Range Minimum Queries in Minimal Space

Luís M. S. Russo
luis.russo@tecnico.ulisboa.pt
INESC-ID and Department of Computer Science and Engineering,
Instituto Superior Técnico, Universidade de Lisboa.

Abstract

We consider the problem of computing a sequence of range minimum queries. We assume a sequence of commands that contains values and queries. Our goal is to quickly determine the minimum value that exists between the current position and a previous position i. Range minimum queries are used as a sub-routine of several algorithms, namely related to string processing. We propose a data structure that can process these command sequences. We obtain efficient results for several variations of the problem, in particular we obtain $O(1)$ time per command for the offline version and $O(\alpha(n))$ amortized time for the online version, where $\alpha(n)$ is the inverse Ackermann function and n the number of values in the sequence. This data structure also has very small space requirements, namely $O(\ell)$ where ℓ is the maximum number of active i positions. We implemented our data structure and show that it is competitive against existing alternatives. We obtain comparable processing time, in the nanosecond range, and much smaller space requirements.

Keywords: Range Minimum Queries, Union-Find, Disjoint Sets, Bulk Queries, String Processing.

1 The Problem

Given a sequence of integers, usually stored in an array A, a range minimum query (RMQ) is a pair of indices (i, j). We assume that $i \leq j$. The solution to the query consists in finding the minimum value that occurs in A between the indices i and j. Formally, the solution is $\min\{A[k] | i \leq k \leq j\}$. There are several efficient solutions for this problem, in this static offline context, see Section 5. In this paper we expand on the work by Alzamel, Charalampopoulos, Iliopoulos, and Pissis [2018], that pointed out that in some cases the number of queries to compute is small compared with the size of A. In this scenario building an index data structure, by preprocessing A, is not necessarily the fastest and most space efficient way to solve the problem. Instead we assume that the elements of A are streamed in a sequential fashion. Likewise we assume that the corresponding queries are intermixed with the values of A and the answers to the operations are computed online. In fact we even consider the real-time case where the data structure is required to be efficient in every single operation. We assume that the input to our algorithm consists of a sequence of the following commands:

`Value` - represented by `V`, is followed by an integer, or float, value v and it indicates that v is the next entry of A, i.e., $A[j] = v$.

`Query` - represented by `Q`, is followed by an integer that indicates a previous index of the sequence. The given integer corresponds to the element i in the query. The element j is the position of the last given value of A. Hence it is only necessary to specify i. This command can only be issued if an `M` command (defined right below) was given at position i and no close command was given with argument i.

`Mark` - represented by `M`, indicates that future queries may use the current position j as element i, i.e., as the beginning of the query.

`Close` - represented by `C`, is also followed by an integer i that represents an index of the sequence. This command essentially nullifies the effect of an `M` command issued at position i. Hence the command indicates that the input contains no more queries that use i. Any information that is being kept about position i can be purged.

For simplicity we assume that the sequence of commands is regular enough that we do not have to consider pathological sequences. Examples of such problems would be: issuing the `Mark` command twice or more or mixed with `Query`; issuing `Mark` commands for positions that have been closed; issuing a `Close` or `Query` command for an index i that was not marked or is already closed. In some configurations of our data structure it is possible to detect these inconsistencies, we explain how at the end of Section 2.

Consider the following example sequence. We will use this sequence throughout the paper.

```
V 22 M V 23 M V 26 M V 28 M V 32 M V 27 M V 35 M V 35 M Q 4 C 3
```

In this paper we study this type of sequences, as they are a natural extension on the approach by Alzamel, Charalampopoulos, Iliopoulos, and Pissis [2018]. Our contributions are the following:

- We propose an algorithm that can efficiently process this type of input sequences. We show that our algorithm produces the correct solution, see Theorem 1.
- We analyze the algorithm and show that it is very fast and requires only a very small amount of space. Specifically the space requirements are shown to be $O(m)$, where m is the number of `Mark` commands. Recall that we do not store the array A. We establish an even smaller bound. Consider at some instant the number of marked positions that have not yet been closed. We refer to these positions as active. The maximum number of active positions over all instants is ℓ. We further bound the space requirements by $O(\ell)$. The query time is shown to be $O(1)$ in the offline version of the problem and $O(\alpha(\ell))$ on the online version, where α is the inverse Ackermann function, see Theorem 2 and Corollary 2 in Section 3.2. We also discuss the use of this data structure for real-time applications. We obtain a high probability $O(\log n)$ time for all operations, Theorem 3. We also discuss a trade-off that reduces this bound to $O(\log \log n)$ for some operations, Theorem 4.

- We implemented the online version of our algorithm and show experimentally that it is very efficient both in time and space. In particular in time it executes in the nanosecond range, meaning the computation overhead is only a factor of the time necessary to stream the input. Likewise the space requirements are significantly smaller than the size necessary to store the underlying array A.

2 Data Structure Outline

Let us now discuss how to solve this problem, by gradually considering the challenge at hand. In this section we describe a simple combination of data structures that obtains $O(m)$ space and $O(\alpha(m))$ amortized expected time for the online version of the problem. In the next section we further reduce the space requirements to $O(\ell)$ and discuss other time trade-offs.

Consider again the sequence in Section 1. Our first data structure is a stack. The process is simple. We start by pushing a $-\infty$ value into the stack, this value will be used as a sentinel. To start the discussion we will assume, for now, that every `Value` command is followed by a `Mark` command, meaning that every position is relevant for future queries.

An important invariant of this stack is that the values form an increasing sequence, we prove this property in Lemma 3. Whenever a value is received it is compared with the top of the stack. While the value at hand is smaller the stack gets popped. At some point the input value will be larger than the top of the stack, even if it is necessary to compare with the sentinel. When the input value is larger than the top value it gets pushed into the stack. An important invariant of this data structure is that S contains the solutions of all range minimum queries (i, j), where j is the current position of the sequence being processed and i is some previous position. We show this property in Theorem 3.

To identify the corresponding stack position it is useful to keep, associated to each stack item, the set of active positions that yield the corresponding item as the RMQ solution. Maintaining this set of positions is fairly simple. Whenever an item is inserted into the stack it is inserted with the current position. We number positions by starting at 1. When an item is popped from the stack the set of active positions associated to that item is transferred into the set of active positions of the item below it. In our example the `Value` 27 command puts the positions 4 and 5 into the same set. To process a `Close` command we remove the corresponding position from whatever set it belongs to, i.e., command `C` followed by i removes i from a position set.

Figure 1 illustrates the configuration of this data structure as it processes the following sequence of commands:

```
V 22 M V 23 M V 26 M V 28 M V 32 M V 27 M V 35 M V 35 M Q 4 C 3
```

Each gray rectangle shows a different configuration. The leftmost configuration is obtained after the `V 32 M` commands. The middle configuration after the `V 27 M` commands. The rightmost configuration after the `C 3` command. The solution to the `Q 4` command is 27, because it is the stack item associated with the position 4 in the rightmost configuration, these values are highlighted in bold. Also note that we do not want to keep duplicated values in the stack. This is illustrated with the repetitions of `V 35 M` commands. Only one instance

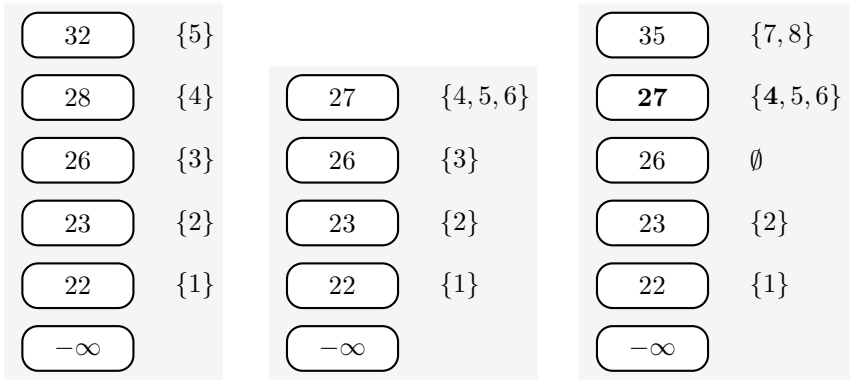

Figure 1: Illustration of structure configuration at different instances. Each gray rectangle shows the stack on the left and the corresponding sets of positions on the right.

of the value 35 is kept on the stack, but both positions 7 and 8 are associated with it.

Using a standard stack implementation it is possible to guarantee $O(1)$ time for the push and pop operations. Hence, ignoring the time required to process the sets of positions, the pairs of `Value` and `Mark` operations require only constant amortized time to compute. In the worst case a `Value` operation may need to discard a big stack, i.e., it may require popping $O(n)$ items, where n is the total number of positions in A. Since each operation executes at most one push operation, the amortized time becomes $O(1)$. Hence the main challenge for this data structure is how to represent the sets of positions. To answer this question we must first consider how to compute the `Query` operation. Given this command, followed by a value i, we proceed to find the set that contains i and report the corresponding stack element. For example to process the `Q` 4 command in the input sequence we most locate the set that contains position 4. In this case the set is $\{4, 5, 6\}$ and the corresponding element is 27. Hence the essential operations that are required for the sets of positions are the union and the find operations. Union is used when merging sets in the `Mark` operation and find is used to identify sets in the `Query` operation.

Hence our functional requirements can be supported by a Union-Find (UF) data structure. This data structure can be used to maintain a collection of sets. The union operation is supported. Once united the sets can no longer be separated. The find operation is also supported, it allows us to determine which set a give element belongs to. A naive implementation requires $O(n)$ time for each operation. Instead we use a dedicated data structure that supports both operations in $O(\alpha(n))$ amortized time, where $\alpha(n)$ is the inverse Ackermann function. The only other requirement for the efficient performance of this data structure is that the sets in question must be disjoint, which is also the case in our application. An elementary description of this data structure was given by Cormen, Leiserson, Rivest, and Stein [2009] and Sedgewick and Wayne [2011].

Although conceptually the `Close` command removes elements from the position sets our data structure ignores these operations. Essentially because we can not efficiently remove elements from the UF data structure. Hence, once

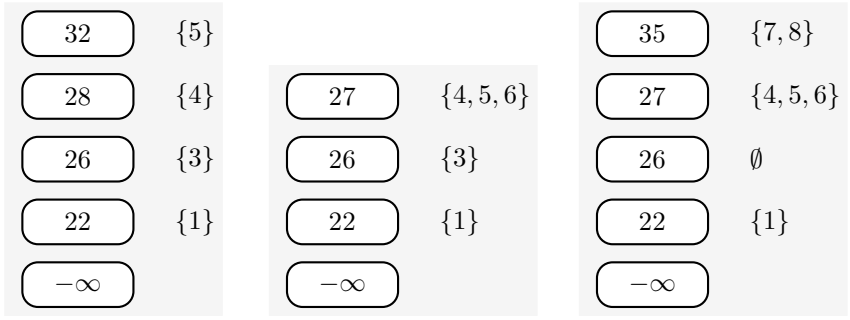

Figure 2: Illustration of structure configuration at different instances. In this sequence of commands there is no `M` command after `V 23`. Each gray rectangle shows the stack on the left and the corresponding sets of positions on the right.

an element is assigned to a set, it can no longer be removed. Fortunately the resulting procedure is still sound, albeit it requires more space. This version does require a large amount of space, specifically $O(n)$ space.

Let us now focus on reducing the space to $O(m)$, where m is the total number of `Mark` commands, which should be greater than or equal to the total number of `Close` commands. Note that m may be much smaller than n as there might be many more `Value` commands than `Mark` commands. Hence bounding the size of the stack by $O(m)$ is better than by $O(n)$. This is achieved by noticing that we only need to store a value in the stack if it is related to a marked position. Hence our previous insertion procedure gets divided into two sub-routines. The `Mark` command pushes the last value into the stack, unless it is there already. The `Value` command pops the larger elements and overwrites the last one, unless the top of the stack is smaller than the new value, in which case it does nothing. Hence in this scenario both the `Mark` and `Value` commands require $O(\alpha(n))$ amortized time.

Let us show an example of the division we have just described. Consider the following sequence of commands:

```
V 22 M V 23 V 26 M V 28 M V 32 M V 27 M V 35 M V 35 M Q 4 C 3
```

We illustrate the state of the resulting data structure in Figure 2. Notice that in this sequence there is no `M` command after `V 23`. Therefore this value never gets inserted into the stack.

To reduce the size of the UF data structure we add a hash table h to it. Without this table every one of the n position values are elements for the UF data structure. Using a hash we can filter out only the marked positions. When a `Mark` command is issued we use the current position j as a hash key and insert the current number of UF elements as the corresponding value, denoted as $h(j)$. This reduces the size of the UF data structure to $O(m)$. Moreover the hash table also requires only $O(m)$ space. The time bound for accessing, inserting, and removing elements from a hash table is $O(\log m)$ with high probability or $O(1)$ in the expected case, given that we keep the load factor below 50%, see Section 5. Hence this data structure requires only $O(m)$ space and can process any sequence of commands in $O(\alpha(m))$ expected amortized time per command, see Theorem 2, or in $O(\log(m))$ time with high probability, see Theorem 3.

When a `Close` i command is issued we mark the key i as inactive in the hash table, but we do not actually remove it from memory. The reason for this process is that a stack item might actually point to position i and removing it would break the data structure. In other words $h(i)$ might be the representing element of a disjoint set. For the $O(m)$ space bound this is not an issue as inactive markings are accounted for by m.

Another important advantage of having a hash table is the possibility of detecting pathological command sequences. First let us consider the simple sequences that can be solved even without a hash table. To avoid `Mark` commands that are repeated twice or more or mixed with `Query` we can keep a state variable t that is associated with the current position j. Every time a `Value` command is issued this variable is set to the unmarked state, because the value of j changes. If a `Mark` command is processed when t is in this state then the sequence is valid, and t is set to marked. If t was already in the marked state then the sequence is invalid. If the `Close` command is called with $i = j$ this variable changes to the closed state. Hence if a `Mark` command is issued when j is in the closed state the sequence is also invalid. This later case identifies sequences that issue `Mark` commands for a position that has been closed.

Now let us consider the problems that can be detected with a hash table. Issuing a `Close` or `Query` command for a positin i that has not been marked or is alredy closed. Every time a `Mark` command is issued the corresponding position j is added to the hash table. Hence before processing a `Close` or `Query` command we search for the corresponding position i in the hash table. If the position is not on the table than the sequence is invalid. Moreover it may be that the position is on the table but is marked inactive. In this case the sequence is also invalid. These inactive markings are set by the `Close` commands, as discussed above.

In the next section we discuss several nuances of this data structure, including how to further reduce the space requirements to $O(\ell)$ space and alternative implementations. Note that the patological sequence detection procedures we have just discussed can also be used in this reduced space. The only critical point is that this process will discard inactive keys from the hash table, but since missing keys also trigger an invalid sequence detection the procedures are still valid.

3 The Details

In this Section we will prove that the algorithm is correct and analyze its performance. We start off by giving a pseudo code description of the algorithms used for each command, Algorithms 3, 4, 5 and 6. In these algorithms we make some simplifying assumptions and use some extra commands that we will now define. A summary of these assumptions and notation is given in Table 1.

For simplicity we describe the data structure that does not use a hash-table. We use S to represent the stack data structure, but we also use $S[k']$ to reference the element at position k'. Hence S is actually an augmented stack. Stacks usually do not support this lookup operation. This is easy to obtained, in $O(1)$ time, by using an array implementation. Note that this extension does not disrupt the expected stack behavior because it is generally used as a read only operation. The only exception is the overwrite in line 7 of Algorithm 5,

however this operation overwrites the top element of the stack and therefore it is equivalent to a sequence of `Pop` and `Push` operations. We will also assume, for now, that this array contains enough space to hold all the necessary elements. This restriction is lifted in Section 3.2.

In general the top of the stack is at position k, which also corresponds to the number of elements in the stack. We use k as a global variable. We also use k as a bounded variable in the lemma statements. Hence the value of k must be derived from context. This is usually not a problem and in fact it is handy for the proofs, which most of the time only need to consider when k is the top of the stack. We use the notation `Top`() to refer to the element at top of the stack, i.e., $S[k]$. Note that this means that the element $S[k-1]$ is just below the `Top`() element. Algorithms 1 and 2 are used to manipulate the stack status and are given for completion. Another important detail about our stack is its initial configuration. We mentioned in Section 2 that we assumed that S contained an $-\infty$ value as its lowest element. However, for the particular pseudo-code we give in Algorithm 5, this is not enough. In particular that algorithm does not work properly when this sentinel becomes the `Top`() element. Hence to circumvent this issue we add yet another sentinel element, with value $+\infty$. Therefore in the initial configuration the stack S contains only these two values and the `Top`() element is $+\infty$. This second sentinel is overwritten in the first call to a `Value` command, which should be the first command in any sequence. The main point of this comment is to highlight that the pseudo-code we give in Algorithm 5 assumes that S always contains at least two elements, which is a property we then prove in Lemma 1.

The set of positions associated with each stack item is denoted with the letter P. In our example we have that $P[4] = \{4, 5, 6\}$, see Figure 1. More precisely we have that $P[k']$ is the set of active positions that corresponds to $S[k']$. As mentioned in Section 2 an important property of these sets is that they are disjoint , i.e., if $k' \neq k''$ then $P[k'] \cap P[k''] = \emptyset$. In Algorithm 3 we assume that the result of the `Find` command is directly a position index of S, so that we may use the expression $S[\texttt{Find}(i)]$ in Algorithm 3. More precisely the expected behavior of the `Find`(i) function is to return the value k' such that whenever i is an active position we have $i \in P[k']$. This is also a simplification as in general the `Find` operation returns a representative element from the corresponding set. For example if the representative of the set $\{4, 5, 6\}$ is 5 then the `Find`(4) command would return 5. Instead we assume that `Find`(4) returns k', such that $S[k'] = 27$, in the middle and right configurations of Figure 1, or $S[k'] = 28$ in the left configuration. This has the advantage that we do not have to explicitly keep track of a map between the representatives and the positions in S. On the other hand we need to provide these k' positions as arguments to the `Make-Set` and `Union` operations. The `NOP` command does nothing, it is used to highlight that without a hash table there is nothing for the `Close`(i) command to execute. From the theoretical perspective this causes the value i to be removed from the corresponding $P[k']$ set.

The `Make-Set`(j, k') function is used to create a set in the UF data structure, the first argument indicates the element that is stored in the set, position j. The second argument indicates its position on the stack S. It is the value given in this second argument that we expect the `Find` operation to returns.

Likewise the `Union` function receives three arguments, the sets that we want to unite and the resulting stack position k'. Note that in Algorithm 6 we use $\{j\}$

A	Array of values
i	Index for RMQ
j	Index for RMQ, also current index over A
$v = A[j]$	Last argument given to the `Value` command
S	Augmented stack
k	Number of elements in S, also the index of the top element
$S[k']$	Element in position k' in S
`Top`$() = S[k]$	Top element in S
`Pop`$()$	Remove the top element in S
`Push`(v)	Insert value v into the top of S
$P[k']$	Set of active positions that corresponds to $S[k']$
`Find`(i)	Whenever i is an active position it should return the value k' such that $i \in P[k']$
`Make-Set`(j, k')	Creates a singleton set containing j, so that `Find`(j) returns k'
`Union`$(\mathcal{A}, \mathcal{B}, k')$	Unites sets $\mathcal{A}$ and $\mathcal{B}$ so that `Find`(j) returns k' for any active positions $j \in \mathcal{A} \cup \mathcal{B}$.

Table 1: Notation, APIs, and global variables. The elements in the top section are related to A, the elements in the middle section to S and the elements in the bottom section to the UF data structure.

as one of the arguments to `Union` operation. In this case we are assuming that this operation makes the corresponding `Make-Set` operation, moreover in this case the second arguments of `Make-Set` is not relevant as it will be immediately replaced by k'.

Besides k we use j and v as global variables. Variable j indicates the current position in A. Variable v is the argument given in the last `Value` command. Both these variables are used in the `Mark` command, which therefore has no arguments.

Algorithm 1

```
1: procedure PUSH(v)                          ▷ Insert element into stack
2:     k ← k + 1
3:     S[k] = v
4: end procedure
```

Algorithm 2

```
1: procedure POP                              ▷ Remove element from stack
2:     k ← k − 1
3: end procedure
```

3.1 Correctness

In this Section we establish that our algorithm is correct, meaning the values obtained from our data structure actually correspond to the solutions of the

Algorithm 3

```
1: procedure QUERY(i)                                   ▷ Return RMQ (i, j)
2:     return S[Find(i)]
3: end procedure
```

Algorithm 4

```
1: procedure CLOSE(i)                                   ▷ Ignore command
2:     NOP
3: end procedure
```

Algorithm 5

```
1: procedure VALUE(v)                                   ▷ Squeeze stack
2:     if S[k] > v then                                 ▷ Test element at the Top().
3:         while S[k − 1] ≥ v do                        ▷ Test element below the Top().
4:             Union(P[k], P[k − 1], k − 1)             ▷ Unite top position sets.
5:             Pop()
6:         end while
7:         S[k] = v
8:     end if
9:     j ← j + 1
10: end procedure
```

Algorithm 6

```
1: procedure MARK                                       ▷ Put v into the stack
2:     if S[k] < v then
3:         Push(v)                                      ▷ Insert v into S.
4:         Make-Set(j, k)                               ▷ Associate with k.
5:     else
6:         Union(P[k], {j}, k)                          ▷ Assume it calls Make-Set.
7:     end if
8: end procedure
```

given range minimum queries. We state several invariant properties that the structure needs to maintain.

We consider the version of the data structure that consists of a stack and a UF structure. The version containing a hash is relevant for obtaining a small structure but does not alter the underlying operation logic. Hence the correctness of the algorithm is preserved, only its description is more elaborate.

We prove the invariant properties by structural induction, meaning that we assume that they are true before a command is processed and only need to prove that the property is maintained by the corresponding processing. For this kind of argument to hold it is necessary to verify that the given properties are also true when the structure is initialized, this is in general trivially true so we omit this verification from the following proofs. Another simplifying observation is that the `Query` and `Close` commands do not alter our data structure and therefore are also omitted from the following proofs.

Let us start by establishing some simple properties.

Lemma 1. *The stack S always contains at least two elements.*

Proof. In this particular proof it is relevant to mention the initial state of the stack S. The stack is initialized with two sentinel values, $-\infty$ followed by $+\infty$. Hence it initially contains at least two elements.

- The `Mark` command. This command does not use the `Pop` operation and therefore never reduces the number of elements. The result follows by induction hypothesis.
- The `Value` command. For the `Pop` operation in line 5 of Algorithm 5 to execute the **while** guard in line 3 must be true. Note that when $k = 2$ this guard consists in testing whether $-\infty = S[1] \geq v$, which is never the case and therefore a `Pop` operation is never executed in a stack that contains 2 elements.

□

Lemma 2. *If v was the argument of the last `Value` command and k is the top level of that stack S then $S[k] \leq v$.*

Proof.

- The `Mark` command. When the **if** condition of Algorithm 6 is true we have that line 3 executes. After which $S[k] = v$ and the lemma's conclusion is verified. Otherwise the **if** condition is false and the stack is kept unaltered, in which case the result follows by induction hypothesis.
- The `Value` command. When the **if** condition of Algorithm 5 fails the lemma property is immediate. Hence we only need to check the case when the **if** condition holds. In this case line 7 must eventually execute at which point we have that $S[k] = v$ and the lemma's conclusion is verified.

□

Let us now focus on more global properties. Next we show that the values stored in S are in increasing order.

Lemma 3. *For any indexes k and k' of the stack S we have that if $k' < k$ then $S[k'] < S[k]$.*

Proof.

- The `Value` command. This command does not push elements into the stack, instead it pops elements. This means that, in general, a few relations are discarded. The remaining relations are preserved by the induction hypothesis. The only change that we need to verify is if the `Top`() of the stack S changes, line 7 of Algorithm 5. Hence we need to check the case when k is the top level of the stack. Note that line 7 occurs immediately after the **while** cycle, which means that its guard is false, i.e., we have that $S[k-1] < v = S[k]$. Hence the desired property was established for $k' = k - 1$. For any other $k' < k - 1$ we can use the induction hypothesis to conclude that $S[k'] < S[k-1]$, which combined with the previous inequality and transitivity yields the desired property that $S[k'] < S[k]$.
- The `Mark` command. The only operation performed by this command is to push the last element into the stack. Hence when k is below the top of the stack the property holds by induction. Let us analyze the case when the top of the stack changes, i.e., when k is the top level of the stack. The change occurs in line 3 of Algorithm 6 in which case we have that $S[k-1] < v = S[k]$. Hence we extend the argument for $k' < k - 1$ as in the `Value` command by induction hypothesis and transitivity.

□

Likewise the converse of this lemma can now be established.

Lemma 4. *For any indexes k and k' of the stack S we have that if $S[k] < S[k']$ then $k < k'$.*

Proof. Assume by contradiction that there are k and k' such that $S[k] < S[k']$ and $k' \leq k$. Because $S[k] \neq S[k']$ we have that $k \neq k'$, since we are using S as an array. Hence we must have that $k' < k$ and can now apply Lemma 3 to conclude that $S[k'] < S[k]$, which contradicts the order relation in our hypothesis. □

This sorted property also gives structure to the sets of positions.

Lemma 5. *For any indexes $k' < k$ and positions $p' \in P[k']$ and $p \in P[k]$ we have that $p' < p$.*

Proof.

- The `Mark` command. This operation inserts the current position j into the set that corresponds to the top of the stack. The top might have been preserved or created by the operation, both cases can be justified in the same way. We only need to consider the case when `Top`() $= S[k]$ and $p = j$, any other instantiating of the variables in the lemma will correspond to relations that were established before the structure was modified. Hence we only need to show that $p' < j$ for any p' in any $P[k']$. This is trivial because j represents the current position in A, which is therefore larger than any previous position of A that may be represented by p'.

- The `Value` command. As this command pops elements from the stack, it has the side effect of merging the position sets. Hence the only new relation is for positions at the top of the stack, i.e., when $p \in P[k]$ and `Top`() $= S[k]$. We only need to consider where position p was before the operation, i.e., $p \in P_b[k_b]$, were $P_b[k_b]$ represents a set of positions before the operation is executed. Because the `Value` command merges the position sets which are highest on the stack we have that $k \leq k_b$. Now, for any $k' < k$ and $p' \in P[k']$, we have that $P[k'] = P_b[k']$ because the sets of positions below the top of the stack are not altered by the operation. In essence we have that $k' < k_b$ and $p' \in P_b[k']$ and $p \in P_b[k_b]$, therefore by induction hypothesis we obtain $p' < p$, as desired.

□

We can now state our main invariant, which establishes that our algorithm is correct. More precisely it guarantees that the value computed by Algorithm 3 is correct, provided i is an active position.

Theorem 1. *At any given instant when j is the current position over A we have that if $i \in P[k']$ then RMQ$(i, j) = S[k']$.*

Proof.

- The `Mark` command. This command does not alter the sequence A. Therefore none of the RMQ(i, j) values change. Since almost all positions and position sets $P[k']$ are preserved the implication is also preserved. The only new position is $j \in P[k]$, therefore the only case we need to consider is when $i = j$ and k' is the top level of the stack S, i.e., $k' = k$. In this case we have that RMQ$(j, j) = A[j] = v$, where v is the argument given in the last `Value` command. Now let us consider the **if** condition in line 2 of Algorithm 6. This further divides the argument into two cases:
  - When this condition holds then line 3 of Algorithm 6 executes and makes $S[k] = v$. Hence RMQ$(j, j) = S[k]$.
  - When this condition fails we have $v \leq S[k]$. Applying Lemma 2 we obtain $S[k] \leq v$ and therefore conclude that $S[k] = v$. Hence RMQ$(j, j) = S[k]$.
- The `Value` command. This command essentially adds a new value v at the end of A, i.e., it sets $A[j] = v$, where j is now the last position of A. This implies that j is not yet a marked position. Therefore for this command we do not need to consider $i = j$ because j is not a member of a position set $P[k']$.

  Thus we only need to consider the cases when $i < j$. Consider such an index i, which moreover belongs to the position set $P[k']$, i.e., $i \in P[k']$. The position i must necessarily occur in some set $P_b[k'_b]$, which is a set of positions that exists before the `Value` operation alters the stack. In this case we have by induction hypothesis that RMQ$(i, j-1) = S_b[k'_b]$. We now divide the proof into two cases:

– When $S_b[k'_b] \leq v$, in which case $\mathrm{RMQ}(i,j) = S_b[k'_b]$. In this case we only need to show that the `Value` command does not alter the index k'_b of the stack, i.e., that $i \in P[k'_b]$ and that $S_b[k'_b] = S[k'_b]$. Therefore the desired property holds for $k' = k'_b$. This is immediate as the case hypothesis means that even if the `Value` operation happens to extrude level k'_b to the top of the stack it does eliminate it, because Lemma 3 implies that $S_b[k'_b - 1] < S_b[k'_b] \leq v$, and therefore the while guard in line 3 fails.

– When $v < S_b[k'_b]$, in which case $\mathrm{RMQ}(i,j) = v$. In this case the value $S_b[k'_b]$ will be discarded by the `Value` command. Let k correspond to the level that is at the top of the stack, after the command. By Lemma 2 we have that $S[k] \leq v$ combining both these inequalities yields $S[k] < S_b[k'_b]$. Using Lemma 3 we have that $S[k-1] < S[k]$, note that Lemma 1 guarantees that the level $k-1$ exists. Moreover because k is the top level of S after the command we have $S_b[k-1] = S[k-1]$. Combining these relations we obtain that $S_b[k-1] < S_b[k'_b]$, to which we apply Lemma 4, to conclude that $k - 1 < k'_b$. Therefore either $k = k'_b$ or the level k'_b was excluded from the stack. In both cases position i must be in $P[k]$, either because it was already there or it was eventually transferred by the union commands in line 4. Hence we only need to check that $S[k] = v$. Let k_b be the `Top`() of stack S_b before the command is executed. Hence $k'_b \leq k_b$ and by Lemma 3 we obtain $S_b[k'_b] \leq S_b[k_b]$. Using this case hypothesis and transitivity we obtain that $v < S_b[k_b]$. This implies that the condition of the **if** in line 2 of Algorithm 5 is true. Therefore line 7 eventually executes and obtains the condition $S[k] = v$ as desired.

□

In the next section we analyze the space and time performance of our data structure. Hence we can finish this section by showing that all the values in S correspond to some $\mathrm{RMQ}(i,j)$ values, i.e., there are no unnecessary values in S.

Lemma 6. *At any given instant when j is the current position over A on a sequence with no `Close` commands and k' is any index k' of S that is not a sentinel, then the set $P[k']$ is non-empty.*

Proof.

- The `Mark` command. The induction hypothesis establishes the desired result for all $k' < k$. Hence the only set that we need to consider is $P[k]$. This set is guaranteed to contain at least j. When the **if** condition in line 2 of Algorithm 6 holds this set is created with only the position j. On the other hand if this condition fails the previous set $P_b[k]$ gets united with the set $\{j\}$ to form $P[k]$. Hence in both cases we conclude that $j \in P[k]$. Therefore $P[k]$ is non-empty.

- The `Value` command. Again the induction hypotheses guarantees that all the $P[k']$ sets are non-empty, for $k' < k$. The set $P[k]$ results from uniting the previous set $P_b[k]$ with the sets above it, line 4. By induction

hypothesis the set $P_b[k]$ is non-empty, therefore the set $P[k]$ is also non-empty.

□

We can now combine this information with Theorem 1.

Corolary 1. *At any given instant when j is the current position over A on a sequence with no* `Close` *operations and k' is any index of S that is not a sentinel, then $S[k'] = RMQ(i, j)$, for some position i that received a* `Mark` *command.*

Proof. In these conditions we can conclude from Lemma 6 that $P[k']$ is not empty. Therefore $P[k']$ must contain some position i, that was inserted by some `Mark` command. We can now apply Theorem 1 to obtain that $S[k'] = \text{RMQ}(i, j)$. □

Note that this result can be used to bound the size of S by $O(m)$, because the position sets are disjoint and contain only marked positions. Also notice that the fact that we restrict our analysis to sequences without `Close` commands is not a severe limitation. Striping these commands from a sequence still yields a valid sequence, moreover the resulting sequence has the same number of `Mark` commands, i.e., the same m value. Hence the bound by $O(m)$ on S applies to any sequence of commands.

3.2 Analysis

In this section we discuss several issues related to the performance of our data structure. Namely we start off by reducing the space requirements from $O(m)$ to $O(\ell)$. First we need to notice in which ways our data structure can waste space. In particular the `Close` command wastes space in the stack itself. In the rightmost structure of Figure 1 we have that the set $P[3]$ becomes empty after the `C 3` command. This set corresponds to $S[3] = 26$ on the stack. In essence the item $S[3]$ is no longer necessary in the stack. However it is kept inactive in the stack, the hash table and the UF data structure. It is marked as inactive in the hash table, but it still occupies memory.

Recall that our data structure consists of three components: a stack, a hash table and a Union-Find data structure. These structures are linked as follows: the stack contains values and pointers to the hash table; the hash-table uses sequence positions as keys and UF elements as values; the Union-Find data structure is used to manipulate sets of reduced positions and each set in turn points back to a stack position.

Let us now use an amortizing technique to bound the space requirements of this structure. We start off by allocating a data structure that can contain at most a elements, where a is a small initial constant. Allocating a structure with this value implies the following guarantees:

- It is possible to insert a elements into the stack without overflow, i.e., the size of the underlying array is at least a.
- It is possible to insert a elements into the hash table and the load factor is always less than half. This guarantees average and high probability efficient insertions and searches.

- It is possible to use a positions for Union-Find operations.

Hence we can use this data structure until we reach the limit a. When the limit is reached we consider the number of currently active marked positions, i.e., the number of positions i such that `M` was issued at position i, but up to the current position no `Close` i was never issued. To determine this value it is best to keep a counter c. This counter is increased when a `Mark` command is issued, unless the previous command was also a Mark command, in which case it is a repeated marking for a certain position. The counter is decreased when a `Close` i is issued, provided position i is currently active, i.e., it was activated by some `Mark` command and it has not yet been closed by any other `Close` command. Hence by consulting this counter c we can determine in $O(1)$ time the number of active positions at this instant. We can now allocate a new data structure with $a' = 2c$, i.e., a data structure that can support twice as many elements as the number of current active positions. Then we transfer all the active elements from the old data structure to the new data structure. The process is fairly involved, but in essence it requires $O(a \times \alpha(a))$ time and when it finishes the new data structure contains all the active positions, which occupy exactly half of the new data structure. This factor is crucial as it implies that the amortized time of this transfer is in fact $O(\alpha(a))$ and moreover that the allocated size is $O(2\ell)$.

We now describe how to transfer only the active elements from the old data structure to the new data structure. First we mark all the elements in the old stack as inactive. In our implementation we make all the values negative, as the test input sequences contained no negative values but other marking schemes may be used. This is also the scheme we used to mark inactive hash entries.

Now traverse the old hash table and copy all the active values to the new hash table. Also initialize the pointers from the new hash table to the new UF data structure. The new UF positions are initialized incrementally, starting at 1. Hence every insertion into the new hash function creates a new UF position, that is obtained incrementally from the last one. We also look up the old UF positions that are given by active entries of the old hash table. We use those old active sets to reactivate the old stack entries. This process allowed us to identify which stack entries are actually relevant in the old stack. With this information we can compact the old stack by removing the inactive positions. We compact the old stack directly to the new stack, so the new stack contains only active positions. We also add pointers from the old stack to the new stack. Each active entry of the old stack points to its correspondent in the new stack. In our implementation this was done by overriding the pointers to the old hash table, as they are no longer necessary.

At this point the new stack contains the active values, but it still has not initialized the pointers to the new hash table. These pointers are in fact position values, because positions are used as keys in the hash-table. To initialize these pointers we again traverse the active entries of the old hash table and map them to the old UF positions and to the corresponding old stack items. We now use the pointer from the old stack item to the new stack item and update the position pointer of the new stack to the key of the active entry of the new hash that we are processing. This assignment works because positions are kept invariant from the old data structure to the new one. Therefore these positions are also keys of the new hash. We finish this process by updating the pointers

of the new UF data structure to point to the corresponding items of the new stack. Since we now know the active items in the new stack and have pointers from the new stack to the new hash and from the new hash to the new UF position, we can simply assign the link from the new UF set back to the item of the new stack item. Thus closing this reference loop.

At this point almost all of the data structure is linked up. The new stack points to the new hash table, the new hash table points to the new UF structure and the sets of the new UF structure point to the new stack. The only missing ingredient is that the sets of the new UF structure are still singletons, because no `Union` operations have yet been issued. The main observation to recover this information is that several positions in the new UF structure point to the same item in the new stack. Those positions need to be united into the same set. To establish these unions we traverse the new UF data structure. For each UF position we determine its corresponding stack item, note that this requires a Find operation. We then follow its pointer to an item in the new hash, and a pointer from that item back to a position in the new UF data structure. Now we unite two UF sets, the one that contained the initial position and the one that contains the position that was obtained by passing through the stack and the hash.

Theorem 2. *It is possible to process online a sequence of RMQ commands in $O(\ell)$ space using $O(\alpha(\ell))$ expected amortized time per command.*

Proof. The discussion in this section essentially establishes this result. We only need to point out the complexities of the data structures that we are using. As mentioned before the UF structure requires $O(\alpha(n))$ amortized time. The stack is implemented over an array and therefore requires $O(1)$ per `Push` and `Pop` command. In theory we consider a hash-table with separate chaining and a maximum load factor of 50%, which obtains $O(1)$ expected time per operation. In practice we implemented a linear probing approach.

The final argument is to show that the transfer process requires $O(\alpha(\ell))$ amortized time. Whenever a transfer process terminates the resulting structure is exactly half full. As the algorithm progresses elements are inserted into the structure until it becomes full. Whenever an element is inserted we store 2 credits. Hence when the structure is full there is a credit for each element it contains, therefore there are enough credits to amortize a full transfer process. We assume that these credits are actually multiplied by $\alpha(\ell)$ and whatever the constant of the transfer procedure is. □

One important variation of the above procedure is the offline version of the problem. Meaning that we are given the complete sequence of commands and are allowed to process them as necessary to obtain better performance. In this case we can use a more efficient variant of the Union Find data structure and obtain $O(1)$ time per operation, proposed by Gabow and Tarjan [1985].

Corolary 2. *It is possible to process offline a sequence of RMQ commands in $O(\ell)$ space using $O(1)$ expected amortized time per command.*

On the other extreme we may be interested in real time applications. Meaning that we need to focus on minimizing the worst case time that is necessary to process a given command. In this case we can modify our data structure to avoid excessively long operations, i.e., obtain stricter bounds for the worst case

time. As an initial result let us de-amortize the transfer procedure, assuming the same conditions as in Theorem 2.

Lemma 7. *Given a sequence of RMQ commands it is possible to processes them so that the transfer procedures require an overhead of $O(\alpha(\ell))$ expected amortized time per command.*

Proof. Note that the transfer process requires $O(a \times \alpha(a))$ amortized time to transfer a structure that supports a elements.

We modify the transfer procedure so that it transfers two full structures at the same time, by merging their active elements into a new structure. The process is similar to the previous transfer procedure, with a few key differences.

An element can only be considered active if it is not marked as inactive in one of the old hashes. More precisely: if it is marked as active in one hash and as inactive in the other then it is inactive; if it is marked as active in one hash and does not exists in the other then it is active; if it is marked as active in both then it is active.

Once the active elements of the old stacks are identified they are merged into the new stack, by using the same merging procedure that is used in mergeSort algorithm, with the proviso that there should be only one copy of the sentinel in the merged stack. The third important synchronization point is the union commands. Before starting this process it is necessary that all the information from the old structures has been transferred to the new one, recall that this process generally iterates over the new structure, not the old ones.

When the old structures can support a_1 and a_2 elements respectively the merging process requires $O(a_1 + a_2)$ operations. Note that we do not mean time, instead we mean primitive operations on the data structures that compose the overall structure, namely accessing the hash function, following pointers or calling union or find. Given this merging primitive we can now de-amortize our transfer process. Instead of immediately discarding a structure that hits its full occupancy we keep it around because we can not afford to do an immediate transfer. Instead when we have at least two full structures we initiate the transfer process. Again to avoid exceeding real time requirements this process is kept running in parallel, or interleaved, with the processing of the remaining commands in the sequence. Since this procedure requires $O(a_1+a_2)$ operations, it is possible to tune it to guarantee that it is terminated by the time that at most $(a_1 + a_2)/2$ commands are processed. In this case each command only needs to contribute $O(1)$ operations to the merging process. Each operation requires has an expected $O(\alpha(\ell))$ time, which yields the claimed value.

Hence, at any given instant, we can have several structures in memory. In fact we can have at most four, which serve the following purposes:

- One active structure. This structure is the only one that is currently active, meaning that it is the only structure that still supports `Mark` and `Value` commands.
- Two static full structures that are currently being merged.
- One destination structure that will store the result of the merged structures. In general this structure is in some inconsistent state and does not process `Query` commands. The only command that it accepts is `Close`.

At any point of the execution some or all of the previous structures may be in memory. The only one that is always guaranteed to exist is the active structure. Now let us discuss how to process commands with these structures.

- The `Query` command is processed by all structures, except the destination structure which is potentially inconsistent. From the three possible values we return the overall minimum. In this case we are assuming that if the query position i is smaller than the minimum position index stored in the structure than it returns its minimum value, i.e., the value above the $-\infty$ sentinel.
- The `Mark` and `Value` commands modify only the active structure.
- The `Close` command is applied to all the structures, including the destination structure. This causes no conflict or inconsistency. Recall that elements are not removed from the hashes, they are only marked as inactive.

If we have only the active structure in memory, we use it to process the `Mark` and `Value` commands. When this active structure gets full we mark it as static and ask for a new structure that supports the same number a of elements. This structure becomes the new active structure. Note that requesting memory may require $O(a)$ time, assuming we need to clean it. This can be mitigated by using approaches as Briggs and Torczon [1993] or assuming that this process was previously executed, which is possible in our approach.

As soon as the second structure becomes full we start the merging process to a new destination structure. We consult the number of active elements in each one, c_1 and c_2. We request the destination structure to support exactly $c_1 + c_2$ elements. This implies that once the merge procedure is over the destination structure is full and no further elements can be inserted into it. At which point we need to request another active structure. If the full structures have sizes a_1 and a_2 we ask for an active structure that can support $(a_1 + a_2)/2$ elements. As argued above this active structure only gets full after the merging process finishes. At that point the original full structures can be discarded and again we have two full structures, the result of the previous merger and the filled up active structure. Then we repeat the process.

The reason to have a division by 2 associated with a_1+a_2 is that its iteration yields a geometric series that does not exceed 2ℓ. Hence implying that none of the structures need to support more that 2ℓ elements. This can also be verified by induction. Assuming that the original allocated size a is also less than 2ℓ, we have by induction hypothesis that $a_1 \leq 2\ell$ and $a_2 \leq 2\ell$ therefore $(a_1+a_2)/2 \leq (2\ell+2\ell)/2 = 2\ell$. Also by the definition of ℓ we also have that $c_1 \leq \ell$ and $c_2 \leq \ell$ which implies that the destination structures also support at most 2ℓ elements. Since the algorithm uses a at most 4 structures simultaneously, we can thus conclude that the overall space requirements of the procedure is $O(\ell)$. □

Note that in the worst case the time bound of the UF structures is $O(\log \ell)$ rather than $O(\alpha(\ell))$. Also note that using a strict worst case analysis would yield an $O(\ell)$ worst case time for our complete data structure, because it contains a hash-table. To avoid this pathological analysis we instead consider a high probability upper bound. In this context we obtain an $O(\log \ell)$ time bound

with high probability, for all commands except the `Value` command. Hence let us now address this command.

Theorem 3. *It is possible to process, in real time, a sequence of RMQ commands in $O(\ell)$ space and in $O(\log \ell)$ time per operation with high probability.*

Proof. Given the previous observations we can account $O(\log \ell)$ time for the UF structure and the hash table, with high probability, see Mitzenmacher and Upfal [2017]. Lemma 7 de-amortized the transfer operation, hence in this proof we only need to explain how to de-amortize the `Value` operation.

Algorithm 5 specifies that given an argument v this procedure removes from the stack S the elements that are strictly larger than v. This process may end up removing all the elements from the stack, except obviously the $-\infty$ sentinel. Hence its worst case time is $O(m)$, where m is the maximum number of elements in the stack. The transfer procedure guarantees that the stack does not accumulate deactivated items and therefore we have that $m = O(\ell)$. This is still too much time for a real time operation. Instead we can replace this procedure by a binary search over S, i.e., we assume that stack is implemented on an array and therefore we have direct access to its elements in constant time. As shown in Lemma 3 the elements of S are sorted. Therefore we can compute a binary search for the position of v and discard all the elements in S that are larger than v in $O(\log \ell)$ time. Recall that we use variable k to indicate the top of the stack. Once the necessary position is identified we update k.

However Algorithm 5 also specifies that each element that is removed from the stack invokes a `Union` operation, line 4. To perform these unions in real time we need a different UF data structure.

Most UF structures work by choosing a representative element for each set. The representative is the element that is returned by the `Find` operation. This representative is usually an element of the set it represents. The representative either possesses, or is assigned, some distinct feature that makes it easy to identify. In the UF structure by Tarjan and van Leeuwen [1984] a representative is stored at the root of a tree.

Lemma 5 essentially states that the sets that we are interested in can be sorted, without inconsistencies among elements of different sets. Hence this provides a natural way for choosing a representative. Each set can be represented by its minimum element. With this representation the `Find`(p) operation consists in finding the largest representative that is still less than or equal to p, i.e., the Predecessor. The Union operation simply discards the largest representative and keeps the smallest one. Hence we do not require an extra data structure, it is enough to store the minimums along with values within the stack items. To compute the Predecessors we perform a binary search over the minimums. This process requires $O(\log \ell)$ time. Moreover the variable k allows us to perform multiple `Union` operations at once. Let us illustrate how to use this data structure for our goals. Recall the sample command sequence:

```
V 22 M V 23 M V 26 M V 28 M V 32 M V 27 M V 35 M Q 4 C 3
```

Now assume that after this sequence we also execute the command `V 10`. We illustrate how a representation based on minimums processes these commands, Figure 3. The structure on left is the configuration after the initial sequence of

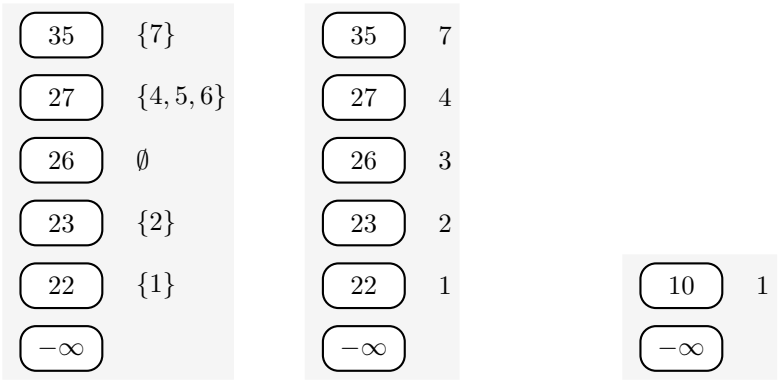

Figure 3: Illustration structure configuration using minimums to represent position sets.

commands. The structure in the middle represents the actual configuration that is stored in memory. Note that for each set we store only its minimum element. In particular note that the set associated with value 26 is represented by 3, even though position 3 was already marked as closed. As mentioned the hash-table keeps track of which positions are still open and closed positions are removed during transfer operations. This means that until then it is necessary to use all positions, closed or not, for our UF data structure. Hence the representative of a set is the minimum over all positions that are related to the set, closed or not. The structure on the right represents the structure after processing the `V 10` command.

Note that in this final configuration the set, of active positions, associated with value 10 should be $\{1, 2, 4, 5, 6, 7\}$. However it is represented only by the value 1. This set should be obtained by the following sequence of `Union` operations $\{1\} \cup \{2\} \cup \{4, 5, 6\} \cup \{7\}$. This amounts to removing the numbers 2, 4 and 7, which is obtained automatically when we alter the variable k.

Summing up, our data structure consists of the following elements:

- An array storing stack S. Each element in the stack contains a value v and position i, which is the minimum of the position set it represents.
- A hash-table to identify the active positions. In this configuration no mapping is required, it is enough to identify the active positions.

The general procedure for executing commands and the respective time bounds are the following:

- The `Value` command needs to truncate the stack, by updating variable k. This process requires $O(\log \ell)$ time because of the binary search procedure, but it can actually be improved to $O(1 + \log d)$ time where d is the number of positions removed from the position tree, by using an exponential search that starts at the top of the stack. Using an exponential search the expected amortized time of this operation is $O(1)$.
- The `Mark` command needs to add an element to the hash-table and an element to the stack S. This requires $O(\log \ell)$ time with high probability.

The `Make-Set` or `Union` operations require only $O(1)$ time hence the overall time is dominated by $O(\log \ell)$. The expected time of this operation is $O(1)$.

- The `Query` command needs to search for an element in the hash-table and compute a `Find` operation. The `Find` operation is computed with a binary search over minimums stored in the items of the stack. This operation requires $O(\log \ell)$ time with high probability. The expected amortized time is also $O(\log \ell)$, but it can be improved to $O(1 + \log(j - i + 1))$ for a query with indices (i, j), by using an exponential search from the top of the stack.
- The `Close` command needs to remove an element from the hash-table. This requires $O(\log \ell)$ time with high probability and $O(1)$ expected time.

□

The data structure of the previous theorem is simple because most of the complex de-amortizing procedure is handled in Lemma 7. We now focus on how to further reduce the high probability time bounds to $O(\log \log n)$. A simple way to obtain this is to have $\ell = O(\log n)$, i.e., having $O(\log n)$ active positions at each time. This may be achieved if `Query` positions are not necessarily exact, meaning that the data structure actually returns the solution for a query (i', j) instead of (i, j). The goal is that $j - i$ is similar in size of $j - i'$. Meaning that $j - i \leq j - i' < 2(j - i)$. In this scenario it is enough to keep $O(\log n)$ active positions, i.e., positions i' for which $j - i' = 2^c$ for some integer c. Since the data structure of Theorem 3 does not use the hash-table to reduce the position range, we can bypass its use in these queries. It is enough to directly determine the predecessor of i among the minimums stored in the stack S. Which is computed with a binary search or exponential search as explained in the proof.

The problem with this specific set of positions is that when j increases the active positions no longer provide exact powers of two. This is not critical because we can adopt an update procedure that provides similar results. Let $i_1 < i_2 < i_3$ represent three consecutive positions that are currently active. When j increases we check whether to keep i_2 or discard it. It is kept if $j - i_1 > 2(j - i_3)$, otherwise it is discarded. Hence we keep a list of active positions that gets updated by adding the new position j and checking two triples of active positions. We keep an index that indicates which triple to check and at each step use it to check two triples, moving from smaller to larger position values. The extremes of the list are not checked. We show the resulting list of positions in Table 2, where the bold numbers indicate the triples that will be checked in the next iteration. Whenever the triples to check reach the end of the list we have that the size of the list is at most $2 \log_2 n$, because the verification guarantees that the value $j - i$ is divided in half for every other position i. Therefore it takes at most $2 \log_2 n$ steps to traverse the list. Hence this list can contain at most $4 \log_2 n = O(\log n)$ positions and each time j is updated only $O(1)$ time is used.

Another alternative for obtaining $O(\log \log n)$ high probability time is to change the UF structure. In this case we use the same approach as Theorem 3 that relies on predecessor searches to compute the `Find` operation. This time we consider the Van Emde Boas tree that supports this operation efficiently, but requires longer to update.

1									
1	**2**								
1	**2**	**3**							
1	**2**	**3**	4						
1	**3**	**4**	**5**						
1	**3**	**4**	5	6					
1	**4**	**5**	**6**	7					
1	4	**6**	**7**	**8**					
1	**4**	**6**	7	8	9				
1	**4**	**7**	**8**	9	10				
1	4	**7**	**9**	**10**	11				
1	4	7	9	**10**	**11**	**12**			
1	**4**	**7**	9	10	11	12	13		
1	**7**	**9**	**10**	11	12	13	14		
1	**7**	**11**	**12**	13	14	15			
1	7	**11**	**13**	**14**	15	16			
1	7	11	**14**	**15**	**16**	17			
1	7	11	14	**16**	**17**	**18**			
1	**7**	**11**	14	16	17	18	19		
1	7	**11**	**14**	**16**	17	18	19	20	
1	7	11	**16**	**17**	**18**	19	20	21	
1	7	11	**16**	**19**	**20**	21	22		
1	7	11	16	**19**	**21**	**22**	23		
1	7	11	16	19	21	**22**	**23**	**24**	
1	**7**	**11**	16	19	21	22	23	24	25
1	**11**	**16**	**19**	21	22	23	24	25	26
1	11	**19**	**21**	**22**	23	24	25	26	27
1	11	**19**	**23**	**24**	25	26	27	28	
1	11	19	**24**	**25**	**26**	27	28	29	
1	11	19	**24**	**27**	**28**	29	30		

Table 2: Sequence of active position lists

Theorem 4. *It is possible to process, in real time, a sequence of RMQ commands in $O(\ell)$ space and in $O(\log\log\ell)$ time with high probability, for all operations except* `Value`*, which requires $O(\sqrt{\ell})$ time with high probability.*

Proof. First note that the `Value` command is not used in the de-amortized transfer procedure described in Lemma 7. Thus guaranteeing that the overhead per command will be only $O(\log\log\ell)$ time, once the statement of the Theorem is established.

One important consideration is to reduce the high probability time of the `Insert` and `Lookup` operations of the hash-table to $O(\log\log\ell)$ instead of $O(\log\ell)$. For this goal we modify the separate chaining to the 2-way chaining approach proposed by Azar, Broder, Karlin, and Upfal [1999], also with a maximum load factor of 50%.

We can now analyze the Van Emde Boas tree (VEB). This data structure is used as in Theorem 3 to store the minimum values of each set. Hence the underlying universe are the positions over A. Since this structure uses linear space in the universe size this would yield $O(n)$ space. However in this case we

can use the hash-table to reduce the position range and thus the required space becomes $O(\ell)$. Note that the reduced positions are also integers and we can thus correctly use this data structure.

Given that the time to compute a predecessor with this data structure is $O(\log\log\ell)$ this then implies this bound for the RMQ operations except `Value`. For this operation we have two caveats. First the binary search over the values in the stack S still requires $O(\log\ell)$ time. Second the `Union` operations in Algorithm 5 implies that it is necessary to remove elements from the VEB tree. This is not a problem for the `Mark` operation, Algorithm 6, because a single removal in this tree also requires $O(\log\log\ell)$ time. The issue for `Value` is that it may perform several such operations. In particular when d elements are removed from the stack it requires $O(d\log\log\ell)$ time. Recall the example in the proof of Theorem 3, where several union operations where executed to produce the set $\{1\}\cup\{2\}\cup\emptyset\cup\{4,5,6\}\cup\{7\}$. In that Theorem this was done automatically by modifying k, but in this case it is necessary to actually remove the elements 2, 3, 4 and 7 from the VEB tree. Note that the element 3 is the representative of the empty set. Even though it is not active it was still in the VEB tree.

This consists in removing from the VEB tree all the elements that are larger than 1. The VEB tree does not have a native operation for this process. Hence we have thus far assumed that this was obtained by iterating the delete operation. Still it is possible to implement this bulk delete operation directly within the structure, much like it can be done over binary search trees. In essence the procedure is to directly mark the necessary first level structures as empty and then do a double recursion, which is usually strictly avoided in this data structure. Given a variable u that identifies the logarithm of the universe size as $\ell=2^u$, this yields the following time recursion $T(u)=2^{u/2}+2T(u/2)$. Note that $2^{u/2}=\sqrt{\ell}$ is the number of structures that exist in the first level, and potentially need to be modified. This recursion is bounded by $O(2^{u/2})=O(\sqrt{\ell})$. □

As a final remark about this last result note that the time bound for the `Value` command is also $O(\log\log\ell)$ amortized, only the high probability bound is $O(\sqrt{\ell})$. This is because the iterated deletion bound $O(d\log\log\ell)$ that we mentioned in the proof does amortize to $O(\log\log\ell)$ and for each instance of the `Value` command we can choose between $O(d\log\log\ell)$ and $O(\sqrt{\ell})$.

This closes the theoretical analysis of the data structure. Further discussion is given in Section 6.

4 Experimental Results

Let us now focus on testing the performance of this structure experimentally. We implemented the data structure that is described in Theorem 2. We also designed a generator that produces random sequences of RMQ commands[1]. In these generated sequences the array A contained 2^{28} integers, i.e., $n=2^{28}$. Each integer was chosen uniformly between 0 and $2^{30}-1$, with the `arc4random_uniform` function[2].

[1] This prototype is available at `https://github.com/LuisRusso-INESC-ID/RMQminS`

[2] `https://github.com/freedesktop/libbsd`

We first implemented the version of our Algorithm described in Section 2, i.e., without using a hash table nor the transfer process. We refer to the prototype as the vanilla version and use the letter V to refer to it in our tables. We also implemented the version described in Theorem 2, which includes a hash table and requires a transfer process. We use the label T to refer to this prototype.

For a baseline comparison we used the `ST-RMQ-CON` algorithm by Alzamel, Charalampopoulos, Iliopoulos, and Pissis [2018]. We use the letter B to refer to this prototype. The implementation was obtained from their github repository `https://github.com/solonas13/rmqo`. We also compared against the fast RMQ heuristic by Kowalski and Grabowski [2018]. Their prototype is available at `https://github.com/kowallus/BbST`. Specifically we compared against the version that achieved the fastest performance for small queries BbST2 (512, 64). We use the letter F to refer to this prototype.

Our RMQ command sequence generator proceeds as follows. First it generates $n = 2^{28}$ integers uniformly between 0 and $2^{30} - 1$. Then it chooses a position to `Mark`, uniformly among the n positions available. This process is repeated q times. Note that the choices are made with repetition, therefore the same position can be chosen several times. Each marked position in turn will force a query command. All query intervals have the same length $l = j - i + 1$. Under these conditions it is easy to verify that the expected number of open positions at a given time is $l \times q/n$ and the actual number should be highly concentrated around this value. Hence we assume that this value corresponds to our ℓ parameter and therefore determine l as $\ell \times n/q$.

The tests were performed on a 64 bit machine, running `Linux 4.19.0-12`, which contained 32 cores in `Intel(R) Xeon(R) CPU E7- 4830 @ 2.13GHz` CPUs. The system has $256GB$ of RAM and of swap. Our prototypes were compiled with `gcc 8.3.0` and the baseline prototype with `g++`. All prototypes are compiled with `-O3`. We measure the average execution time and peak memory. These values were both obtained with the Unix system `time` command. The results are shown in Table 3 and 4.

The results show that our prototypes are very efficient. In terms of time both V and T obtain similar results, see Table 3. As expected T is slightly slower than V, but in practice this different is less than a factor of 2. The time performance of B is also very similar, in fact V and T are faster, which was not expected as B has $O(1)$ performance per operation and V and T have $O(\alpha(n))$. Even though in practice this difference was expected to be very small we did not expect to obtain faster performance. This is possibly a consequence of the memory hierarchy as B works by keeping A and all the queries in memory.

The prototype F is not always the fastest, in particular when the value of q is small. However for larger values of q it gradually starts to outperform B and also our prototypes. In particular for $q = 2^{26}$ it is significantly faster than all the other algorithms, moreover this difference increases with ℓ, as expected, because this parameter bounds the memory requirements of our data structure. Hence a larger value of ℓ means that our data structure is more susceptible to the memory hierarchy. We highlight this effect by plotting the space and time values for $q = 2^{24}$ in Figures 4 and 5, note as the time increase of prototype T is correlated with the increased memory requirements.

Concerning memory our prototypes also obtained very good performance, see Table 4. In particular we can clearly show a significant difference between using $O(m)$ and $O(\ell)$ space. Consider for example $q = 2^{26}$ and $\ell = 2^{16}$. For

		4	6	8	10	12	14	16	18	20	22	24	26	$\leftarrow \log(\ell)$
4	T	14												
	V	9												
	B	25												
	F	36												
6	T	15	14											
	V	10	10											
	B	25	25											
	F	37	37											
8	T	15	14	14										
	V	12	10	10										
	B	27	27	25										
	F	37	37	37										
10	T	15	14	15	15									
	V	10	10	10	10									
	B	25	25	27	25									
	F	37	37	36	37									
12	T	14	15	14	14	15								
	V	10	10	10	10	9								
	B	25	25	25	27	26								
	F	37	37	38	37	38								
14	T	14	14	15	14	15	14							
	V	10	10	10	10	10	10							
	B	26	26	25	25	27	25							
	F	37	37	37	37	37	37							
16	T	15	15	14	15	15	15	14						
	V	11	11	11	11	11	10	11						
	B	26	27	28	31	27	26	26						
	F	38	36	38	38	38	38	38						
18	T	15	15	15	15	16	16	16	16					
	V	11	10	10	11	12	11	11	11					
	B	29	30	29	28	29	29	28	27					
	F	38	37	38	37	40	37	40	37					
20	T	18	18	18	18	19	19	19	20	20				
	V	13	13	13	13	13	13	13	13	13				
	B	33	32	34	35	35	31	33	35	31				
	F	36	36	36	36	36	36	36	36	37				
22	T	26	26	26	26	26	27	29	30	33	33			
	V	17	17	18	18	18	17	19	20	20	21			
	B	45	46	47	48	49	50	51	54	52	47			
	F	39	38	38	37	38	39	37	37	38	39			
24	T	52	51	50	50	50	51	54	60	74	84	84		
	V	35	36	36	36	36	37	38	44	44	46	53		
	B	86	89	93	96	100	103	107	113	115	113	92		
	F	51	47	43	43	44	43	43	42	43	45	41		
26	T	111	111	107	106	107	110	123	135	178	221	285	293	
	V	93	93	95	98	98	99	100	123	124	119	133	281	
	B	175	184	194	202	211	220	229	248	258	263	261	222	
	F	91	74	66	58	58	58	57	55	58	57	56	53	
$\uparrow \log(q)$														

Table 3: Execution time per command in nanoseconds. The values are obtained by dividing total execution time by $n + q$.

		4	6	8	10	12	14	16	18	20	22	24	26	$\leftarrow \log(\ell)$
4	T	2												
	V	2												
	B	2G												
	F	1G												
6	T	2	2											
	V	2	2											
	B	2G	2G											
	F	1G	1G											
8	T	2	2	2										
	V	2	2	2										
	B	2G	2G	2G										
	F	1G	1G	1G										
10	T	2	2	2	2									
	V	2	2	2	2									
	B	2G	2G	2G	2G									
	F	1G	1G	1G	1G									
12	T	2	2	2	2	2								
	V	2	2	2	2	2								
	B	2G	2G	2G	2G	2G								
	F	1G	1G	1G	1G	1G								
14	T	2	2	2	2	2	3							
	V	2	2	2	2	2	2							
	B	2G	2G	2G	2G	2G	2G							
	F	1G	1G	1G	1G	1G	1G							
16	T	2	2	2	2	2	4	5						
	V	3	3	3	3	3	3	3						
	B	2G	2G	2G	2G	2G	2G	2G						
	F	1G	1G	1G	1G	1G	1G	1G						
18	T	2	2	2	2	2	4	10	14					
	V	6	6	6	6	6	6	6	6					
	B	2G	2G	2G	2G	2G	2G	2G	2G					
	F	1G	1G	1G	1G	1G	1G	1G	1G					
20	T	5	2	2	2	3	5	14	38	50				
	V	21	22	21	22	21	21	21	22	21				
	B	2G	2G	2G	2G	2G	2G	2G	2G	2G				
	F	1G	1G	1G	1G	1G	1G	1G	1G	1G				
22	T	14	5	2	2	2	5	15	49	130	194			
	V	81	81	81	82	81	81	81	81	81	81			
	B	3G	3G	3G	3G	3G	3G	3G	3G	3G	3G			
	F	1G	1G	1G	1G	1G	1G	1G	1G	1G	1G			
24	T	52	14	5	3	3	5	15	46	160	510	770		
	V	319	319	319	319	320	320	319	320	320	320	320		
	B	6G	6G	6G	6G	6G	6G	6G	6G	5G	5G	5G		
	F	1G	1G	1G	1G	1G	1G	1G	1G	1G	1G	1G		
26	T	51	51	14	5	3	5	16	52	182	499	2G	3G	
	V	1G	1G	1G	1G	1G	1G	1G	1G	1G	1G	1G	1G	
	B	14G	14G	14G	14G	14G	14G	14G	14G	14G	14G	14G	12G	
	F	2G	2G	2G	2G	2G	2G	2G	2G	2G	2G	2G	2G	
$\uparrow \log(q)$														

Table 4: Total memory peak in Megabytes, or in Gigabytes indicated by G.

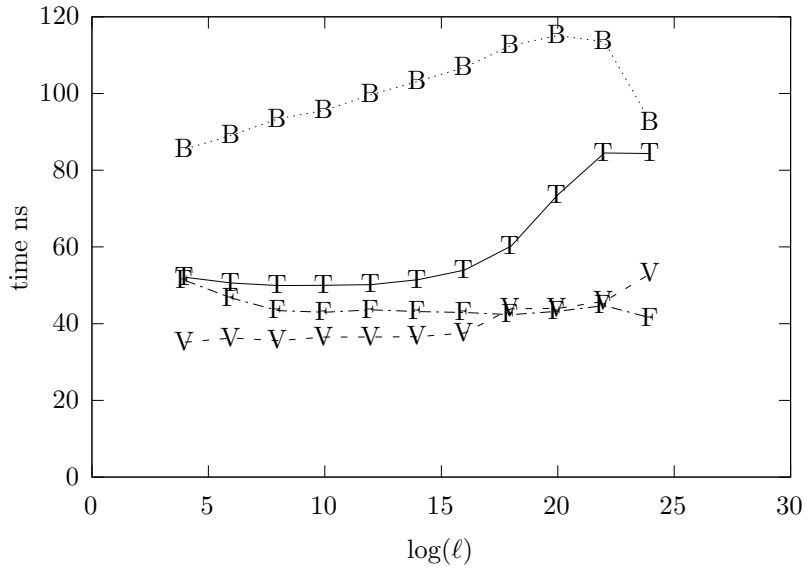

Figure 4: Time performance in nanoseconds for $q = 2^{24}$. The x axis indicates $\log(\ell)$.

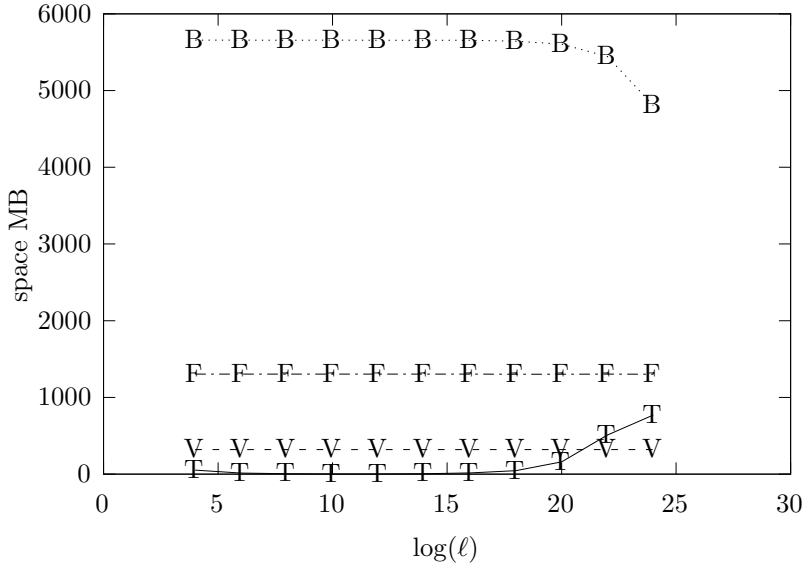

Figure 5: Space usage in Megabytes $q = 2^{24}$. The x axis indicates $\log(\ell)$.

these values V uses more than one Gigabyte of memory, whereas T requires only 15MB, a very large difference. In general T uses less memory than V, except when q and ℓ become similar. For example when $q = \ell = 2^{26}$ V uses around 1 GB of memory, whereas T requires 3 GB. This is expected, up to a given fixed factor. The baseline B requires much more memory as it stores more items in memory. Namely a compacted version of the array A and the solutions to all of the queries. Likewise the space requirements of the fast algorithm F is in general around the same other of magnitude as B, but smaller by a significant factor. Our prototypes V and T do not store query solutions. Instead whenever a query is computed its value is written to a `volatile` variable. This guarantees that all the necessary computation is performed, instead of optimized away by the compiler. However it also means that previous solutions are overwritten by newer results. We deemed this solution as adequate for an online algorithm, which in practice will most likely pass its results to a calling process. Moreover storing the query solutions would bound the experimental results to $\Omega(q)$ space, thus not being a fair test of $O(\ell)$ space.

5 Related Work

The Range Minimum Query problem has been exhaustively studied. This problem was shown to be linearly equivalent to the Lowest Common Ancestor problem in a static tree by Gabow, Bentley, and Tarjan [1984]. A recent perspective on this result was given by Bender and Farach-Colton [2000]. The first major solution to the LCA problem, by Berkman and Vishkin [1993], obtained $O(\alpha(n))$ time, using Union-Find data structures, in a similar way to our data structure. In fact this initial result was a fundamental inspiration for the data structure we propose in this paper. A constant time solution was proposed by Harel and Tarjan [1984]. A simplified algorithm was proposed by Schieber and Vishkin [1988]. A simplified exposition of these algorithms, and linear equivalence reductions, was given by Bender and Farach-Colton [2000]. A fundamental data structure in this algorithms is the Cartesian tree, which can be build in linear time using a stack to keep the rightmost or leftmost branch, see Crochemore and Russo [2020]. This process is similar to the one we use in this paper.

Even though these algorithms were simpler to understand and implement they still required $O(n)$ space to store auxiliary data structures, such as Cartesian trees. Moreover the constants associated with these data structures were large, limiting the practical application of these algorithms. To improve this limitation direct optimal direct algorithms for RMQ were proposed by Fischer and Heun [2006]. The authors also showed that their proposal improved previous results by a factor of two. However they also observed that for several common problem sizes, asymptotically slower variants obtained better performance. Hence a practical approach, that obtained a 5 time speedup, was proposed by Ilie, Navarro, and Tinta [2010]. Their approach was geared towards the Longest Common Extension on strings and leveraged the use its average value to.

A line of research directed by an approach that focused on reducing constants by using succinct and compressed representations was initiated by Sadakane [2007a] and successively improved by Sadakane [2007b], Sadakane and Navarro [2010] and Fischer and Heun [2011]. The last authors provide a systematic

comparison of the different results up to 2011. Their solution provided an $2n + o(n)$ bits data structure the answers queries in $O(1)$ time.

Still several engineering techniques can be used obtain more practical efficient solutions. An initial technique was proposed by Grossi and Ottaviano [2013]. A simplification implemented by Ferrada and Navarro [2017] used $2.1n$ bits and answered queries in 1 to 3 microseconds per query. Another proposal by Baumstark, Gog, Heuer, and Labeit [2017] obtained around a 1 microsecond per query (timings vary depending on query parameters) on an single core of the Intel Xeon E5-4640 CPU.

A new approach was proposed by Alzamel, Charalampopoulos, Iliopoulos, and Pissis [2018] where no index data structure was created by a preprocessing step. Instead all the RMQs are batched together and solved in $n+O(q)$ time and $O(q)$ space. This space was used to store a contracted version of the input array A and the solutions to the queries. This is essentially the approach we follow in this paper. Therefore in Table 3 we independently verify their query times in the nanoseconds. Also Table 4 reports the memory requirements of their structure. In a recent result Kowalski and Grabowski [2018] proposed a heuristic, without constant worst case time and a hybrid variation with $O(1)$ time and $3n$ bits. We compared their best results against our solution, Tables 3 and 4 and Figures 4 and 5. Their results are competitive against existing solutions, and obtain the best performance for large queries for which a large simultaneous number of them overlap.

For completion we also include references to the data structures we used, or mentioned, in our approach. The technique by Briggs and Torczon [1993], also described by Bentley [2016] and Aho and Hopcroft [1974], provides a way to use memory without the need to initialize it. Moreover each time a given memory position needs to be used for the first time it requires only $O(1)$ time to register this change. The trade-off with this initial data structure is that it triples the space requirements. Recently this space overhead has been, dramatically, reduced from the factor of three down to a single bit, see Katoh and Goto [2021].

For now we do not have an implementation of Lemma 7. An experimental study by Fredriksson and Kilpeläinen [2015] concluded that initializable array techniques, similar to the ones mentioned above, are efficient when the access frequency is less than 10% of the size of the array. Since our transfer process will eventually utilize the whole array, a more pragmatical approach is likely to be practical. We note that we can start initializing an array some time before we need to insert data into it. For the destination structure it is not a problem because we can assume that the whole merge process includes the time for the initial clean-up, all within $(a_1 + a_2)/2$ as explained in Lemma 7. The active structure requires some more forethought. In essence when the merge processes starts and we start using an active structure that supports $(a_1+a_2)/2$ elements it is a good time to start cleaning a piece of memory that supports $(a_1 + a_2 + c_1 + c_2)/2$ elements, as this will be the number of elements of the future active structure. We will start using this structure when the current merge finishes. Since this number of elements is at most $a_1 + a_2$ it is possible to finish the clean-up when at most $(a_1 + a_2)/2$ operations have executed, by cleaning two element positions in each operation.

The Union-Find data structure is a fundamental piece of our solution. The original proposal to represent disjoint sets that can support the `Union` and `Find` operations was by Galler and Fisher [1964]. Their complexity was bounded

by $O(\log^*(n))$ amortized time per operation by Hopcroft and Ullman [1973]. The analysis of the time bound was later refined to $O(\alpha(n))$ by Tarjan and van Leeuwen [1984]. Lower bound analysis guarantees that these bounds are optimal Tarjan [1979] and Fredman and Saks [1989]. However in the case where the sequence of operations is known *a priori* it is possible to obtain $O(1)$ amortized time per operation, as shown by Gabow and Tarjan [1985]. An exhaustive survey was given by Galil and Italiano [1991]. An elementary description of this data structure was provided by Cormen, Leiserson, Rivest, and Stein [2009] and Sedgewick and Wayne [2011].

Hash tables date back to the origin of computers. A history on the subject and the first theoretical analysis was given by Knuth [1963]. This analysis established the constant expect time bound. The high probability bound of separate chaining can be derived from a balls and bins model, see Mitzenmacher and Upfal [2017]. Actually a better bound was obtained by Gonnet [1981]. The 2-way chaining hash-table was proposed by Azar, Broder, Karlin, and Upfal [1999], which also established its constant expected time and high probability bound.

Exponential searches where proposed by Bentley and Yao [1976] and Baeza-Yates and Salinger [2010] and can be used to speed-up the binary search algorithm when the desired element is close to the beginning or end of a list. For a detailed presentation of binary search see Knuth [1998], section 6.2.1.

The data structure by van Emde Boas, Kaas, and Zijlstra [1976] provides support for `Predecessor` queries over integers in $O(\log \log n)$ time. The structure is obtained by "essentially" recursively dividing a binary search tree along half its height. All the elements in the top half are stored directly in a structure, that acts like a node with $\sqrt{u}$ children, where u is the size of a universe of integers without missing integers in between any two of its values. The elements in the lower half of the tree are recursively structured. An elementary description is given by Cormen, Leiserson, Rivest, and Stein [2009]. The y-fast trie data structure was proposed by Willard [1983] to reduce the large space requirements of the Van Emde Boas tree. This data structure obtains the $O(\log \log n)$ time bound, only that amortized. For this reason we did not considered it in Theorem 4. Also in the process the authors also describe x-fast tries.

6 Discussion and Conclusion

We can now discuss our results in context. In this paper we started by defining a set of commands that can be used to form sequences. Although these commands are fairly limited they can still be used for several important applications. First notice that if we are given a list of (i, j) RMQs we can reduce them to the classical context. This can be achieved with two hash tables. In the first table store the queries indexed by i and on the second by j. We use the first table to issue `Mark` commands and the second to issue `Query` commands. This requires some overhead but it allows our approach to be used to solve classical RMQ problems. In particular it will significantly increase the memory requirements, as occurs in Table 4 between T and B.

Our data structures can be used in online and real-time applications. Note that in particular we can use our commands to maintain the marked positions in a sliding window fashion. Meaning that at any instant we can issue `Query`

commands for any of the previous ℓ positions. The extremely small memory requirements of our approach makes our data structure suitable to be used in routers, switches or in embedded computation devices with low memory and CPU resources. For example in routing packages a router may choose to send a package to a node not only if its current distance is small, but also if the node has proven to be reliable in the past, i.e., its maximum distance was not too big. In this case we would use range maximum queries, which we can obtain by swapping the signs of the A[i] values. If this decision is static, i.e., if the time interval that a node needs to be stable is pre-determined then it would be enough to keep a single maximum value. However we could use our data structure if this interval depends on the protocol, the application, the end-user, etc.

The simplest configuration of our data structure consists of a stack combined with a Union-Find data structure. For this structure we can formally prove that our procedures correctly compute the desired result, Theorem 1. We then focused on obtaining the data structure configuration that yielded the best performance. Obtained $O(\alpha(n))$ amortized time and $O(\ell)$ space, see Theorem 2. This result is in theory slower than the result by Alzamel, Charalampopoulos, Iliopoulos, and Pissis [2018], which obtained $O(1)$ amortized query time. We compared experimentally these approaches in Section 4. The results showed that out approach was competitive, both in terms of time and space, our prototype V was actually faster than the prototype B by Alzamel et al. [2018] and obtained similar performance to prototype F by Kowalski and Grabowski [2018]. The prototype T was slower than V by around, at most, a factor of 2. However, in general, it required orders of magnitude less space. We also showed that it was possible for our data structure to obtained $O(1)$ amortized query time (Corolary 2), mostly for theoretical competitiveness. We did not implement this solution.

We described how to reduce the space requirements down to $O(\ell)$, by transferring information among structures and discarding structures that became full, see Lemma 7. In theory this obtained the same $O(\alpha(n))$ amortized time but significantly reduced space requirements. We also implemented this version of the data structure. In practice the time penalty was less than a 2 factor. Moreover, for some configurations, the memory reduction was considerable, see Table 4.

Lastly we focused on obtaining real time performance. We obtained a high probability bound of $O(\log n)$ amortized time per query, see Theorem 3. This bound guarantees real time performance. We then investigated alternatives to reduce this time bound to $O(\log \log n)$. We proposed two solutions. In one case we considered approximate queries, thus reducing the necessary amount of active positions to $O(\log n)$. In the other case we used the Van Emde Boas tree, which provided a $O(\log \log n)$ high probability time bound for all commands except `Value`, see Theorem 4. In this later configuration the `Value` command actually obtained an $O(\sqrt{\ell})$ bound, which is large, but the corresponding amortized value is only $O(\log \log n)$.

7 Acknowledgments

The work reported in this article was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UIDB/50021/2020 and project NGPHYLO PTDC/CCI-BIO/29676/2017.

We are grateful to Kowalski and Grabowski for providing support with the preparation of their prototype for testing.

References

Mai Alzamel, Panagiotis Charalampopoulos, Costas S. Iliopoulos, and Solon P. Pissis. How to answer a small batch of RMQs or LCA queries in practice. In *Lecture Notes in Computer Science*, pages 343–355. Springer International Publishing, 2018.

Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-wesley professional, 2011.

Harold N. Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–221, apr 1985.

Preston Briggs and Linda Torczon. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems*, 2(1-4):59–69, mar 1993.

Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press, 2017.

Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–281, mar 1984.

Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. *SIAM Journal on Computing*, 29(1):180–200, jan 1999.

Tomasz M. Kowalski and Szymon Grabowski. Faster range minimum queries. *Software: Practice and Experience*, 48(11):2043–2060, 2018.

Harold N. Gabow, Jon Louis Bentley, and Robert E. Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing - STOC 84*. ACM Press, 1984.

Michael A. Bender and Martín Farach-Colton. The LCA problem revisited. In *Lecture Notes in Computer Science*, pages 88–94. Springer Berlin Heidelberg, 2000.

Omer Berkman and Uzi Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2):221–242, apr 1993.

Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, may 1984.

Baruch Schieber and Uzi Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM Journal on Computing*, 17(6):1253–1262, dec 1988.

Maxime Crochemore and Luís M.S. Russo. Cartesian and Lyndon trees. *Theoretical Computer Science*, 806:1–9, feb 2020.

Johannes Fischer and Volker Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Combinatorial Pattern Matching*, pages 36–48. Springer Berlin Heidelberg, 2006.

Lucian Ilie, Gonzalo Navarro, and Liviu Tinta. The longest common extension problem revisited and applications to approximate string searching. *Journal of Discrete Algorithms*, 8(4):418–428, dec 2010.

Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, feb 2007a.

Kunihiko Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms*, 5(1):12–22, mar 2007b.

Kunihiko Sadakane and Gonzalo Navarro. Fully-functional succinct trees. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, jan 2010.

Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2): 465–492, jan 2011.

Roberto Grossi and Giuseppe Ottaviano. Design of practical succinct data structures for large data collections. In *Experimental Algorithms*, pages 5–17. Springer Berlin Heidelberg, 2013.

Héctor Ferrada and Gonzalo Navarro. Improved range minimum queries. *Journal of Discrete Algorithms*, 43:72–80, mar 2017.

Niklas Baumstark, Simon Gog, Tobias Heuer, and Julian Labeit. Practical Range Minimum Queries Revisited. In *16th International Symposium on Experimental Algorithms (SEA 2017)*, volume 75 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-036-1.

Jon Bentley. *Programming pearls*. Addison-Wesley Professional, 2016.

Alfred V Aho and John E Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974.

Takashi Katoh and Keisuke Goto. In-place initializable arrays, 2021.

Kimmo Fredriksson and Pekka Kilpeläinen. Practically efficient array initialization. *Software: Practice and Experience*, 46(4):435–467, feb 2015. doi: 10.1002/spe.2314. URL `https://doi.org/10.1002%2Fspe.2314`.

Bernard A. Galler and Michael J. Fisher. An improved equivalence algorithm. *Communications of the ACM*, 7(5):301–303, may 1964.

J. E. Hopcroft and J. D. Ullman. Set merging algorithms. *SIAM Journal on Computing*, 2(4):294–303, dec 1973.

Robert Endre Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences*, 18(2):110–127, apr 1979.

M. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing - STOC 89*. ACM Press, 1989.

Zvi Galil and Giuseppe F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys*, 23(3):319–344, sep 1991.

Don Knuth. Notes on "open" addressing. 1963.

Gaston H. Gonnet. Expected length of the longest probe sequence in hash code searching. *Journal of the ACM*, 28(2):289–304, apr 1981.

Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82 – 87, 1976. ISSN 0020-0190.

Ricardo Baeza-Yates and Alejandro Salinger. Fast intersection algorithms for sorted sequences. In *Algorithms and Applications*, pages 45–61. Springer Berlin Heidelberg, 2010.

D.E. Knuth. *The Art of Computer Programming: Volume 3: Sorting and Searching*. Pearson Education, 1998. ISBN 9780321635785.

P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10(1):99–127, dec 1976.

Dan E. Willard. Log-logarithmic worst-case range queries are possible in space Θ(n). *Information Processing Letters*, 17(2):81–84, aug 1983.