Algorithmica
https://doi.org/10.1007/s00453-022-01046-3

Resilient Level Ancestor, Bottleneck, and Lowest Common Ancestor Queries in Dynamic Trees

Luciano Gualà[1] · **Stefano Leucci**[2] · **Isabella Ziccardi**[2]

Received: 19 December 2021 / Accepted: 23 September 2022

Abstract
We study the problem of designing a *resilient* data structure maintaining a tree under the Faulty-RAM model [Finocchi and Italiano, STOC'04] in which up to δ memory words can be corrupted by an adversary. Our data structure stores a rooted dynamic tree that can be updated via the addition of new leaves, requires linear size, and supports *resilient* (weighted) level ancestor queries, lowest common ancestor queries, and bottleneck vertex queries in $O(\delta)$ worst-case time per operation.

Keywords Level ancestor queries · Lowest common ancestor queries · Bottleneck vertex queries · Resilient data structures · Faulty-RAM model · Dynamic trees

1 Introduction

Due to a diverse spectrum of reasons, ranging from manufacturing defects to charge collection, the data stored in modern memories can sometimes face corruptions, a problem that is exacerbated by the recent growth in the amount of stored data. To make matters worse, even a single memory corruption can cause classical algorithms and data structures to fail catastrophically. One mitigation approach relies on low-level error-

Luciano Gualà, Stefano Leucci and Isabella Ziccardi have contributed equally to this work.

✉ Stefano Leucci
stefano.leucci@univaq.it

Luciano Gualà
guala@mat.uniroma2.it

Isabella Ziccardi
isabella.ziccardi@graduate.univaq.it

[1] Department of Enterprise Engineering, University of Rome "Tor Vergata", Via del Politecnico 1, 00133 Rome, RM, Italy

[2] Department of Information Engineering, Computer Science, and Mathematics, University of L'Aquila, Via Vetoio, 67100 Coppito, AQ, Italy

Published online: 08 October 2022

correcting schemes that transparently detect and correct such errors. These schemes however either require expensive hardware or employ space-consuming replication strategies. Another approach, which has recently received considerable attention [1–7], aims to design *resilient* algorithms and data structures that are able to remain operational even in the presence of memory faults, at least with respect to the set of uncorrupted values.

In this paper we tackle the problem of designing resilient data structures that store a *dynamic* rooted tree T while answering several types of queries. More formally, we focus on maintaining a tree that initially consists of a single vertex (the root of the tree) and can be dynamically augmented via the `AddLeaf`(v) operation that appends a new leaf as a child of an existing vertex v.[1] It is possible to *query* T in order to obtain information about its current topology. We consider the following well-known query types:

(Weighted) Level Ancestor Queries: Given a vertex v and an integer k, the query `LA`(v, k) returns the *k-parent of* v, i.e., the vertex at distance k from v among the ancestors of v. In the weighted version of the problem each vertex of the tree T is associated with a small (polylogarithmic) positive integer weight, and a query needs to report the closest ancestor u of v such that the total weight of the path from v to u in T is at least k.[2]

Lowest Common Ancestor Queries: Given two vertices u, v, the query `LCA`(u, v) returns the vertex at maximum depth in T that is simultaneously an ancestor of both u and v.

Bottleneck Queries: In this problem, each vertex has an associated integer weight and, given two vertices u, v, a `BVQ`(u, v) query reports the minimum/maximum-weight vertex in the path between u and v in T.[3] When T is a path, the above problem can be seen as a dynamic version of the classical *range minimum query* problem which asks to answer `RMQ`(i, j) queries reporting the minimum element between the i-th and the j-th element of a (static) input sequence [11].

For all of the above problems, *linear-size* data structures are known for the *non-resilient* case, i.e., on the unit-cost RAM model with a word size of $\Theta(\log n)$, where n is the maximum number of vertices in the tree [9, 12]. These data structures support both the `AddLeaf` and the query operations in constant worst-case time. It is then natural to investigate what can be achieved for the above problem when the sought data structures are required to withstand memory faults.

To precisely capture the behaviour of resilient algorithms, one needs to employ a model of computation that takes into account potential memory corruptions. To this aim, we adopt the *faulty-RAM* model introduced by Finocchi and Italiano in [1]. This model is similar to the classical RAM model except that all but $O(1)$ memory words can be subject to corruptions that alter theirs contents to an arbitrary value, and

[1] In the literature this setting is also called *incremental* or *semi-dynamic* to emphasize that arbitrary insertions and deletions of tree vertices/edges are not supported. In this paper, unless otherwise specified, we follow the terminology of [8] by considering *dynamic* trees that only support insertion of leaves.

[2] We inherit the restriction on vertex weights from [9], whose data structure we use in our constructions.

[3] It is easy to see that this also captures the well-known *bottleneck edge query* variant [10], in which weights are placed on edges instead of vertices.

that cannot be detected by the algorithm. The overall number of corruptions is upper bounded by a parameter δ and such corruptions are chosen in a *worst-case* fashion by a computationally unbounded *adversary*. We consider a word size of $\Theta(\log n)$. A more detailed description of the faulty-RAM model can be found in Sect. 2.

A simple error-correcting strategy based on replication provides a general scheme for obtaining resilient versions of any classical *non-resilient* data structure at a cost of a $\Theta(\delta)$ blowup in both the time needed for each operation and the size of the data structure. This space overhead is undesirable, especially when δ can be large. For the above reason, the main goal in the area is obtaining compact solutions with a particular focus on linear-size data structures [1–6, 13, 14]. However, for linear-size data structures, even $\delta = \omega(1)$ corruptions can be already sufficient for the adversary to irreversibly corrupt some of the stored elements [13]. The solution adopted in the literature is that of suitably relaxing the notion of correctness by only requiring queries to answer correctly with respect to the portions of the data structure that are uncorrupted. Notice that this is not easy to obtain since corruptions in unrelated parts of the data structure can still misguide the execution of a query (see [13] for a discussion).[4]

1.1 Our Results

We design a data structure maintaining a dynamic tree that can be updated via the addition of new leaves, and supports *resilient* (weighted) `LA`, `LCA`, and `BVQ` queries.

Our data structure stores each vertex of the current tree T in a single memory word of $\Theta(\log n)$ bits. We will say that a vertex v is corrupted if the memory word associated with v has been modified by the adversary. A resilient query is required to correctly report the answer when no vertex in the tree path between the two vertices explicitly or implicitly defined by the query is corrupted. For example, a `LA`(v, k) query correctly reports the k-parent u of v whenever every vertex in the unique path from u to v in T is uncorrupted.

We deem our notion of resilient query to be quite natural since, in any reasonable representation of T, the adversary can locally corrupt the parent-children relationship and hence change the observed topology of T. See Fig. 1 for an example.

Our data structure occupies linear (w.r.t. the current number of nodes) space, and supports the `AddLeaf` operation and the `LA`, `BVQ`, and `LCA` queries in $O(\delta)$ worst-case time. For weighted `LA` queries, the above bound on the query time holds as long as $\delta = O(\text{polylog} n)$. No constraint on δ is required for `BVQ`, `LCA`, and unweighted `LA` queries.

We point out that our solution is obtained through a general vertex-coloring scheme which is, in turn, used to "shrink" T down to a compact tree Q of size $O(n/\delta)$ that can be made resilient via replication. Each edge of Q represents a path of length δ between two consecutive colored nodes in T. If no corruption occurs, this coloring

[4] For example, the authors of [13] consider the problem of designing linear-size resilient dictionaries and adopt a notion of *resilient search* that requires the search procedure to answer correctly w.r.t. all uncorrupted keys (see Sect. 1.2 for a more precise definition). Notice how the classical solutions based on search trees do not meet this requirement since a single unrelated corruption can destroy the search path leading to the sought key.

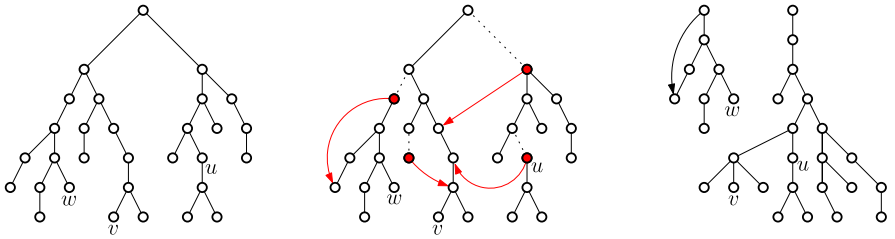

Fig. 1 Illustration of resilient LA queries. The current tree T logically maintained by the data structure is depicted in (**a**). In this example, each vertex maintains a reference to its parent in T. In (**b**) some of the parent-child relationships have been altered by the adversary by corrupting the nodes highlighted in red. Since the algorithm cannot distinguish corrupted memory words from uncorrupted ones, its (defective) view of T is shown in (**c**). Nevertheless, a resilient data structure must still be able to correctly answer queries involving uncorrupted paths. For example, the query $\texttt{LA}(v, k)$ is required to answer correctly for all (meaningful) values of k since the path from v to the root is uncorrupted, while query $\texttt{LA}(w, k)$ is required to answer correctly for $k \leq 2$. Since u is corrupted, the query $\texttt{LA}(u, k)$ is allowed to answer incorrectly regardless of the value of k (Color figure online)

scheme is regular and will color all vertices having a depth that is a multiple of δ. While it is possible for corruptions to *locally* destroy the above pattern, our coloring is able to automatically recover as soon as we move away from the corrupted portions of the tree. We feel that such a scheme can be of independent interest as a useful tool to design other resilient data structures involving dynamic trees.

We leave the problem of understanding whether, similarly to other resilient data structures [4, 13], one can prove a lower bound of $\Omega(\delta)$ on the time needed to perform `AddLeaf` operation and/or to answer our queries.

1.2 Related Work

1.2.1 Non-resilient Data Structures

Before discussing the known results in our faulty memory model, we fist give an overview of the closest related results in the fault-free case. Since the landscape of data structures that answer queries on dynamic trees is vast and diverse, we will focus only on the best-known data-structures capable of answering `LA`, `BVQ`, or `LCA` queries.

As far as `LA` queries are concerned, the problem has been first formalized in [15] and in [8]. Both papers consider the case in which the tree T is *static* and show how to build, in linear-time, a data structure that requires linear space and that answers queries in constant worst-case time (albeit the hidden constant in [15] is quite large). A simple and elegant construction achieving the same (optimal) asymptotic bounds is given in [16]. In [8], the *dynamic* version of the problem was also considered: the authors provide a data structure supporting both `LA` queries and the `AddLeaf` operation in constant *amortized* time. The best known *dynamic* data structure is the one of [9], which implements the above operations in constant *worst-case* time. This data structure also supports constant-time `BVQ` queries and constant-time weighted `LA`

queries when the vertex weights are polylogarithmically bounded integers.[5] Moreover, the solution of [9] also provides amortized bounds for the problem of maintaining a forest of n nodes under *link* operations and `LA` queries. Here a *link* operation is a more general update than `AddLeaf` since it allows for the addition of a new edge that connect any two vertices in different trees of the forest. In this case, a sequence of m operations requires $O(m\alpha(m, n))$ time, where α is the inverse Ackermann function.

Regarding `BVQ` queries with integer weights, in addition to the solution discussed above (which supports leaf additions and queries in constant-time), [18] shows how to also support leaf deletions using $O(1)$ amortized time per update and constant worst-case query time.

The problem of answering `LCA` queries is a fundamental problem which has been introduced in [19]. In [20], Harel and Tarjan show how to preprocess in linear time any *static* tree in order to build a linear-space data structure that is able to answer `LCA` queries in $O(1)$ time. The case of *dynamic* trees is also well-understood: it is possible to simultaneously support (i) insertions of leaves and internal nodes, (ii) deletion of leaves and internal nodes with a single child, and (iii) `LCA` queries, in constant worst-case time per operation [12].

1.2.2 Resilient Data Structures

As already mentioned, the Faulty-RAM model has been introduced in [1] and used in the context of resilient data structures in [2] where the authors focused on designing resilient dictionaries, i.e., dynamic sets that support insertions, deletions, and lookup of elements. Here the lookup operation is only required to answer correctly if either (i) the searched key k is in the dictionary and is uncorrupted, or (ii) k is not in the dictionary and no corrupted key equals k. The best-known (linear-size) resilient dictionary is provided in [3] and supports each operation in the optimal $O(\log n + \delta)$ worst-case time, where n is the number of stored elements. The Faulty-RAM model has also been adopted in [4], where the authors design a (linear-size) resilient priority queue, i.e., a priority queue supporting two operations: *insert* (which adds a new element in the queue) and *deletemin*. Here *deletemin* deletes and returns either the minimum uncorrupted value or one of the corrupted values. Each operation requires $O(\log n+\delta)$ amortized time, while $\Omega(\log n + \delta)$ time is needed to answer an *insert* followed by a *deletemin*.

The Faulty-RAM model has also been adopted in the context of designing resilient algorithms. We refer the reader to [5] for a survey on this topic.

A resilient dictionary for a variant of the Faulty-RAM model in which the set of corruptible memory words is random (but still unknown to the algorithm) has been designed in [7].

In a broader sense, problems that involve non-reliable computation have received considerable attention in the literature, especially in the context of sorting and searching. See for example [21–27].

[5] If we do no require the vertex weights to be polylogarithmically bounded, then the lower bound for the predecessor problem apply. See [17] and the discussion in [9].

1.3 Structure of the Paper

The paper is organized as follows. Section 2 introduces the used notation and formally defines the Faulty-RAM model. It also briefly describes the error-correcting replication strategy mentioned in the introduction. For technical convenience, in Sects. 3 and 4 we describe our data structure for LA queries only. This allows us to introduce all the ideas behind the more general coloring scheme discussed above. As a warm up, we first consider the simpler case in which the tree T is *static* and is already known at construction time (Sect. 3), and we then tackle the dynamic version of the problem (Sect. 4) for which we give our main result. The description of how to modify our data structure to handle the other types of queries can be found in Sect. 5.

2 Preliminaries

2.1 Notation

Let T be a rooted tree. For each node[6] $v \in T$, we denote with $\text{parent}(v)$ the parent of v. If π is a path, we denote by $\|\pi\|$ its length, i.e., the number of its edges. Given any two nodes u, v, we denote by $d_T(u, v)$ the length of the (unique) path between u and v in T. Moreover, if π traverses u and v, we denote by $\pi[u : v]$ the subpath of π between u and v, endpoints included. We will use round brackets instead of square brackets to denote that the corresponding endpoint is excluded (so that, e.g., $\pi(u : v]$ denotes the subpath of π between u and v where u is excluded and v is included).

2.2 Faulty Memory Model

We now formally describe the *Faulty-RAM* model introduced by Finocchi and Italiano in [1]. In this model the memory is divided in two regions: a *safe region* with $O(1)$ memory words, whose locations are known to the algorithm designer, and the (unreliable) *main memory*. An adaptive adversary can perform up to δ corruptions, where a corruption consists in instantly modifying the content of a word from the main memory. The adversary knows the algorithm and the current contents of the memory, has an unbounded computational power, and can simultaneously perform one or more corruptions at any point in time. The safe region cannot be corrupted by the adversary and there is no error-detection mechanism that allows the algorithm to distinguish the corrupted memory locations from the uncorrupted ones.

Without assuming the existence of $O(1)$ words of safe memory, no reliable computation is possible: in particular, the safe memory can store the code of the algorithm itself, which otherwise could be corrupted by the adversary.

As observed in [13] (and already mentioned in the introduction), there is a simple strategy that allows any non-fault tolerant data structure on the RAM model to also work on the Faulty-RAM model, albeit with a multiplicative $\Theta(\delta)$ blow-up in its time and space complexities. Essentially, such a solution implements a trivial error-

[6] We use the terms "node" and "vertex" interchangeably.

correcting mechanism by simulating each memory word w in the RAM model with a set W of $2\delta + 1$ memory words in the Faulty-RAM model: writing a value x to w means writing x to all words in W, and reading w means computing the majority value of the words in W (which can be done in $O(\delta)$ time, and $O(1)$ space using the safe memory region and the Boyer-Moore majority vote algorithm [28]). We refer to such technique as the *replication strategy*.

3 Warming Up: LA Queries in Static Trees

In order to introduce our ideas, in this section we show how to build a simplified version of our resilient data structure when the tree T cannot be dynamically modified. Our simplified data structure requires linear space and answers level-ancestor queries in $O(\delta)$ time. As opposed to our *dynamic* data structure, in this special case the tree T must be known in advance and hence we need to initialize our data structure from an input tree T. For simplicity, we assume that no corruptions occur while our data structure is being built.

3.1 Description of the Data Structure

Let T be a rooted tree with n nodes. To define the data structure for T, we need to divide the nodes of T into two sets: the *black* nodes and the *white* nodes. We define the set of black nodes to ensure that its cardinality is $O(n/\delta)$: a node v in T is *black* if we simultaneously have that (i) its depth in T is a multiple of δ, and (ii) the subtree of T rooted in v has height at least $\delta - 1$. A node v in T is *white* if it is not black. We notice that for each black node v in T there are at least δ distinct nodes (i.e., all the vertices in the path from v to any vertex having depth $\delta - 1$ in the subtree of T rooted at v), thus implying that the total number of black nodes in T is at most n/δ.

If we define a relation of parenthood for the black nodes of T, we can define a new *black tree* Q in which each vertex $\overline{v}$ is associated with a black vertex v of T. The parent of $\overline{v}$ in Q is the vertex $\overline{u}$ corresponding to the lowest black proper ancestor u of v in T. See Fig. 2 for an example.

Our data structure stores the (colored) tree T, as described in the following, along with an additional data structure D_Q that is able to answer LA queries on Q. The tree T is stored as an array of records, where each record is associated with a vertex of T, occupies $\Theta(\log n)$ bits, and is stored in a single memory word. The memory word associated with a node v stores:

- a pointer p_v to parent(v), if any. If v is the root of T then $p_v = \texttt{null}$;
- a pointer q_v to the corresponding node $\overline{v}$ in Q, if any. If no such node exists, i.e., if v is white, then $q_v = \texttt{null}$.

Moreover we maintain, for each vertex $\overline{v}$ of Q, a pointer to the corresponding vertex v of T as satellite data. The data structure D_Q is the resilient version of any (non-resilient) data structure that is capable of answering LA queries on static trees in constant time and requires linear space (see, e.g., the data structure in [16]).

Algorithm 1: Answers a level ancestor query $\texttt{LA}(v, k)$ in the special case of static trees.

```
1 if k ≤ 2δ then return climb(v, k);
2 Climb up the tree T from v for 2δ nodes searching a black node;
3 if the previous procedure did not find a black node then return error;
4 v' ← the black node found in the previous procedure;
5 d ← distance between v' and v; k' ← k − d;
6 u' ← LA_Q(q_{v'}, ⌊k'/δ⌋);
7 k_rest ← k' − ⌊k'/δ⌋ · δ;
8 return climb(u', k_rest);
```

As we observed before, any data structure can be made resilient with a multiplicative $\Theta(\delta)$ blow-up in its time and space complexities using the replication strategy described in Sect. 2. In our case, since the number of vertices in Q is $O(n/\delta)$, the final space required to store D_Q is $O(n)$ and the query time becomes $O(\delta)$. Notice that, in spite of the (at most δ) memory corruptions performed by the adversary, the data structure D_Q always returns the correct answer to all possible LA queries on Q. We will denote by $\texttt{LA}_Q(\overline{v}, k)$ the level-ancestor query on Q, which returns the vertex of T corresponding to the k-parent of $\overline{v}$ in Q (if no corruptions occur in T then $\texttt{LA}_Q(\overline{v}, k)$ is exactly the δk-parent of v in T).

3.2 The Resilient Level-Ancestor Query

In this section we show how to implement our resilient LA query. We start by defining a routine that will be useful in the sequel: if v is a node of T and i is a non-negative integer, we denote by $\texttt{climb}(v, i)$ a procedure that returns the vertex reached by a walk on T that starts from v and iteratively moves to the parent of the current vertex i times. When the procedure encounters a vertex u with pointer $p_u = \texttt{null}$ that has to be followed, $\texttt{climb}(v, i)$ reports that the root has been reached. Notice that $\texttt{climb}(v, i)$ requires $O(i)$ time and, whenever no corrupted vertices are encountered during the walk, it correctly returns the i-parent of v. Although the $\texttt{climb}(\cdot, \cdot)$ procedure could immediately be used to answer an LA query, doing so requires $\Omega(n)$ time in the worst case. To improve the query time we use the data structure D_Q described above and we distinguish between *short* and *long* $\texttt{LA}(v, k)$ queries depending on the value of k.

Short queries, i.e., queries $\texttt{LA}(v, k)$ with $k \leq 2\delta$, are handled by simply invoking $\texttt{climb}(v, k)$ and, from the above observation, it follows that this is a resilient query. For longer queries the idea is that of locating a nearby black ancestor of v, performing an $\texttt{LA}_Q$ query on Q to quickly reach a black descendant u' of the k-parent w of v such that $d(u', w) \leq \delta$, and finally using the climb procedure once more to reach w from u'. See Algorithm 1.

During the execution of our resilient query algorithm we always ensure that all followed pointers are valid. Since we are dealing with a static tree T, we can handle invalid pointers (e.g., pointers p_v that do not refer to some node of T) simply by halting the whole query procedure and reporting an error. A slightly more sophisticated

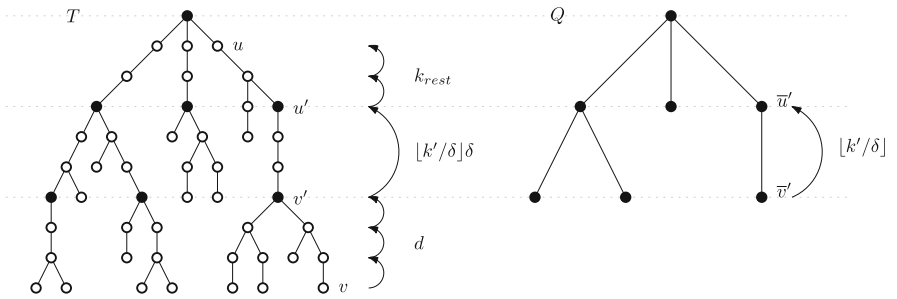

Fig. 2 Left: A static tree T that has been colored according to the scheme in Sect. 3 for $\delta = 3$. Right: the corresponding black tree Q. We also show the path climbed while answering the query $\texttt{LA}(v, k)$ with $k = 8$. In this case $d = 3$, $\lfloor k'/\delta \rfloor = 1$ and $k_{\text{rest}} = 2$. Notice how Q is used to quickly reach u' from v'

handling of invalid pointers will be used to tackle the dynamic case. An example LA query is given in Fig. 2.

The correctness of the above algorithm immediately follows from the fact that, when no vertex between v and the k-parent of v is corrupted, v must have a black ancestor at distance at most 2δ and from the fact that the replication strategy ensures that all queries on Q are always answered correctly.[7]

To show that Algorithm 1 answers an LA query in $O(\delta)$ time, we notice that the climb operations in lines 1 and 8 require time $O(\delta)$, and so does line 2. Moreover, the query to D_Q (line 6) can also be performed in $O(\delta)$ time as discussed above.

4 LA Queries in Dynamic Trees

In this section we provide our main result for LA queries. In Sect. 5, we show how our ideas can be extended to also handle weighted LA, BVQ, and LCA queries.

4.1 Description of the Data Structure

Some of the key ideas behind our data structure for LA queries in dynamic trees are extensions of the ones used for the static case. Namely, the n nodes of T are colored with either *black* or *white*, the set of *black* nodes has size $O(n/\delta)$, and it corresponds to the vertex set of an auxiliary *black forest* Q. Ideally, in absence of corruptions, Q is exactly the black tree as defined in the static case, namely the tree in which the parent of each (black) node $\overline{v}$ in Q is the vertex $\overline{u}$ associated with the lowest black proper ancestor u of v in T.

Moreover, we would still like the vertices of T having a depth that is a multiple of δ to be colored black, similarly to the static case. However, we can no longer afford to maintain such a rigid coloring scheme since the tree is now being dynamically constructed via successive AddLeaf operations, and the corruptions of the adversary

[7] Here the distance of 2δ is essentially tight as it can be seen, e.g., by considering a tree T consisting of a path of length $2\delta - 2$ rooted in one of its endpoints. The only black vertex of T is the root. Notice how the vertex u at depth δ is white since the subtree of T rooted in u has height $\delta - 2$.

might cause vertices to become miscolored. We however ensure that such a regular coloring pattern will be followed by the portions of T that are sufficiently distant from the corruptions. This allows us to answer LA queries using a strategy similar to the one employed for the static case.

Our data structure stores the following information: The record of a node v maintains, in addition to the pointer p_v to its parent and to the pointer q_v to the corresponding node $\overline{v}$ in Q (if any), an additional field flag_v. Intuitively, flag_v can be thought of as a Boolean value in $\{\bot, \top\}$. The initial value of a flag is $\bot$ and we say that the flag is *unspent*. Spending a flag means setting it to $\top$. We will spend δ of these flags to "pay" for the creation of a new black node. Spent flags will also signal the presence of a nearby black ancestor.

For technical reasons, if flag_v is unspent, we allow for it to be additionally *annotated* with a pair (x, i) where x is (the name of) a node and i is an integer. In practice this amounts to setting flag_v to (x, i), which is logically interpreted as $\bot$. Such an annotated flag is still unspent. This provides an additional safeguard against corruptions that may occur during the execution of our leaf insertion algorithm (see Sect. 4.2).

The node records are stored into a dynamic array $\mathcal{A}$, whose current size n is kept in the safe region of memory. This array supports both elements insertions and random accesses in constant worst-case time.[8]

The pointer p_v is then the index (in $\{0, \ldots, n-1\}$) of the record corresponding to the parent of v in $\mathcal{A}$. Initially, $\mathcal{A}$ only contains the root r of T at index 0. Moreover, we will always store new leaves at the end of $\mathcal{A}$ so that, in absence of corruptions, the index of a vertex v in $\mathcal{A}$ is always smaller than the index of any of its descendants. As a consequence, whenever we observe the index stored in pointer p_v is greater than or equal to the index of v itself, we know that v must have been corrupted by the adversary. We say that the pointers p_v such that $p_v \geq v$ are *invalid* (notice that a corrupted pointer is not necessarily invalid). We find convenient to use the above fact to simplify the handling of corrupted vertices: whenever we encounter an invalid pointer p_v we treat it as being equal to 0, i.e., an invalid pointer p_v always refers to the root r of T. This rule also applies to any read pointer, including those accessed by the `climb`$(\cdot,\cdot)$ procedure already defined in Sect. 3.

Then the (possibly corrupted) contents of $\mathcal{A}$, at any point in time, induce a *noisy tree* $\mathcal{T}$ whose root is r, and the parent of each vertex $v \neq r$ is the vertex pointed by p_v according to the above rule. Clearly, T and $\mathcal{T}$ coincide when there are no corruptions.

Moreover, we store a resilient data structure D_Q that, in addition to the already-defined $\text{LA}_Q(\overline{v}, k)$ query, also supports the following additional operations in $O(\delta)$ time.

> $\text{NewTree}_Q(v)$: Given a vertex v of T, it creates a new tree in the forest Q consisting of a single vertex $\overline{v}$ associated to v, and it returns a pointer to $\overline{v}$.

[8] The standard textbook technique which handles insertions into already full array by moving the current elements into a new array of double capacity already achieves $O(1)$ amortized time per insertion. With some additional technical care, the above bound also holds in the worst case. The idea is to distribute the work needed to move elements into the new array over the insertions operations that would cause the current array to become full (it suffices to move 2 elements per insertion). Using this scheme, at any point in time, each element is stored into a single memory word.

`AddLeaf`$_Q(\overline{u}, v)$: Given a vertex $\overline{u}$ of Q, and a vertex v of T, it creates a new vertex $\overline{v}$ associated to v as a child of $\overline{u}$ in Q. Finally, it returns a pointer to the newly added vertex $\overline{v}$.

This data structure D_Q is the resilient version, obtained using the replication strategy, of the linear-size data structure that supports both the `AddLeaf` operation and `LA` queries in constant time [9]. Notice that D_Q always returns the correct answer to all possible `LA` queries on Q. Moreover, once we ensure that the number of vertices that become black (and hence the size of Q) is always $O(n/\delta)$, we have that the (resilient) data structure D_Q requires $O(n)$ space (this will be shown formally in the proof Theorem 1).

4.2 The `AddLeaf` Operation

Before describing our implementation of the `AddLeaf` operation, it is useful to give some additional definitions. We say that v is *near-a-black* in a tree $\tilde{T}$ if there exists some $k \in \{1, 2, \ldots, \delta\}$ such that the k-parent of v in $\tilde{T}$ is black. Moreover, we say that v is *black-free* in $\tilde{T}$ if no k-parent of v in $\tilde{T}$ for $k \in \{1, 2, \ldots, 2\delta - 1\}$ is black.

The procedure `AddLeaf`(x_{par}) takes a vertex x_{par} of T as input and adds a new child x of x_{par} to T (see Algorithm 2). The record corresponding to new vertex x is appended at the end of the dynamic array $\mathcal{A}$. For simplicity we will assume that, during the execution of `AddLeaf`(x_{par}), the record of vertex x is never corrupted by the adversary. This can be guaranteed without loss of generality since a (temporary) record for x can be kept in safe memory and copied back to $\mathcal{A}$ (which is stored in the unreliable main memory) at the end of the procedure.

Our algorithm consists of a first *discovery* phase and possibly of a second additional *execution* phase. The aim of the discovery phase is that of deciding whether a white ancestor of x_{par} should be colored black following the insertion of x. A necessary condition for this to happen is that the flags of the δ closest proper ancestors of x are unspent. The discovery phase is also responsible of locating the node to recolor and of determining the corresponding black ancestor that becomes its parent in Q (if any). The execution phase takes care of the actual recoloring, updates Q accordingly, and spends the δ unspent flags to "pay" for the creation of the new vertex in Q.

More precisely, the discovery phase of Algorithm 2 explores the current tree by climbing δ levels of $\mathcal{T}$ from the newly inserted node x, reaching a vertex y, and checking during the process that all the flags associated with the traversed nodes are unspent. If any of these flags is spent, we immediately return from the `AddLeaf`(x_{par}) procedure without performing the execution phase. Otherwise, the algorithm climbs $2\delta - 1$ further levels from y to determine whether y appears to be black-free or near-a-black. In the latter case, it keeps track of the distance ℓ from y to the closest black proper ancestor y' of y that is encountered. If y is black-free or near-black we move on to the execution phase, otherwise we return from the `AddLeaf`(x_{par}) procedure (without performing the execution phase). A technical detail of the discovery phase is the following: while climbing from x_{par} to y, the generic i-th unspent flag is annotated with (x, i) (possibly overwriting any existing previous annotation) and will be checked by the execution phase. Recall that these flags remain unspent.

The execution phase once again climbs δ levels of $\mathcal{T}$ staring from x, with the goal of changing the color of an existing white vertex to black (hence creating a corresponding black node in Q). This is guaranteed to happen unless the annotations of the unspent flags of the vertices in the path from x_{par} to y set during the discovery phase reveal that one such vertex has been corrupted in the meantime. The creation of a new black vertex in Q is "paid for" by *spending* these δ unspent flags (i.e., setting to $\top$ the flags of the vertices in the path from x_{par} to y). The position of the new black vertex depends on whether y was near-a-black or black-free. If y was near-a-black, then the vertex y' discovered in the first phase is the δ-parent of the new black vertex x', and a new leaf $\overline{x}'$ is appended to $\overline{y}'$ in Q. Otherwise, if y was black-free, then y becomes black and a new tree containing a single vertex $\overline{y}$ is added to Q. Notice that, if a vertex b is colored black during the `AddLeaf` operation, the execution phase always spends flag_b.

4.3 Analysis of the Data Structure

In this section we analyze our data structure. The core of the analysis is to show that the `AddLeaf` operation in Algorithm 2 guarantees that in $\mathcal{T}$, if we are sufficiently distant from all the corrupted vertices, the black nodes are regularly distributed. The formal property is stated in Lemma 5. We first need to prove some auxiliary properties. In Fig. 3 we give an example that shows that, even in an uncorrupted path, if we are not sufficiently distant from corruptions, the black nodes can form irregular patterns in the path.

The following lemma shows that if the flag of a vertex w appears to be spent, then either there must be a nearby black ancestor of w, unless a nearby corruption occurred. See Fig. 4a.

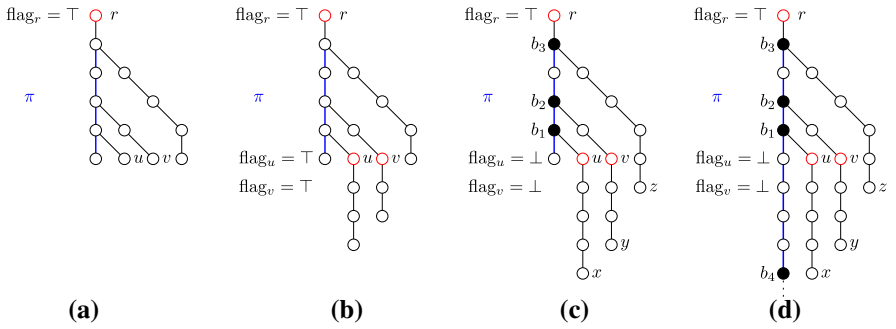

Fig. 3 An example with $\delta = 5$ showing that an uncorrupted path π (depicted in blue) can exhibit an irregular pattern of black vertices (**d**). Situation (**a**) can be reached when the adversary corrupts r by setting $\text{flag}_r = \top$ before the insertions of the other nodes take place. To obtain (**b**), the adversary can set $\text{flag}_u = \text{flag}_v = \top$, thus corrupting u and v before u and v's descendants are inserted. If the adversary sets flag_u and flag_v back to $\bot$ before x, y, and z are inserted (in this order), we arrive in configuration (**c**) in which b_1, b_2, and b_3 have been colored black. Inserting the remaining vertices yields (**d**) (Color figure online)

Algorithm 2: `AddLeaf`(x_{par}): Adds a new leaf x as a child of vertex x_{par} in T. Returns a reference to x.

```
1  Add a new record x at the end of A;  p_x ← x_par;  flag_x ← ⊥;  q_x ← null;
   // Discovery Phase
   // Check the flags of the lowest δ proper ancestors of x
2  y ← x;
3  for i = 1, ..., δ do
4      if y = r then return x;                          // The root r was reached
5      y ← p_y;
6      if flag_y = ⊤ then return x;  // Return immediately if a spent flag is found
7      flag_y ← (x, i);                                          // Annotate flag_y
   // Check whether y is near-a-black.
   // ℓ will store the distance to the closest black
   // ancestor y' of y, if any
8  y' ← y;   ℓ ← 0;   near_black ← false;
9  while ℓ < δ and y' ≠ r and near_black = false do
10     y' ← p_y';   ℓ ← ℓ + 1;
11     if y' is black then near_black ← true;
   // If y is not near-a-black, check whether it is black-free
12 if near_black = false then
13     z ← y';
14     for δ − 1 times do
15         if z = r then break;
16         z ← p_z;
17         if z is black then return x;
   // Execution Phase. Node y was either near-a-black or black-free
   // Acquire the flags of the lowest δ proper ancestors of x
18 z ← x;
19 for i = 1, ..., δ do
20     if z = r then return x;
21     z ← p_z;
22     if flag_z ≠ (x, i) then return x;          // Check the annotation of flag_z
23     if near_black = true and i = ℓ then
24         x' ← z ;                                    // y' is the δ-parent of x'
25     flag_z ← ⊤;                                                // Spend flag_z
26 if near_black = true then
27     q_x' ← AddLeaf_Q(q_y', x');
28 else
29     q_y ← NewTree_Q(y);
30 return x;
```

Lemma 1 *Let w and z be two nodes such that z is the δ-parent of w in T and such that no node in the path π from z to w in T has been corrupted. If* $\text{flag}_w = \top$*, then there exists a black node in $\pi(z : w]$.*

Proof Let x be the node whose insertion in T caused flag_w to be set to $\top$. Moreover, let P be the path of length δ from x to y traversed in the discovery phase of Algorithm 2

in lines 2–7. Similarly, let P' be the path from x to y traversed in the execution phase of Algorithm 2 in lines 18–25.

Clearly, P' contains w. Moreover, if w is the i-th node traversed in P', then $\text{flag}_w = (x, i)$ in the execution phase and (since w is uncorrupted), flag_w was set to (x, i) in the discovery phase. As a consequence, w is also the i-th node in P and $P[w : y] = P'[w : y]$. Hence, y is at distance $\delta - i \leq \delta - 1$ from w in P (and in T) showing that z is a proper ancestor of y. Therefore all nodes in $P'[w : y]$ are uncorrupted, and the loop in in lines 18–25 of Algorithm 2 is executed to completion. This ensures that the execution phase will color a node b black. We distinguish two cases depending on whether y was observed to be near-a-black or black-free in the discovery phase.

If y is black-free, then b is exactly y and the claim follows. Otherwise, y is near-a-black and the discovery phase computed the distance ℓ between y and its closest black proper ancestor. If $\ell \geq i$, then Algorithm 2 colors a vertex in $P[w : z) = \pi[w : z)$ black. Otherwise, if $\ell < i$, the discovery phase observed that the ℓ-parent y' of y was black. Since $i \leq \delta$, y' lies in $\pi[y : z)$. □

Next lemma shows that an uncorrupted path of length at least 3δ must contain a black vertex.

Lemma 2 *Let x and z be nodes in T such that z is the 3δ-parent of x in T and such that no node in the path π from x to z in T has been corrupted. Then, there exists a black node w in $\pi[z : x)$.*

Proof Since no vertex in π has been corrupted, the path π must also belong to the noisy tree $\mathcal{T}$. In the rest of the proof we assume that $\pi[z : x)$ contains no black nodes and show that this leads to a contradiction.

Let y be the δ-parent of x in π and let t_x be the time at which the `AddLeaf(·)` operation that adds x to T is invoked. We know that, at time t_x, there exists no node w in $\pi[y : x)$ such that $\text{flag}_w = \top$ since otherwise Lemma 1 would immediately imply the existence of a black node in $\pi[z : w]$ contradicting the initial assumption. Then, the invocation of Algorithm 2 that inserts x also performs its execution phase.

Moreover, y must be black-free at time t_x, and hence it is colored black during such a phase (refer to the pseudocode of Algorithm 2, and recall that a black-free node is not near-a-black). Since y is not corrupted it must still be black, leading to a contradiction. □

To provide an intuition of the role of next lemma, consider an uncorrupted path π of length between δ and 2δ with a black vertex z on top. While the vertex y at distance δ from z would also be colored black in the static case (since each uncorrupted path contains a black vertex every δ levels), this is not necessarily true in our dynamic data structure. Nevertheless, when the only back vertex in π is z, all flags associated with the descendants of y in π are guaranteed to be unspent. In some sense, the data structure is preparing to recolor the missing black vertex. This will happen once δ unspent flags are available. See Fig. 4b.

Lemma 3 *Let x and z be two nodes in T such that: z is an ancestor of x in T, no node in the path π from z to x in T has been corrupted, and $\delta \leq \|\pi\| < 2\delta$. We have that, immediately after vertex x is inserted, if the only black vertex in π is z then all the nodes w in π at distance at least δ from z in T are such that $\text{flag}_w \neq \top$.*

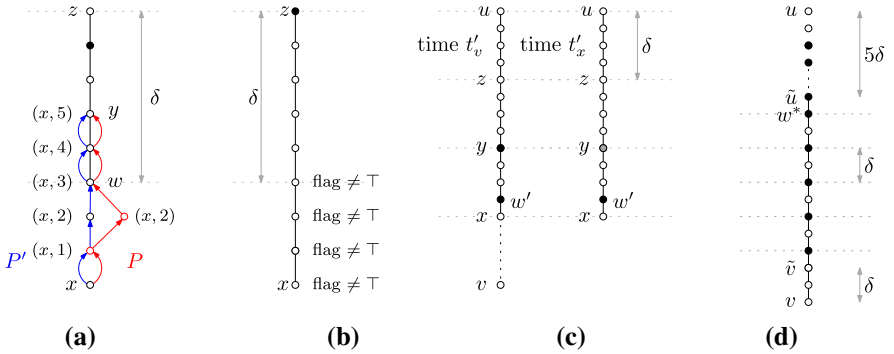

Fig. 4 **a** Graphical representation of the proof of Lemma 1, for $\delta = 5$. **b**, **c**, **d** Representations of the statements of Lemma 3, Lemma 4, and Lemma 5, respectively

Proof Since no vertex in π has been corrupted, the path π must also belong to the noisy tree $\mathcal{T}$. In what follows, we prove that, immediately after vertex x is inserted, the existence of a node w between x and z in π such that $d_{\mathcal{T}}(w, z) \geq \delta$ and $\text{flag}_w = \top$ leads to a contradiction. Indeed, since $\text{flag}_w = \top$, Lemma 1 implies the existence of a black node in $\pi(z : w]$, and this contradicts the fact that z is the only black node in $\pi[z : x]$. □

The next technical lemma is about the time at which the vertices of a long uncorrupted path become black. This will be instrumental to prove Lemma 5. See Fig. 4c.

Lemma 4 *Let u and v be two nodes in T such that u is an ancestor of v, $d_T(u, v) \geq 3\delta$ and no node in the path π from v to u in T has been corrupted. Let y (resp. x) be the node in π at distance 2δ (resp. 3δ) from u in π. Let t'_v (resp. t'_x) be the time immediately after the vertex v (resp. x) is inserted. If the node y is black at time t'_v, then there exists a node w' in $\pi[y : x]$ that is black at time t'_x.*

Proof Since no vertex in π has been corrupted, the path π must also belong to the noisy tree $\mathcal{T}$. In the rest of the proof we assume towards a contradiction that y is black at time t'_v, yet there are no black nodes in $\pi[y : x]$ at time t'_x.

Let z be the δ-parent of y in π. Let $\bar{t}_y$ be the time immediately before y is colored black. At time $\bar{t}_y$ there are only two possible scenarios:

Scenario 1: At time $\bar{t}_y$, the node y is black-free;
Scenario 2: At time $\bar{t}_y$, the node z is the only black node in T in $\pi[z : y]$.

We denote with t_x the time immediately before vertex x is inserted in T and we consider the two scenarios separately. Notice that $\bar{t}_y$ refers to a later time than t_x since y is white at time t'_x by hypothesis. We split scenario 1 into two additional subcases:

Subcase 1.1: at time t_x all the nodes w in $\pi[y : x)$ are such that $\text{flag}_w \neq \top$;
Subcase 1.2: at time t_x there is a node w in $\pi[y : x)$ such that $\text{flag}_w = \top$.

We start considering subcase 1.1. Since $\bar{t}_y$ follows t_x, and y is black-free at time $\bar{t}_y$, vertex y must also be black-free at time t_x. Then, during the insertion of x, Algorithm 2 colors y black yielding a contradiction.

We now analyze subcase 1.2. Since $\text{flag}_w = \top$, Lemma 1 implies the existence of a black node b in $\pi[w, z)$ and, since we assume that there are no black nodes in $\pi[y : x]$, b is in $\pi(z : y)$. This shows that y cannot be black-free at time $\bar{t}_y$ and contradicts the hypothesis of scenario 1.1.

We now consider Scenario 2, which we subdivide into three subcases:

Subcase 2.1: at time t_x all the nodes w in $\pi[y : x)$ are such that $\text{flag}_w \neq \top$ and z is white;
Subcase 2.2: at time t_x all the nodes w in $\pi[y : x)$ are such that $\text{flag}_w \neq \top$ and z is black;
Subcase 2.3: at time t_x there is a node w in $\pi[y : x)$ such that $\text{flag}_w = \top$.

We start by handling subcase 2.1. For the initial assumption, and for definition of this case, we have that there are no black nodes in $\pi[z : x]$ at time t_x. Since z is colored black at some time $\bar{t}_z$ following t_x, we know that the $\delta - 1$ nodes ancestor of z are not black at time t_x, since this is incompatible with the fact that z will become black. Since π is not corrupted, we know that y is black-free in T at time t_x. This implies that y is colored black during the insertion of x in T, and hence y is black at time t'_x contradicting our hypotheses.

We proceed by analyzing subcase 2.2. At time $\bar{t}_y$ all nodes in $\pi[z : y]$, except for z, are white and hence the same is true at time t_x. Since z is black at time t_x and $\text{flag}_w \neq \top$ for all nodes w in $\pi[y : x)$, the `AddLeaf` procedure adding x will color y black. Hence y is black at time t'_x. This is a contradiction.

We now consider subcase 2.3. Together with Lemma 1, $\text{flag}_w = \top$ implies the existence of a black node b in $\pi(z : w]$. Since we assume all the nodes in $\pi[y : x]$ to be white, the black node b is in $\pi(z, y)$, contradicting the hypothesis of scenario 2. □

Now, we are ready to prove our main property about the pattern of black vertices discussed at the beginning of this section. See Fig. 4d.

Lemma 5 *Let u and v be two nodes in T such that u is an ancestor of v, the distance between u and v is at least 7δ, and no node in the path π from u to v has been corrupted. Let $\tilde{u}$ be the node at distance 5δ from u in π and let $\tilde{v}$ be the node at distance δ from v in π. Then there is a black node w^* in $\pi[\tilde{u} : v]$ such that:*

- *The distance between w^* and $\tilde{u}$ is at most δ.*
- *A generic node in $\pi[w^* : \tilde{v}]$ at distance d from w^* is black iff d is a multiple of δ. Moreover, if w is a black vertex in $\pi(w^*, \tilde{v}]$ and $\overline{w}$ is the associated black vertex in Q, the parent of $\overline{w}$ in Q corresponds to the δ-parent of w in π.*

Proof Since no vertex in π has been corrupted, the path π must also belong to the noisy tree $\mathcal{T}$. Then, Lemma 2 ensures that, at any time following the insertion of $\tilde{u}$ in T, there exists a black ancestor y of $\tilde{u}$ such that $d_\pi(y, \tilde{u}) \leq 3\delta$. Such a vertex y is the δ-parent of some vertex x in π. We denote by u' the 2δ-parent of y in π and by t'_x the time immediately after x is inserted. Since the length of $\pi[u' : v]$ is at least 3δ and y must be black when v is inserted, we can invoke Lemma 4 to conclude that there exists a node in $\pi[y : x]$ that is black at time t'_x. We choose w_0 as the closest ancestor of x that is black at time t'_x. Moreover, for $i = 1, \ldots, \lfloor \|\pi[w_0 : v]\|/\delta \rfloor$ we

let w_i be the unique vertex at distance δi from w_0 in $\pi[w_0, v]$. Finally, let t'_i be the time immediately after the insertion of w_i into T.

We will prove by induction on $i \geq 1$ that (i) at time t'_i, all vertices $w_0, w_1, \ldots, w_{i-1}$ are black; (ii) from time t'_i onward, all vertices in $\pi[w_0, w_i]$ that do not belong to $\{w_0, w_1, \ldots, w_i\}$ are white.

We start by considering the base case $i = 1$. Regarding (i), we know that w_0 is black at time t'_x, and hence w_0 is also black at time t'_1 (which cannot precede t'_x). Regarding (ii), by our choice of w_0 we know that at time t'_x, the only black vertex in $\pi[w_0, x]$ is w_0. Moreover, Algorithm 2 can only color a node b black if none of the $\delta - 1$ lowest proper ancestors of b is black. This implies that no vertex in $\pi(w_0, w_1)$ will be colored black.

We now assume that the claim is true up to $i \geq 1$ and prove it for $i + 1$. We first argue that the following property holds: (*) at time t'_{i+1} all vertices in $\pi(w_i : w_{i+1})$ are white. Indeed, suppose towards a contradiction that there exists some black vertex b in $\pi(w_i : w_{i+1})$ at time t'_{i+1}. When b was colored black, either its δ-parent b' was black or b was black-free. In the former case we immediately have a contradiction since b' must be a vertex of $\pi(w_{i-1}, w_i)$ but all such vertices are white by the induction hypothesis. In the latter case b must have been colored black after the insertion of w_i but, by the induction hypothesis, we know that from time t'_i onwards w_{i-1} is black. This contradicts the hypothesis that b was black-free.

Next, we prove (i). Suppose towards a contradiction that w_i is white at time t'_{i+1}. Then, using (*) and the induction hypothesis, we can invoke Lemma 3 on the subpath of π between w_{i-1} and the parent of w_{i+1} to conclude that all nodes w in $\pi[w_i : w_{i+1})$ are such that $\text{flag}_w \neq \top$. Hence, during the insertion of w_{i+1}, Algorithm 2 reaches line 7 and checks whether w_i is near-a-black. Since this is indeed the case, a new black vertex is created in $\pi[w_i : w_{i+1})$, providing the sought contradiction. Let $\overline{w}_i$ (resp. $\overline{w}_{i-1}$) be the vertex in Q associated with w_i (resp. w_{i-1}). Notice that this argument also shows that, at time t'_{i+1}, $\overline{w}_i$ is a child of $\overline{w}_{i-1}$ in Q since w_i becomes black after time t'_i and not later than time t'_{i+1}, when w_{i-1} was already black.

To prove (ii) it suffices to notice that, by inductive hypothesis, we only need to argue about the nodes in $\pi(w_i : w_{i+1})$. From (*) we know that these nodes are white at time t'_{i+1}, while (i) ensures that w_i is black at time t'_{i+1}. Then, a similar argument to the one used in the base case shows that Algorithm 2 will never color any node in $\pi(w_i : w_{i+1})$ black (as long as the nodes in π remain uncorrupted). This concludes the proof by induction.

Let w' be the node at distance δ from $\tilde{u}$ in $\pi[\tilde{u} : v]$. Notice that w_0 belongs to $\pi[u : w']$. If w_0 lies in $\pi(\tilde{u} : w']$, we can choose $w^* = w_0$. Otherwise, w_0 is an ancestor of $\tilde{u}$ and, from (i) and (ii), there is exactly one black vertex b in $\pi(\tilde{u} : w']$ and we choose $w^* = b$. □

4.4 LA Queries

Lemma 5 suggests a natural query algorithm. The query procedure is similar to the one for static case. When $k \leq 7\delta$ we climb in $\mathcal{T}$ the nodes of the path from v to the k-parent of v in a trivial way. Otherwise, Lemma 5 ensures that if no vertex in the path

Algorithm 3: Answers a level ancestor query $\texttt{LA}(v, k)$ in dynamic trees.

```
1  if k ≤ 7δ then
2      return climb(v, k);
3  ṽ ← climb(v, δ);
4  Climb up the tree T from ṽ for up to δ nodes searching a black node;
5  if the previous procedure did not find a black node then
6      return error;
7  v′ ← a black node found in the previous procedure;
8  d ← distance between v′ and v;
9  k′ ← k − d − 5δ;
10 u′ ← LA_Q(q_{v′}, ⌊k′/δ⌋);
11 k_rest ← k′ − ⌊k′/δ⌋ · δ + 5δ;
12 return climb(u′, k_rest) ;
```

P from v to its level ancestor in T was corrupted by the adversary, then every other δ-th vertex of P is colored black except, possibly, for an initial subpath of length δ and for a trailing subpath of length 5δ. The query procedure explicitly "climbs" these portions of P and queries D_Q to quickly skip over its remaining "middle" part. The pseudo-code is given in Algorithm 3.
We are now ready to prove the main theorem of this section.

Theorem 1 *Our data structure requires linear space, supports the* `AddLeaf` *operation in $O(\delta)$ worst-case time, and can answer resilient* `LA` *queries in $O(\delta)$ worst-case time.*

Proof The correctness of the query immediately follows from Lemma 5. Moreover, the time required to perform an `AddLeaf` or an `LA` operation is $O(\delta)$ since in both cases $O(\delta)$ vertices of $\mathcal{T}$ are visited and a single $O(\delta)$-time operation involving D_Q is performed.

We now discuss the size of our data structure. Clearly, the space used to store the array $\mathcal{A}$ of all records is $O(n)$. We only need to argue about the size of D_Q. Recall that D_Q is the resilient version, obtained using the replication strategy, of the data structure of [9] that requires linear space and takes constant time to answer each `LA` query and to perform each `AddLeaf` operation. In order to show that D_Q requires $O(n)$ space we will argue that the number of black vertices is $O(\frac{n}{\delta})$. As consequence we have that the size of D_Q is $O(n)$.

To bound the number of vertices in Q, notice that in order to add a new vertex to Q we need to spend δ flags that were previously unspent. Moreover, a spent flag never becomes unspent unless the adversary corrupts the record of the corresponding node (by using one of its δ corruptions). As a consequence the nodes in Q are at most $(n + \delta)/\delta = O(n/\delta)$. □

5 Handling Weighted LA, LCA, and BVQ Queries

5.1 Weighted LA Queries

In this section we show how to handle weighted LA queries when δ and the weights of the nodes are polylogarithmically-bounded positive integers. Recall that, the answer to a weighted LA query $\texttt{LA}(v, k)$ is the deepest ancestor u of v in T such that the total weight of the vertices in the path from u to v in T is *at least* k. The record of each node v stores, along with the fields described in Sect. 4, an additional field containing the weight of v. To store Q we use a resilient data structure D_Q that maintains a forest of rooted trees in which every vertex has an associated weight. D_Q is also able to answer weighted LA queries on Q in $O(\delta)$ time. For technical convenience we assume that a weighted level-ancestor query $\texttt{LA}_Q(v, k)$ in Q reports the vertex u of minimum depth among the ancestors of v such that the total weight of the vertices in the path between u and v (endpoints included) in Q is *at most* k. This data structure is the resilient version, obtained using the replication strategy, of the one in [9] which answers weighted LA queries in constant time when vertex weights are polylogarithmically-bounded.

We modify Algorithm 2 in two ways: (i) during the discovery phase, we keep track of the total weight W of the vertices between x (included) and the closest black proper ancestor y' of y (y' excluded); (ii) during the execution phase, we keep track of the total weight W' of the vertices between x (included) and x' (excluded). Recall that when a vertex x' becomes black in the execution phase of Algorithm 2 since y was observed to be near-a-black in the discovery phase, the corresponding vertex $\overline{x}'$ is added to Q via the $\texttt{AddLeaf}_Q$ operation on line 27. To handle weighted LA queries, we also need to assign a weight the the new vertex $\overline{x}'$. Specifically, we choose this weight to be $W - W'$. Notice that, in the absence of corruptions, $W - W'$ is exactly the total weight of the vertices in path between x' (included) and y' (excluded). Moreover, when a vertex y becomes black in the execution phase of Algorithm 2 because it was observed to be black-free in the discovery phase, we set the weight of the corresponding node $\overline{y}$ in Q to the weight of y in T.[9]

We now describe how to answer a query $\texttt{LA}(v, k)$. We start by optimistically assuming that the (unweighted) distance between v and the sought vertex is short. We do so by climbing (up to) 10δ levels from v while keeping track of the total weight of the traversed vertices. We stop at and return the first encountered vertex for which such a weight is at least k.

If the above procedure is unable to locate the sought vertex, we proceed as follows. We climb δ levels from v, and we then search for a nearby black node b among the closest δ proper ancestors of the reached vertex. During this process, we keep track of the total weight W of the traversed nodes between v (included) and b (excluded). Let $\overline{b}$ be the vertex in Q that is associated with b. We now perform an $\texttt{LA}_Q(\overline{b}, k - W)$ query D_Q to find the shallowest ancestor $\overline{b}'$ of $\overline{b}$ such that the overall weight W' of

[9] Notice that, in absence of corruptions, the weight assigned to a vertex in Q is always positive and polylogarithmically-bounded since so is δ. The above property might not be true when the observed values are corrupted but we artificially enforce it by constraining weights to be in this range. This also applies to weights read by the query algorithm explained in the following.

the vertices in the path between $\overline{b}'$ and $\overline{b}$ in Q is at most $k - W$. Let b' be the node of T that is associated to $\overline{b}'$. Finally, we iteratively climb from b' towards its ancestors until we reach a vertex u such that the total weight of the path between b' (excluded) to u (included) is at least $k - W - W'$. We then return u. As we argue below, this final climbing procedure requires at most 6δ steps in the absence of corruptions. Therefore, if this threshold is exceeded we immediately stop the query and report an error.

We now discuss the correctness of the query. Let u^* be the deepest ancestor of v in T such that the total weight between v and u^* is at least k, and assume that the path π between u^* and v is uncorrupted. To prove that the query procedure is correct it is sufficient to show that (i) the vertex b' belongs π, and (ii) the (unweighted) length of $\pi[u^* : b']$ is at most 6δ.

To see (i), assume by contradiction that b' is not in π, and let $\overline{b}_1$ be the deepest ancestor of $\overline{b}$ in Q such that the vertex in T corresponding to the parent $\overline{b}_2$ of $\overline{b}_1$ in Q does not belong to π. Let b_2 be the vertex in T associated to $\overline{b}_2$ (vertex $\overline{b}_2$ must exists since we assumed that b' is not in π). As a consequence, since π is uncorrupted, the total weight of the path in Q between $\overline{b}_1$ (excluded) and $\overline{b}$ is equal to the total weight of $\pi(b_1, b]$. Moreover, the weight of $\overline{b}_1$ in Q is at least the total weight of $\pi[u^* : b_1]$. This implies that W' must be strictly greater than $k - W$ since $\overline{b}_2$ has weight at least 1. This is a contradiction.

It remains to prove (ii). Since the number of vertices of π is at least 10δ, we invoke Lemma 5 to conclude that b' must be at distance at most 6δ from u^*. Indeed, if this was not the case, then the δ-parent of b' would be black and would belong to π, which implies that $\overline{b}'$ could not be the vertex returned by the query on D_Q.

5.2 BVQ Queries

To support BVQ queries, the record of each node v stores the weight of v and maintains an additional field depth_v that intuitively keeps track of the depth of v in T. Initially, when T is first built and consists only of the root r, we set $\text{depth}_r = 0$. Whenever a new node v is appended as a child of u via the AddLeaf operation, we set $\text{depth}_v = \text{depth}_u + 1$.

To store Q we use a resilient data structure D_Q that maintains a forest of rooted trees which can be updated by adding leaves in $O(\delta)$ time per operation. D_Q is also able to answer (unweighted) LA and BVQ queries on Q in $O(\delta)$ time. This data structure is the resilient version, obtained using the replication strategy, of the one in [9] which answers both LA and BVQ queries in constant time.

Moreover, we slightly modify the execution phase of Algorithm 2 in the case in which y was observed to be near-a-black in the discover phase. In this scenario a vertex x' becomes black, and a corresponding vertex $\overline{x}'$ is added to Q via the $\texttt{AddLeaf}_Q$ operation on line 27. In our modification, we additionally climb the path from x' (included) to y' (excluded) while keeping track of the vertex w' of minimum weight among the encountered nodes. We assign the weight of w' to $\overline{x}'$ in Q and we store a reference to w' as (replicated) satellite data attached to $\overline{x}'$. When instead the vertex that becomes black is y since y was observed to be black free during the discovery

phase, we assign weight $+\infty$ to the corresponding black node $\overline{y}$ in Q (in this case $\overline{y}$ is the root of a new tree in Q, and no satellite data is needed).

We now describe how to answer a $\texttt{BVQ}(u, v)$ query. In particular, we only need to consider the case in which u is an ancestor of v since we can always perform an $\texttt{LCA}(u, v)$ query (we will show how to handle LCA queries later in this section) to find the lowest common ancestor z of u and v in T and then return the minimum of the two bottleneck queries $\texttt{BVQ}(z, u)$ and $\texttt{BVQ}(z, v)$ which satisfy the above requirement.

Hence we assume that no corrupted vertex exists in the path π from u to v in T and we start by computing the quantity $d = \text{depth}_u - \text{depth}_v$. Notice that, while depth_u (and depth_v) might not contain the actual depth of u (and v) in T, due to a corruption in some ancestor of u, the value of d always matches the distance between u and v in T, i.e., the length of π.[10]

If $d < 10\delta$, we answer the query using the trivial algorithm $\texttt{BVQ}(u, v)$ that climbs the path π one edge at a time from v to u and returns the vertex of minimum weight encountered in the process. Clearly this algorithm is resilient and requires $O(\delta)$ time. Otherwise, we use a strategy similar to the one used for LA queries in Algorithm 3. We climb δ levels from v, and we then search for a nearby black node b among the δ ancestors of the reached vertex. During this process, we keep track of the node w_1 of minimum weight among those we encounter. Next, we perform an LA query on D_Q to find the black node b' that is $(\lfloor d/\delta \rfloor - 7)$-parent of b (Lemma 5 ensures that this vertex exists and is black). Finally, we climb from b' to u in $O(\delta)$ time and keep track of a node w_2 having minimum weight among those encountered during the process. The answer to $\texttt{BVQ}(u, v)$ is the vertex of minimum weight between w_1, w_2, and the node returned by a bottleneck vertex query $\texttt{BVQ}(b', b)$ in Q.

5.3 LCA Queries

In this section we show how to handle an $\texttt{LCA}(u, v)$ query. We first discuss the rough idea of our solution. Intuitively, we use two separate approaches to manage *long* and *short* queries, respectively. We say that a query is *short* if at least one of u and v is at a distance of at most 10δ from the lowest common ancestor w of u and v in T, and *long* otherwise. As we will show, short queries can be easily answered using a combination of LA queries and a procedure similar to the climbing strategy of Sect. 3. Hence, the main technical difficulty lies in handling long queries. To get an intuition of our approach, consider the case in which both u and v are black vertices and there are no corruptions. Let $\overline{y}'$ be the LCA of $\overline{u}$ and $\overline{v}$ in Q and consider the first vertex $\overline{a}$ (resp. $\overline{b}$) in the unique path from $\overline{y}'$ to $\overline{u}$ (resp. $\overline{v}$) in Q. Then, w is also the lowest common ancestor of a and b in T, and y' is an ancestor of w (see Fig. 5). We can hence reduce the long query $\texttt{LCA}(u, v)$ to the short query $\texttt{LCA}(a, b)$.

However, in presence of corruptions, the nodes in the path between y' (included) and w (excluded) in T might be corrupted, and this could prevent us from discovering the "right" nodes $\overline{a}$ and $\overline{b}$. For example, this can happen when the vertices $\overline{u}$ and $\overline{v}$

[10] The adversary could cause the values stored in depth_u or depth_v to overflow. However, with some additional technical care (by interpreting these fields as unsigned integers in modular arithmetic) one can always recover d.

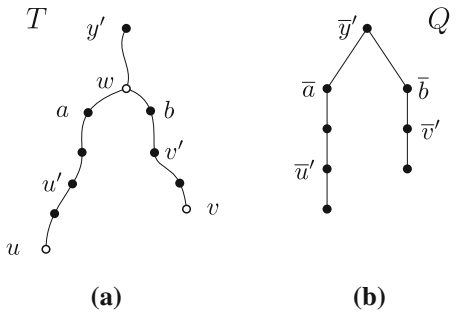

Fig. 5 **a** A representation of the topological relationship between significant vertices used to answer an LCA query in T, as discussed in Sect. 5.3. **b** The corresponding black tree Q

belong to different trees in Q. Nevertheless, with some technical care, we can ensure that at least one of $\overline{a}$ and $\overline{b}$ is the root of the corresponding tree in Q, and the associated vertex in T (i.e., either a or b) is a close descendant of w. This will suffice to reduce to the case of a short LCA query.

We now formally describe our data structure for LCA queries. The record of each node v stores, along with the fields described in Sect. 4, a field $\texttt{depth}_v$ managed as discussed for the BVQ query, and an additional field $\texttt{cba}_v$ which intuitively stores a pointer to the vertex in Q associated with the closest black ancestor of v in T. When v is inserted $\texttt{cba}_v$ is *unset*, and it will be possibly set during the execution phase of some later AddLeaf operation. Similarly to $\texttt{flag}_v$, we allow a field $\texttt{cba}_v$ that is unset to be annotated with a pair (x, i), where x is a vertex that is being inserted and i is the observed distance between x and v.

To store Q we use a resilient data structure D_Q that maintains a forest of rooted trees which can be updated by adding leaves in $O(\delta)$ time per operation. D_Q is also able to answer LA and LCA queries on Q in $O(\delta)$ time. This data structure can be built as combination of the resilient versions (obtained through the replication strategy) of the data structures in [9, 12] which answer LA and LCA queries in constant time.

We modify both the discovery and the execution phases of Algorithm 2. Recall that in the discovery phase the algorithm locates the δ-parent y of x, and the closest black proper ancestor y' of y, if any. In our modification, when we traverse a generic ancestor z of the inserted vertex x, we check $\texttt{cba}_z$. If $\texttt{cba}_z$ is unset, we annotate it with (x, i) where i is the observed distance between x and z (possibly overwriting any previous annotation). Moreover, we also store $q_{y'}$ in a variable $\overline{y}'$ in safe memory. In the execution phase, we only need to handle the case in which y appeared to be near-a-black during the discovery phase. In this case, let x' be the vertex such that y' is the δ-parent of x' (see line 24). We extend the for loop of line 18 in order to reach y'. We still check and spend the encountered flags only for the fist δ vertices, as before. In addition, for each vertex z at distance i from x, such that z is in the path between x' (included) and y' (excluded), we check that $\texttt{cba}_z$ is either set to $\overline{y}'$ or it is unset and (correctly) annotated with (x, i). In the latter case, we set $\texttt{cba}_z$ to $\overline{y}'$. If neither of the previous conditions is met (i.e., $\texttt{cba}_z$ is set to some vertex other than $\overline{y}'$ or it is unset and incorrectly annotated) we are in an *exceptional situation* and $\texttt{cba}_z$ is left unaltered.

Finally, we modify line 27 in which x' is colored black via the addition of a corresponding vertex $\overline{x}'$ to Q. Our modification is as follows: if we are not in an exceptional situation, we proceed as before and we add $\overline{x}'$ as child of $\overline{y}'$ in Q. Otherwise, in the exceptional situation, we add $\overline{x}'$ as a new root in Q.

Before describing how to answer an `LCA` query, we argue that the above modifications guarantee stronger structural properties than the ones given in Sect. 4. In particular, we start by showing that Lemma 5 still holds. First of all, notice that our modifications do not affect vertex colors. Hence, we only need to show that the parent-child relationships between black vertices in Q are preserved. Since the only way to alter these relationships is for an exceptional situation to happen during the execution of `AddLeaf` that colors some node x' black, we only need to show that no exceptional situation can arise when a (sufficiently deep) vertex of an uncorrupted path becomes black. This is proven in the following lemma.

Lemma 6 *Let π be an ancestor-descendant path in T of length at least 2δ, and let x' be the deepest vertex of π. If x' is black and no vertex in π has been corrupted, then the execution of `AddLeaf` that colored x' black did not encounter an exceptional situation.*

Proof Let x be the node whose insertion in T causes x' to be colored black, and let t_x the time immediately before x is inserted in T. In the rest of the proof, we assume that the execution of `AddLeaf` inserting x in T is in an exceptional situation, and we prove that this leads to a contradiction.

Since we are in an exceptional situation, at time t_x the δ-parent y' of x' must be black and all the other nodes in $\pi(y' : x')$ must be white. Let $\overline{y}' = q_{y'}$. Then, the exceptional situation was caused by a node w in $\pi(y' : x']$ such that $\texttt{cba}_w = \overline{z}$ and $\overline{z} \neq \overline{y}'$. Let z be the node in T that is associated with vertex $\overline{z}$ and notice that $\overline{z} \neq \overline{y}'$ implies $z \neq y'$. Since no vertex in π is corrupted, the existence of w implies the existence of an ancestor z of w which is black at time t_x and such that $d(z, w) \leq \delta$ (see Fig. 6a). By hypothesis, all the nodes in $\pi(y' : x')$ are white at time t_x, and hence $z \neq y'$ must be a proper ancestor of y'. Node z satisfies the following conditions: (i) $d(y', z) \leq \delta - 1$ (since $d(w, z) \leq \delta$), and (ii) y' was white when $\texttt{cba}_w$ was set to $\overline{z}$ (since $\texttt{cba}_w = \overline{z}$ and no vertex in π is corrupted). This implies that, when y' was colored black, there was a black node z such that $d(y', z) \leq \delta - 1$ and this is a contradiction. □

We now prove a structural property that will be exploited in the query procedure. More precisely, let us assume that we need to answer a long `LCA`(u, v) query, that the path π between u and v in T is uncorrupted, and let w be the lowest common ancestor of u and v. We use the vertices u and v to pinpoint two new vertices in Q, respectively named $\overline{a}$ and $\overline{b}$ (and their corresponding vertices a, b in T). We define $\overline{a}$ (resp. $\overline{b}$) as the value $\overline{z}$ at the end of the following algorithm: we first climb δ levels from u (resp. v) in T and then we search for the closest black proper ancestor of the current vertex (Lemma 5 ensures that such a black vertex exists and is at distance at most δ). We initialize $\overline{z}$ as the vertex in Q corresponding to such an ancestor. Next, we iteratively move from the current vertex $\overline{z}$ to its parent until we reach the first vertex that is either the root of its tree, or has a parent whose corresponding vertex z in T does not lie in $\pi(w : u]$ (resp. $\pi(w : v]$).

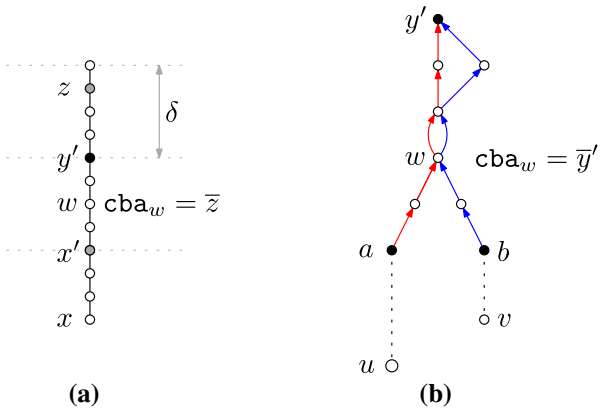

Fig. 6 **a** Graphical representation of the proof of Lemma 6, for $\delta = 4$. **b** Graphical representation of the proof of Lemma 7, for $\delta = 5$. Here we are considering a long query $\texttt{LCA}(u, v)$ and the path from u to v is uncorrupted. Notice how Lemma 7 still holds even if the parent of w is corrupted, causing the subsequent invocations of the `AddLeaf` procedure to observe two different (sub)paths from w to y'

We prove the following lemma.

Lemma 7 *The distance between w and a (resp. b) in T is at most 6δ. Moreover, if both $\overline{a}$ and $\overline{b}$ have a parent in Q, then such parents coincide.*

Proof The bound on the distance between w and a (resp. b) in T immediately follows from Lemma 5, hence in the rest of the proof we focus on showing that the parents of $\overline{a}$ and $\overline{b}$ (if they exist) must coincide.

Assume, w.l.o.g., that $\overline{a}$ was inserted in Q before $\overline{b}$, and let $\overline{y}'$ be the parent of $\overline{a}$ in Q. Notice that, when a was colored black, the corresponding `AddLeaf` operation inserted vertex $\overline{a}$ as a child of $\overline{y}'$. Hence, the black vertex y' in T corresponding to $\overline{y}'$ was observed to be an ancestor of both a and w (by the choice of $\overline{a}$) in the discovery phase. As a consequence, the execution phase ensured that the value of $\texttt{cba}_w$ was exactly $\overline{y}'$ (as otherwise the `AddLeaf` operation would have encountered an exceptional situation and $\overline{a}$ would have been a root of a tree in Q). Analogously, the `AddLeaf` operation that colored b black was not in an exceptional situation and, by the definition of $\overline{b}$, we know that w lies in the (observed) path between b and its (observed) δ-parent z. Since w is uncorrupted, and the above operation successfully checked that $\texttt{cba}_w$ matched q_z, we can conclude that $q_z = \overline{y}'$. Hence, the parent of $\overline{b}$ in Q is $\overline{y}'$. See Fig. 6b for an example. □

We are now ready to describe how to answer an $\texttt{LCA}(u, v)$ query. We first describe a simple resilient *naive strategy* to answer $\texttt{LCA}(u, v)$ queries. This strategy always returns an answer when the query is short and π is uncorrupted, while it might be *inconclusive* when the query is long or some vertex of π is corrupted. If an answer is provided and π is uncorrupted, then the returned vertex will always be the LCA w between u and v.

Let k be the difference between the depth of v and the depth of u.[11] We describe the case $k \geq 0$ (the case $k < 0$ is symmetric). We perform an $\texttt{LA}(v, k)$ query to find the k-parent v' of v. Notice that, in absence of corruptions on π, the distance between w and u is the same as the distance between w and v'. We now iteratively perform the following steps. We check whether $v' = u$ and, if this is the case, we answer the query by reporting v' as the sought lowest common ancestor. Otherwise, we move u and v' to their respective parents and repeat. If the parent of u or v does not exist or we are unable to answer the query within 10δ iterations, we stop the above procedure and say that the naive strategy is inconclusive.

We now need to handle the case in which the naive strategy is inconclusive, we hence assume that π is uncorrupted and that the query $\texttt{LCA}(u, v)$ is long. Let u' (resp. v') be a black ancestor of u (resp v) computed as follows. We first climb δ levels from u (resp. v) and then climb again in order to reach the first black proper ancestor of the current vertex. Notice that, by Lemma 5, u' (resp. v') is at distance at most 2δ from u (resp. v).

Let $\overline{u}'$ and $\overline{v}'$ be the vertices in Q corresponding to u' and v', respectively. We perform an LCA query in Q to find the lowest common ancestor $\overline{y}'$ of $\overline{u}'$ and $\overline{v}'$, if any. We assume also that, if such a vertex exists, this query is able to return the two vertices $\tilde{a}$ and $\tilde{b}$ of Q such that $\tilde{a}$ (resp. $\tilde{b}$) lies in the path between $\overline{y}'$ and $\overline{u}'$ (resp. between $\overline{y}'$ and $\overline{v}'$) in Q, and $\overline{y}'$ is the parent of both $\tilde{a}$ and $\tilde{b}$.[12] Notice that from Lemma 7, it must be $\overline{a} = \tilde{a}$ and $\overline{b} = \tilde{b}$. If $\overline{y}'$ exists, we return the outcome of the naive LCA strategy on a and b, where a (resp. b) is the black vertex in T corresponding to $\overline{a}$ (resp. $\overline{b}$). Lemma 7 ensures that the vertices a and b are close descendants of w, and hence the naive query correctly finds w.

It remains to handle the case in which there is no LCA between $\overline{v}'$ and $\overline{u}'$ in Q, i.e., $\overline{u}'$ and $\overline{v}'$ belong to different trees of Q. In this case, we let $\overline{a}'$ (resp. $\overline{b}'$) be the root of the tree in Q that contains $\overline{u}'$ (resp. $\overline{v}'$). From Lemma 7, it must be that $\overline{a}' = \overline{a}$ or $\overline{b}' = \overline{b}$ (possibly both). In this case, we inspect the fields $\text{depth}_{u'}$, $\text{depth}_{v'}$, $\text{depth}_{a'}$ and $\text{depth}_{b'}$ and we consider the vertex among a' and b' that appears to be deeper in T.[13] W.l.o.g., let a' be such vertex and let $k_{a'}$ be the observed difference in levels between u' and a'. We check that $k_{a'}$ is non-negative and that $\texttt{LA}(u', k_{a'}) = a'$. If the above condition is met, we use the naive strategy to answer an LCA query between a' and v' and return the resulting vertex (notice that this cannot be inconclusive). Otherwise, if $k_{a'} < 0$ or the answer to $\texttt{LA}(u', k_{a'})$ was not a', we must have $b' = b$ and we return the vertex found by using the naive strategy to answer the short LCA query between b' and u'.

[11] Similarly to the case of BVQ queries, this value can be recovered from depth_v and depth_u.

[12] It is easy to support such a query with a constant number of LCA and LA queries on Q.

[13] In order to carefully manage possible overflows and corruptions, this can be done by first recovering the difference $\Delta_{v',u'}$ between $\text{depth}_{v'}$ and $\text{depth}_{u'}$, the difference $\Delta_{u',a'}$ between $\text{depth}_{u'}$ and $\text{depth}_{a'}$, and the difference $\Delta_{v',b'}$ between $\text{depth}_{v'}$ and $\text{depth}_{b'}$. Then, it suffices to compare $\Delta_{u',a'} + \Delta_{v',u'}$ with $\Delta_{v',b'}$.

Author Contributions All authors contributed equally to this work.

Funding Open access funding provided by Università degli Studi dell L'Aquila within the CRUI-CARE Agreement. Partial financial support was received by Luciano Gualà under Project No. E89C20000620005 "ALgorithmic aspects of BLOckchain TECHnology" (ALBLOTECH) of the University of Rome "Tor Vergata". No additional funding was received to assist with the preparation of this manuscript.

Declarations

Conflict of interest The authors have no competing interests to declare that are relevant to the content of this article.

Ethical Approval The study did not involve and did not use data from any of the following: human subjects, animals or biological material, plants, algae, fungi, palaeontological specimens, or geological samples.

Consent to Participate The study did not involve human subjects.

Consent for Publication The study did not involve human subjects.

References

1. Finocchi, I., Italiano, G.F.: Sorting and searching in the presence of memory faults (without redundancy). In: Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing, STOC'04. New York, NY, USA: Association for Computing Machinery, pp. 101–110 (2004)
2. Finocchi, I., Grandoni, F., Italiano, G.F.: Resilient search trees. In: Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA'07. USA: Society for Industrial and Applied Mathematics, pp. 547–553 (2007)
3. Brodal, G.S., Fagerberg, R., Finocchi, I., Grandoni, F., Italiano, G.F., Jørgensen, A.G., et al.: Optimal resilient dynamic dictionaries. In: Arge, L., Hoffmann, M., Welzl, E. (eds.) Algorithms—ESA 2007, pp. 347–358. Springer Berlin Heidelberg, Berlin (2007)
4. Jørgensen, A.G., Moruz, G., Mølhave, T.: Priority queues resilient to memory faults. In: Dehne, F.K.H.A., Sack, J., Zeh, N. (eds.) Proceedings of the 10th International Workshop on Algorithms and Data Structures (WADS'07). Lecture Notes in Computer Science, vol. 4619, pp. 127–138. Springer, Berlin (2007)
5. Italiano, G.F.: Resilient algorithms and data structures. In: Calamoneri, T., Díaz, J. (eds.) Algorithms and Complexity, Proceedings of the 7th International Conference, CIAC 2010, Rome, Italy, May 26-28, 2010. Lecture Notes in Computer Science, vol. 6078, pp. 13–24. Springer, Berlin (2010)
6. Petrillo, U.F., Grandoni, F., Italiano, G.F.: Data structures resilient to memory faults: an experimental study of dictionaries. ACM J. Exp. Algorithmics **8**, 2444022 (2013). https://doi.org/10.1145/2444016.2444022
7. Leucci, S., Liu, C., Meierhans, S.: Resilient dictionaries for randomly unreliable memory. In: Bender, M.A., Svensson, O., Herman, G. (eds.) 27th Annual European Symposium on Algorithms, ESA 2019, September 9-11, 2019, Munich/Garching, Germany. LIPIcs, vol. 144, p. 70:1-70:16. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, Wadern (2019)

8. Dietz, P.F.: Finding level-ancestors in dynamic trees. In: Dehne, F., Sack, J.R., Santoro, N. (eds.) Algorithms and Data Structures, pp. 32–40. Springer Berlin Heidelberg, Berlin (1991)
9. Alstrup, S., Holm, J.: Improved algorithms for finding level ancestors in dynamic trees. In: Proceedings of the 27th International Colloquium on Automata, Languages and Programming, ICALP'00. Berlin, Heidelberg: Springer, pp. 73–84 (2000)
10. Demaine, E.D., Landau, G.M., Weimann, O.: On Cartesian trees and range minimum queries. Algorithmica **68**(3), 610–625 (2014). https://doi.org/10.1007/s00453-012-9683-x
11. Bender, M.A., Farach-Colton, M.: The LCA Problem Revisited. In: Gonnet, G.H., Panario, D., Viola, A. (eds.) LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 1014, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1776, pp. 88–94. Springer, Berlin (2000)
12. Cole, R., Hariharan, R.: Dynamic LCA queries on trees. SIAM J Comput. **34**(4), 894–923 (2005). https://doi.org/10.1137/S0097539700370539
13. Finocchi, I., Grandoni, F., Italiano, G.: Resilient dictionaries. ACM Trans. Algorithms **12**, 6 (2009). https://doi.org/10.1145/1644015.1644016
14. Finocchi, I., Grandoni, F., Italiano, G.F.: Optimal resilient sorting and searching in the presence of memory faults. Theor. Comput. Sci. **410**(44), 4457–4470 (2009). https://doi.org/10.1016/j.tcs.2009.07.026
15. Berkman, O., Vishkin, U.: Finding level-ancestors in trees. J. Comput. Syst. Sci. **48**(2), 214–230 (1994). https://doi.org/10.1016/S0022-0000(05)80002-9
16. Bender, M., Farach-Colton, M.: The level ancestor problem simplified. Theor. Comput. Sci. **01**(321), 5–12 (2004). https://doi.org/10.1016/j.tcs.2003.05.002
17. Beame, P., Fich, F.E.: Optimal bounds for the predecessor problem and related problems. J. Comput. Syst. Sci. **65**(1), 38–72 (2002). https://doi.org/10.1006/jcss.2002.1822
18. Brodal, G.S., Davoodi, P., Srinivasa, Rao S.: Path minima queries in dynamic weighted trees. In: Dehne, F., Iacono, J., Sack, J.R. (eds.) Algorithms and Data Structures, pp. 290–301. Springer Berlin Heidelberg, Berlin (2011)
19. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: On finding lowest common ancestors in trees. SIAM J. Comput. **5**(1), 115–132 (1976). https://doi.org/10.1137/0205011
20. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM J Comput. **13**(2), 338–355 (1984). https://doi.org/10.1137/0213024
21. Pelc, A.: Searching games with errors—fifty years of coping with liars. Theor. Comput. Sci. **270**(1–2), 71–109 (2002). https://doi.org/10.1016/S0304-3975(01)00303-6
22. Geissmann, B., Leucci, S., Liu, C., Penna, P., Proietti, G.: Dual-mode greedy algorithms can save energy. In: Lu, P., Zhang, G. (eds.) 30th International Symposium on Algorithms and Computation, ISAAC 2019, December 8-11, 2019, Shanghai University of Finance and Economics, Shanghai, China. LIPIcs, vol. 149, p. 64:1-64:18. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, Wadern (2019)
23. Geissmann, B., Leucci, S., Liu, C., Penna, P.: Optimal Sorting with Persistent Comparison Errors. In: Bender, M.A., Svensson, O., Herman, G. (eds.) 27th Annual European Symposium on Algorithms, ESA 2019, September 9-11, 2019, Munich/Garching, Germany. LIPIcs, vol. 144, p. 49:1-49:14. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, Wadern (2019)
24. Feige, U., Raghavan, P., Peleg, D., Upfal, E.: Computing with noisy information. SIAM J. Comput. **23**(5), 1001–1018 (1994). https://doi.org/10.1137/S0097539791195877
25. Cicalese, F.: Fault-Tolerant Search Algorithms—Reliable Computation with Unreliable Information. Monographs in Theoretical Computer Science. An EATCS Series. Springer, Berlin (2013)
26. Chen, X., Gopi, S., Mao, J., Schneider, J.: Competitive analysis of the top-K ranking problem. In: Klein, P.N. (ed.) Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19, pp. 1245–1264. SIAM, Philadelphia (2017)
27. Dereniowski, D., Łukasiewicz, A., Uznański, P.: An Efficient Noisy Binary Search in Graphs via Median Approximation. In: Flocchini, P., Moura, L. (eds.) Combinatorial Algorithms, pp. 265–281. Springer, Cham (2021)
28. Boyer, R.S., Moore, J.S.: MJRTY: a fast majority vote algorithm. In: Boyer, R.S. (ed.) Automated Reasoning: Essays in Honor of Woody Bledsoe. Automated Reasoning Series, pp. 105–118. Kluwer, Dordrecht (1991)