{3}$$

If $\max(A) > \max(B)$, *then we have* $pl(a_z) = 0$ *and*

$$pl(b_v) = -\log a''_{z-1}. \tag{4}$$

Otherwise, if $\max(A) < \max(B)$, *we have* $pl(b_v) = 0$ *and*

$$pl(a_z) = -\log b''_{v-1}. \tag{5}$$

Note that the potential loss associated with operation MERGE(A, B) is κ_d times the sum of all defined $\text{pl}(a_i)$ and $\text{pl}(b_j)$, where κ_d is the constant in the potential function.

Lemma 18. *Consider gap a_i for any $1 \le i < z$. Let $a_i^+ = \max(a_i', a_i'')$ and $a_i^- = \min(a_i', a_i'')$. Then we have*

$$2^{-pl(a_i)} \le \frac{a_i}{a_i^+} \le \frac{a_i}{a_i^-} \le 2^{pl(a_i)} \quad \textit{and} \quad 2^{-pl(a_i)} \le \frac{a_i^+}{a_i^-} \le 2^{pl(a_i)}.$$

Similarly, consider gap b_j for any $1 \le j < v$, where $b_j^+ = \max(b_j', b_j'')$ and $b_j^- = \min(b_j', b_j'')$. Then we have

$$2^{-pl(b_j)} \le \frac{b_j}{b_j^+} \le \frac{b_j}{b_j^-} \le 2^{pl(b_j)} \quad \textit{and} \quad 2^{-pl(b_j)} \le \frac{b_j^+}{b_j^-} \le 2^{pl(b_j)}.$$

Proof. We will prove the first part dealing with gap a_i. The proof of the second part is symmetric. Assume w.l.o.g. that $a_i' \le a_i''$. We have

$$\begin{aligned} \log a_i'' + \log a_i' - 2\log a_i &= -\mathrm{pl}(a_i) \\ \log \frac{a_i}{a_i'} + \log \frac{a_i}{a_i''} &= \mathrm{pl}(a_i) \end{aligned}$$

which gives us

$$\log \frac{a_i}{a_i''} \le \mathrm{pl}(a_i) \quad \text{and} \quad \log \frac{a_i}{a_i'} \le \mathrm{pl}(a_i),$$

and consequently

$$\frac{a_i}{a_i''} \le 2^{\mathrm{pl}(a_i)} \quad \text{and} \quad \frac{a_i}{a_i'} \le 2^{\mathrm{pl}(a_i)}.$$

Because $a_i' \le a_i'' \le a_i$, we have

$$\frac{a_i}{a_i''} \le \frac{a_i}{a_i'} \le 2^{\mathrm{pl}(a_i)}. \tag{6}$$

Also, note that

$$\begin{aligned} \frac{a_i}{a_i'} &\le 2^{\mathrm{pl}(a_i)} \\ \frac{1}{2^{\mathrm{pl}(a_i)}} &\le \frac{a_i'}{a_i} \\ \frac{1}{2^{\mathrm{pl}(a_i)}} \le \frac{a_i'}{a_i} &\le \frac{a_i}{a_i'}, \end{aligned}$$

and similarly,

$$\frac{1}{2^{\mathrm{pl}(a_i)}} \le \frac{a_i''}{a_i} \le \frac{a_i}{a_i''}. \tag{7}$$

Lastly, observe that

$$\begin{aligned} \frac{a_i''}{a_i'} &\le \frac{a_i}{a_i'} \\ &\le 2^{\mathrm{pl}(a_i)} \quad \text{by (6)}, \end{aligned}$$

and

$$\frac{a_i''}{a_i} \leq \frac{a_i''}{a_i'}$$
$$\frac{1}{2^{\mathrm{pl}(a_i)}} \leq \frac{a_i''}{a_i'} \quad \text{by (7).}$$

This concludes the proof. Second part of the lemma can be proven using symmetric arguments. □

Lemma 19. *Let $I(A,B) = \{A_1, B_1, A_2, B_2, \ldots\}$ be the set of segments of with respect to operation* MERGE(A,B). *For any i and j, where $1 < i < z$ and $1 < j < v$, let*

$$\alpha_i = \max\left(2^{pl(a_{i-2})}, 2^{pl(b_{i-2})}, 2^{pl(a_{i-1})}, 2^{pl(b_{i-1})}, 2^{pl(a_i)}, 2^{pl(b_i)}, 2^{pl(a_{i+1})}\right)$$

and

$$\beta_j = \max\left(2^{pl(b_{j-2})}, 2^{pl(a_{j-1})}, 2^{pl(b_{j-1})}, 2^{pl(a_j)}, 2^{pl(b_j)}, 2^{pl(a_{j+1})}, 2^{pl(b_{j+1})}\right).$$

Then for $1 < i < z$ and $1 < j < v$, we have

$$F(A_i) = \mathcal{O}(\log \alpha_i) \quad \text{and} \quad F(B_j) = \mathcal{O}(\log \beta_j).$$

Proof. We will present the proof of the first equality. The proof of the second one is analogous. Let $a_{i-1}'' = b_{i-1}' = x$. Then, by Lemma 18 we have

$$\frac{1}{2^{\mathrm{pl}(a_{i-1})}} \leq \frac{a_{i-1}'}{a_{i-1}''} \leq 2^{\mathrm{pl}(a_{i-1})} \quad \text{and} \quad \frac{1}{2^{\mathrm{pl}(b_{i-1})}} \leq \frac{b_{i-1}''}{b_{i-1}'} \leq 2^{\mathrm{pl}(b_{i-1})}$$

which imply

$$\frac{x}{\alpha_i} \leq \frac{x}{2^{\mathrm{pl}(a_{i-1})}} \leq a_{i-1}' \leq x2^{\mathrm{pl}(a_{i-1})} \leq x\alpha_i \tag{8}$$

and

$$\frac{x}{\alpha_i} \leq \frac{x}{2^{\mathrm{pl}(b_{i-1})}} \leq b_{i-1}'' \leq x2^{\mathrm{pl}(b_{i-1})} \leq x\alpha_i \tag{9}$$

Note that $a_i' = b_{i-1}''$, and $b_{i-2}'' = a_{i-1}'$. By Lemma 18, we have

$$\frac{1}{2^{\mathrm{pl}(a_i)}} \leq \frac{a_i''}{a_i'} \leq 2^{\mathrm{pl}(a_i)} \quad \text{and} \quad \frac{1}{2^{\mathrm{pl}(b_{i-2})}} \leq \frac{b_{i-2}'}{b_{i-2}''} \leq 2^{\mathrm{pl}(b_{i-2})}$$

which imply

$$\frac{x}{\alpha_i^2} \leq \frac{1}{2^{\mathrm{pl}(a_i)}} \cdot \frac{x}{\alpha_i} \leq a_i'' \leq 2^{\mathrm{pl}(a_i)} \cdot x\alpha_i \leq x\alpha_i^2 \tag{10}$$

and

$$\frac{x}{\alpha_i^2} \leq \frac{1}{2^{\mathrm{pl}(b_{i-2})}} \cdot \frac{x}{\alpha_i} \leq b_{i-2}' \leq 2^{\mathrm{pl}(b_{i-2})} \cdot x\alpha_i \leq x\alpha_i^2 \tag{11}$$

Note that $a_{i-2}'' = b_{i-2}'$, and $b_i' = a_i''$. By Lemma 18, we have

$$\frac{1}{2^{\mathrm{pl}(a_{i-2})}} \leq \frac{a_{i-2}'}{a_{i-2}''} \leq 2^{\mathrm{pl}(a_{i-2})} \quad \text{and} \quad \frac{1}{2^{\mathrm{pl}(b_i)}} \leq \frac{b_i''}{b_i'} \leq 2^{\mathrm{pl}(b_i)}$$

which imply

$$\frac{x}{\alpha_i^3} \le \frac{1}{2^{\mathrm{pl}(a_{i-2})}} \cdot \frac{x}{\alpha_i^2} \le a'_{i-2} \le 2^{\mathrm{pl}(a_{i-2})} \cdot x\alpha_i^2 \le x\alpha_i^3 \tag{12}$$

and

$$\frac{x}{\alpha_i^3} \le \frac{1}{2^{\mathrm{pl}(b_i)}} \cdot \frac{x}{\alpha_i^2} \le b''_i \le 2^{\mathrm{pl}(b_i)} \cdot x\alpha_i^2 \le x\alpha_i^3 \tag{13}$$

Note that $a'_{i+1} = b''_i$. By Lemma 18, we have

$$\frac{1}{2^{\mathrm{pl}(a_{i+1})}} \le \frac{a''_{i+1}}{a'_{i+1}} \le 2^{\mathrm{pl}(a_{i+1})}$$

which implies

$$\frac{x}{\alpha_i^4} \le \frac{1}{2^{\mathrm{pl}(a_{i+1})}} \cdot \frac{x}{\alpha_i^3} \le a''_{i+1} \le 2^{\mathrm{pl}(a_{i+1})} \cdot x\alpha_i^3 \le x\alpha_i^4 \tag{14}$$

Similarly, by Lemma 18, we have

$$\frac{1}{2^{\mathrm{pl}(a_{i-1})}} \le \frac{a_{i-1}}{a''_{i-1}} \le 2^{\mathrm{pl}(a_{i-1})} \quad \text{and} \quad \frac{1}{2^{\mathrm{pl}(b_{i-1})}} \le \frac{b_{i-1}}{b'_{i-1}} \le 2^{\mathrm{pl}(b_{i-1})}$$

which imply

$$\frac{x}{\alpha_i} \le \frac{x}{2^{\mathrm{pl}(a_{i-1})}} \le a_{i-1} \le x2^{\mathrm{pl}(a_{i-1})} \le x\alpha_i \tag{15}$$

and

$$\frac{x}{\alpha_i} \le \frac{x}{2^{\mathrm{pl}(b_{i-1})}} \le b_{i-1} \le x2^{\mathrm{pl}(b_{i-1})} \le x\alpha_i \tag{16}$$

By Lemma 18, we have

$$\frac{1}{2^{\mathrm{pl}(a_i)}} \le \frac{a_i}{a'_i} \le 2^{\mathrm{pl}(a_i)} \quad \text{and} \quad \frac{1}{2^{\mathrm{pl}(b_{i-2})}} \le \frac{b_{i-2}}{b''_{i-2}} \le 2^{\mathrm{pl}(b_{i-2})}$$

which imply

$$\frac{x}{\alpha_i^2} \le \frac{1}{2^{\mathrm{pl}(a_i)}} \cdot \frac{x}{\alpha_i} \le a_i \le 2^{\mathrm{pl}(a_i)} \cdot x\alpha_i \le x\alpha_i^2 \tag{17}$$

and

$$\frac{x}{\alpha_i^2} \le \frac{1}{2^{\mathrm{pl}(b_{i-2})}} \cdot \frac{x}{\alpha_i} \le b_{i-2} \le 2^{\mathrm{pl}(b_{i-2})} \cdot x\alpha_i \le x\alpha_i^2 \tag{18}$$

By Lemma 18, we have

$$\frac{1}{2^{\mathrm{pl}(a_{i-2})}} \le \frac{a_{i-2}}{a''_{i-2}} \le 2^{\mathrm{pl}(a_{i-2})} \quad \text{and} \quad \frac{1}{2^{\mathrm{pl}(b_i)}} \le \frac{b_i}{b'_i} \le 2^{\mathrm{pl}(b_i)}$$

which imply

$$\frac{x}{\alpha_i^3} \le \frac{1}{2^{\mathrm{pl}(a_{i-2})}} \cdot \frac{x}{\alpha_i^2} \le a_{i-2} \le 2^{\mathrm{pl}(a_{i-2})} \cdot x\alpha_i^2 \le x\alpha_i^3 \tag{19}$$

and

$$\frac{x}{\alpha_i^3} \le \frac{1}{2^{\mathrm{pl}(b_i)}} \cdot \frac{x}{\alpha_i^2} \le b_i \le 2^{\mathrm{pl}(b_i)} \cdot x\alpha_i^2 \le x\alpha_i^3 \tag{20}$$

By Lemma 18, we have

$$\frac{1}{2^{\mathrm{pl}(a_{i+1})}} \leq \frac{a_{i+1}}{a'_{i+1}} \leq 2^{\mathrm{pl}(a_{i+1})}$$

which implies

$$\frac{x}{\alpha_i^4} \leq \frac{1}{2^{\mathrm{pl}(a_{i+1})}} \cdot \frac{x}{\alpha_i^3} \leq a_{i+1} \leq 2^{\mathrm{pl}(a_{i+1})} \cdot x\alpha_i^3 \leq x\alpha_i^4. \tag{21}$$

We proceed as follows. For $1 < i < z$, we have

$$\begin{aligned}
F(A_i) &= \log \frac{w(\max(A_{i-1})) + w(\min(A_{i+1})) + \sum_{x\in A_i} w(x)}{\min(w'(\max(B_{i-1})), w'(\min(A_i)), w'(\max(A_i)), w'(\min(B_i)))} && \text{by definition of } F(\cdot).\\
&\leq \log \frac{a_{i-2} + a_{i-1} + w(\min(A_{i+1})) + \sum_{x\in A_i} w(x)}{\min(w'(\max(B_{i-1})), w'(\min(A_i)), w'(\max(A_i)), w'(\min(B_i)))} && \text{by definition of } w(\cdot).\\
&\leq \log \frac{a_{i-2} + a_{i-1} + a_i + a_{i+1} + \sum_{x\in A_i} w(x)}{\min(w'(\max(B_{i-1})), w'(\min(A_i)), w'(\max(A_i)), w'(\min(B_i)))} && \text{by definition of } w(\cdot).\\
&\leq \log \frac{a_{i-2} + a_{i-1} + a_i + a_{i+1} + 2b_{i-1}}{\min(w'(\max(B_{i-1})), w'(\min(A_i)), w'(\max(A_i)), w'(\min(B_i)))} && \text{by definition of } w(\cdot).\\
&\leq \log \frac{a_{i-2} + a_{i-1} + a_i + a_{i+1} + 2b_{i-1}}{\min(b'_{i-1}, a''_{i-1}, a'_i, b''_{i-1})} && \text{by definition of } w(\cdot).\\
&= \mathcal{O}\left(\log \frac{x\alpha_i^4}{\min(b'_{i-1}, a''_{i-1}, a'_i, b''_{i-1})}\right) && \text{by (19),(15),(17), and (21).}\\
&= \mathcal{O}\left(\log \frac{x\alpha_i^4}{x/\alpha_i}\right) && \text{by (9).}\\
&= \mathcal{O}(\log \alpha_i).
\end{aligned}$$

The proof of the second equality, $F(B_j) = \mathcal{O}(\log \beta_j)$ for $1 < j < v$, is analogous.

□

Lemma 20. *For $1 < i < z$ and $1 < j < v$, we have*

$$7 \cdot \left(\sum_i \log 2^{pl(a_i)} + \sum_j \log 2^{pl(b_j)}\right) > \left(\sum_i \log \alpha_i + \sum_j \log \beta_j\right).$$

Proof. Observe that a gap a_k can be mapped to at most seven times by unique α_i's and β_j's; namely only by $\alpha_{k-1}, \beta_{k-1}, \alpha_k, \beta_k, \alpha_{k+1}, \beta_{k+1}, \alpha_{k+2}$. Similarly, a gap b_k can be mapped to at most seven times by unique α_i's and β_j's; namely only by $\beta_{k-1}, \alpha_k, \beta_k, \alpha_{k+1}, \beta_{k+1}, \alpha_{k+2}, \beta_{k+2}$. The lemma follows. □

We are now ready to bound the amortized time complexity of the MERGE operation.

Theorem 21. *The* MERGE(A, B) *operation has an amortized time complexity of* $\mathcal{O}(\log n)$.

Proof. We will analyze the MERGE operation using the potential method [14]. Recall that D_i represent the data structure after operation i, where D_0 is the initial data structure. The amortized cost of operation i is $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$. Then the amortized cost of the MERGE(A, B) operations is

$$
\begin{aligned}
\hat{c_i} &= c_i + \Delta\Phi \\
&\leq \kappa_e \left(\log n + \sum_{i=2}^{z-1} F(A_i) + \sum_{j=2}^{v-1} F(B_j) \right) + \Delta\Phi && \text{by Theorem 16.} \\
&= \mathcal{O}(\log n) + \kappa_e \left(\sum_{i=2}^{z-1} F(A_i) + \sum_{j=2}^{v-1} F(B_j) \right) + \Delta\Phi \\
&= \mathcal{O}(\log n) + \kappa_e \left(\sum_{i=2}^{z-1} F(A_i) + \sum_{j=2}^{v-1} F(B_j) \right) - \kappa_d \cdot \left(\sum_{i=0}^{z} \mathrm{pl}(a_i) + \sum_{j=0}^{v} \mathrm{pl}(b_j) \right) && \kappa_d \text{ comes from } \Phi. \\
&= \mathcal{O}(\log n) + \kappa_e \left(\sum_{i=2}^{z-1} F(A_i) + \sum_{j=2}^{v-1} F(B_j) \right) - \kappa_d \cdot \left(\sum_{i=2}^{z-1} \mathrm{pl}(a_i) + \sum_{j=2}^{v-1} \mathrm{pl}(b_j) \right) && \text{absorbed by } \mathcal{O}(\log n). \\
&< \mathcal{O}(\log n) + \kappa_e \left(\sum_{i=2}^{z-1} F(A_i) + \sum_{j=2}^{v-1} F(B_j) \right) - \frac{\kappa_d}{7} \cdot \left(\sum_{i=2}^{z-1} \log \alpha_i + \sum_{j=2}^{v-1} \log \beta_j \right) && \text{by Lemma 20.} \\
&\leq \mathcal{O}(\log n) + \kappa_f \left(\sum_{i=2}^{z-1} \log \alpha_i + \sum_{j=2}^{v-1} \log \beta_j \right) - \frac{\kappa_d}{7} \cdot \left(\sum_{i=2}^{z-1} \log \alpha_i + \sum_{j=2}^{v-1} \log \beta_j \right) && \text{by Lemma 19.} \\
&\leq \mathcal{O}(\log n) && \text{Set } (\kappa_d = 7\kappa_f).
\end{aligned}
$$

□

We can now state our main theorem.

Theorem 22. *The Mergeable Dictionary executes a sequence of m* Find, Search, Split, *and* Merge *operations in worst-case $\mathcal{O}(m \log n)$ time.*

Proof. Follows directly from Theorem 9 and Theorem 21. □

References

[1] Pankaj K. Agarwal, Herbert Edelsbrunner, John Harer, and Yusu Wang. Extreme elevation on a 2-manifold. *Discrete & Computational Geometry*, 36(4):553–572, 2006.

[2] Amitabha Bagchi, Adam L. Buchsbaum, and Michael T. Goodrich. Biased skip lists. *Algorithmica*, 42(1):31–48, 2005.

[3] Samuel W. Bent, Daniel Dominic Sleator, and Robert Endre Tarjan. Biased search trees. *SIAM Journal on Computing*, 14(3):545–568, 1985.

[4] Mark R. Brown and Robert Endre Tarjan. A fast merging algorithm. *J. ACM*, 26(2):211–226, 1979.

[5] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Adaptive set intersections, unions, and differences. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 743–752, 2000.

[6] Martin Farach and Mikkel Thorup. String matching in lempel-ziv compressed strings. *Algorithmica*, 20(4):388–404, 1998.

[7] Loukas Georgiadis, Haim Kaplan, Nira Shafrir, Robert Endre Tarjan, and Renato Fonseca F. Werneck. Data structures for mergeable trees. *CoRR*, abs/0711.1682, 2007.

[8] Loukas Georgiadis, Robert Endre Tarjan, and Renato Fonseca F. Werneck. Design of data structures for mergeable trees. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 394–403. ACM Press, 2006.

[9] Katherine Jane Lai. Complexity of union-split-find problems. Master's thesis, Massachusetts Institute of Technology, 2008. Adviser: Erik Demaine.

[10] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, June 1990.

[11] Mihai Pătraşcu and Erik D. Demaine. Lower bounds for dynamic connectivity. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC)*, pages 546–553, 2004.

[12] Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

[13] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[14] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, April 1985.