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

Luciano Gualà
Department of Enterprise Engineering, University of Rome "Tor Vergata", Italy

Stefano Leucci
Dept. of Information Engineering, Computer Science, and Mathematics, University of L'Aquila, Italy

Isabella Ziccardi
Dept. of Information Engineering, Computer Science, and Mathematics, University of L'Aquila, Italy

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.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases level ancestor queries, lowest common ancestor queries, bottleneck vertex queries, resilient data structures, faulty-RAM model, dynamic trees

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2021.66

Related Version *Full Version*: `https://arxiv.org/abs/2109.09588`

Funding *Luciano Gualà*: This work was partially supported by the project E89C20000620005 "ALgorithmic aspects of BLOckchain TECHnology" (ALBLOTECH).

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-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 [8, 17, 19, 23, 24, 25, 27], 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 consider 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 mainly concerned on the following well-known query types:

[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 [14] by considering *dynamic* trees that only support insertion of leaves.

32nd International Symposium on Algorithms and Computation (ISAAC 2021).
Editors: Hee-Kap Ahn and Kunihiko Sadakane; Article No. 66; pp. 66:1–66:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

(Weighted) Level Ancestor Queries: Given a vertex v and an integer k, the query $\texttt{LA}(v, k)$ 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.

Lowest Common Ancestor Queries: Given two vertices u, v, the query $\texttt{LCA}(u, v)$ returns the vertex at maximum depth in T that is simultaneously and 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 $\texttt{BVQ}(u, v)$ query reports the minimum/maximum-weight vertex in the path between u and v in T.[2] It is worth noticing that, when T is a path, the above problem can be seen as a dynamic version of the classical *range minimum query* problem. In the range minimum query problem, a query $\texttt{RMQ}(i, j)$ reports the minimum element between the i-th and the j-th element of a (static) input sequence [4].

For all of the above problems, *linear-size non-resilient* data structures supporting both the `AddLeaf` and the query operations in constant worst-case time are known [2, 11]. 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 [19]. 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. 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*.

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 [8, 16, 17, 18, 19, 23, 24, 27]. 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 [16]. The solution adopted in the literature is that of suitable 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 [16] for a discussion).[3]

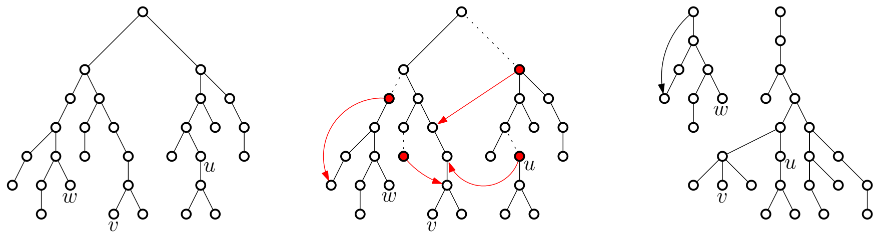
1.1 Our results

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.

---

[2] It is easy to see that this also captures the well-known *bottleneck edge query* variant [12], in which weights are placed on edges instead of vertices.

[3] For example, the authors of [16] consider the problem of designing linear-size resilient dictionaries adopt a notion of *resilient search* that requires the search procedure to answer correctly w.r.t. all uncorrupted keys (see Section 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 tree path leading to the sought key.



(a) (b) (c)

Figure 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 exampple, 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 for every value of k.

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 $\texttt{LA}(v, k)$ 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 Figure 1 for an example.

Our data structure occupies linear (w.r.t. the current number of nodes) space, and supports the `AddLeaf` operation and `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 $\delta = O(\text{polylog} n)$.

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 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 an useful tool to design other resilient data structures involving dynamic trees.

We leave the problem of understanding whether, similarly to other resilient data structures [16, 24], 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

Non-resilient data structures. Before discussing the known result 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 [5] and in [14]. Both papers consider the case in which the tree T is *static* and show how to built, in linear-time, a data structure that requires linear space and that answers queries in constant worst-case time (albeit the hidden constant in [5] is quite large). A simple and elegant construction achieving the same (optimal) asymptotic bounds is given in [3]. In [14], 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 [2], 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. Moreover, the solution of [2] also provides amortized bounds for the problem of maintaining a forest of n nodes under *link* operations (i.e., edge additions that connect two vertices in different trees of the forest) and LA queries. 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), [7] 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 [1]. In [22], Harel and Tarjan show how to preprocess in linear time any *static* tree in order to 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) LA queries, in constant worst-case time per operation [11].

Resilient data structures. As already mentioned, the Faulty-RAM model has been introduced in [19] and used in the context of resilient data structures in [17] 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 [8] 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 [24], 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 [23] 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 [25].

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 [9, 10, 13, 15, 20, 21, 26].

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 Section 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 (Section 3), and we then tackle the dynamic version of the problem (Section 4) for which we give our main result. Due to space limitation, the description of how to modify our data structure to handle the other types of queries is omitted and can be found in the full version of the paper.

2 Preliminaries

Notation. Let T be a rooted tree. For each node $v \in T$, 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).

Faulty memory model. We now formally describe the *Faulty-RAM* model introduced by Finocchi and Italiano in [19]. In this model the memory is divided in two regions: a *safe region* with $O(1)$ memory locations, 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 [16] (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 multiplicative $\Theta(\delta)$ blow-up in its time and space complexities. Essentially, such a solution implements a trivial error-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 [6]). 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 will 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. Notice that this assumption can be removed by carefully using the replication strategy described in Section 2.

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

if $k \leq 2\delta$ **then return** $\texttt{climb}(v, k)$;
Climb up the tree T from v for 2δ nodes searching a black node;
if the previous procedure did not find a black node **then return** error;
$v' \leftarrow$ a black node found in the previous procedure;
$d \leftarrow$ distance between v' and v; $k' \leftarrow k - d$;
$u' \leftarrow \texttt{LA}_Q(q_{v'}, \lfloor k'/\delta \rfloor)$;
$k_{\text{rest}} \leftarrow k' - \lfloor k'/\delta \rfloor \cdot \delta$;
return $\texttt{climb}(u', k_{\text{rest}})$;

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 v node 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 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 Figure 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 $\texttt{LA}$ 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 $\texttt{LA}$ queries on static trees in constant time and requires linear space (see, e.g., the data structure in [3]).

As we observed before, any data structure can be made resilient with a multiplicative $\Theta(\delta)$ blow-up in its time and space complexities. 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 $\texttt{LA}$ 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 v in Q.

The resilient level-ancestor query. In this section we show how to implement our resilient $\texttt{LA}$ 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

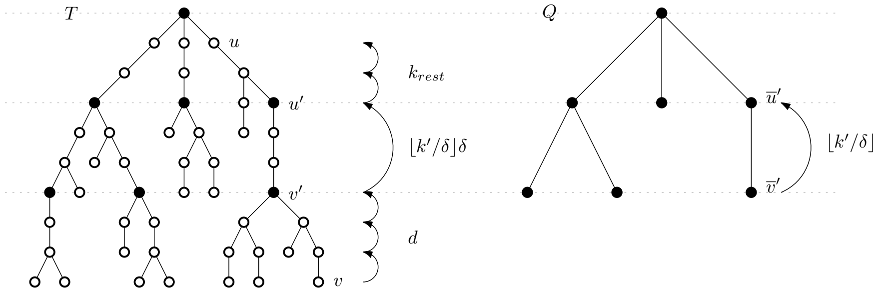


Figure 2 Left: A static tree T that has been colored according to the scheme in Section 3 for $\delta = 3$. Right: the corresponding black tree Q. We also show the path climbed while answering the query $\texttt{LA}(v,k)$ with $k = 8$. In this case $d = 3$, $\lfloor k'/\delta \rfloor = 1$ and $k_{\text{rest}}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

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 require $\Omega(n)$ 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 simply by halting the whole query procedure and reporting an error. A slightly more sophisticated handling of invalid pointers will be used to tackle the dynamic case. An example `LA` query is given in Figure 2.

The correctness of the above algorithm immediately follows from the fact that, when no vertex between v and the k-th ancestor 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.[4]

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 the full version of the paper, we show how our ideas can be extended to also handle weighted `LA`, `BVQ`, and `LCA` queries.

[4] 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 $\delta - 2$.

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 static case. Namely, the nodes of T are colored with either *black* or *white*, the set of *black* nodes will have size $O(n/\delta)$, and it will correspond 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 adversary's corruptions might cause vertices to become miscolored. We will however ensure that such a regular coloring pattern will be followed by the portions of T that are sufficiently distant from the adversary's corruptions. This will allow 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 these flags to "pay" for the creation of new black nodes. Spent flags will also signal the presence of a nearby black ancestor.

For technical reasons, we also allow an unspent flag flag_v 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 Section 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.[5]

The pointer p_v is then the index (in $\{0, \dots, n-1\}$) 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 find convenient to use the above fact to simplify the handling of corrupted vertices: whenever we encounter an invalid pointer $p_v \geq v$ we treat it as being equal to 0, i.e., an invalid pointer p_v always refers to the root r of T. This rules also applies to any read pointer, including those accessed by the $\texttt{climb}(\cdot, \cdot)$ procedure already defined in Section 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, if no corruptions occur T and $\mathcal{T}$ coincide.

---

[5] 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.

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

$\mathtt{NewTree}_Q(v)$: 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 $\overline{v}$.

$\mathtt{AddLeaf}_Q(\overline{u}, v)$: 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 children 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 [2]. 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 6).

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 $\mathtt{AddLeaf}(x_{par})$ 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 $\mathtt{AddLeaf}(x_{par})$, 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 exploring the current tree by climbing up to $3\delta - 1$ levels of $\mathcal{T}$ from x while gathering information for the second phase. In order to do so, Algorithm 2 climbs δ levels of $\mathcal{T}$ from the newly inserted node x, reaching a vertex y, and checks 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 $\mathtt{AddLeaf}(x_{par})$ 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 neither black-free nor near-a-black, we return from the $\mathtt{AddLeaf}(x_{par})$ procedure (without performing the execution phase), otherwise we move on to 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 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 them to $\top$). The position of the new black vertex depends on whether y was

Algorithm 2 `AddLeaf`(x_{par}).

Add a new record x at the end of $\mathcal{A}$; $p_x \leftarrow x_{par}$; $\text{flag}_x \leftarrow \bot$; $q_x \leftarrow$ `null`;
// Discovery Phase
$y \leftarrow x$; // Check the flags of the lowest δ proper ancestors of x
for $i = 1, \ldots, \delta$ **do**
    if $y = r$ **then return** x;
    $y \leftarrow p_y$;
    if $\text{flag}_y = \top$ **then return** x; // Return immediately if a spent flag is found
    $\text{flag}_y \leftarrow (x, i)$; // Annotate flag_y
// Check whether y is near-a-black.
// ℓ will be the distance to the closest proper black ancestor y' of y, if any
$y' \leftarrow y$; $\ell \leftarrow 0$; near_black $\leftarrow$ `false`;
while $\ell < \delta$ **and** $y' \neq r$ **and** near_black= `false` **do**
    $y' \leftarrow p_{y'}$; $\ell \leftarrow \ell + 1$;
    if y' is black **then** near_black $\leftarrow$ `true`;
// If y is not near-a-black, check whether it is black-free
if near_black $=$ `false` **then**
    $z \leftarrow y'$;
    for $\delta - 1$ times **do**
        if $z = r$ **then break**;
        $z \leftarrow p_z$;
        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
$z \leftarrow x$;
for $i = 1, \ldots, \delta$ **do**
    if $z = r$ **then return** x;
    $z \leftarrow p_z$;
    if $\text{flag}_z \neq (x, i)$ **then return** x; // Check the annotation of flag_z
    if near_black $=$ `true` **and** $i = \ell$ **then** $x' \leftarrow z$; // y' is the δ-parent of x'
    $\text{flag}_z \leftarrow \top$; // Spend flag_z
if near_black $=$ `true` **then**
    $q_{x'} \leftarrow$ `AddLeaf`$_Q(q_{y'}, x')$;
else
    $q_y \leftarrow$ `NewTree`$_Q(y)$;
return x;

near-a-black or black-free. In the former case the vertex y' discovered in the first phase will be the δ-parent of the new black vertex x', and a new leaf $\overline{x}'$ is appended to $\overline{y}'$ in Q. In the latter case, y will become 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

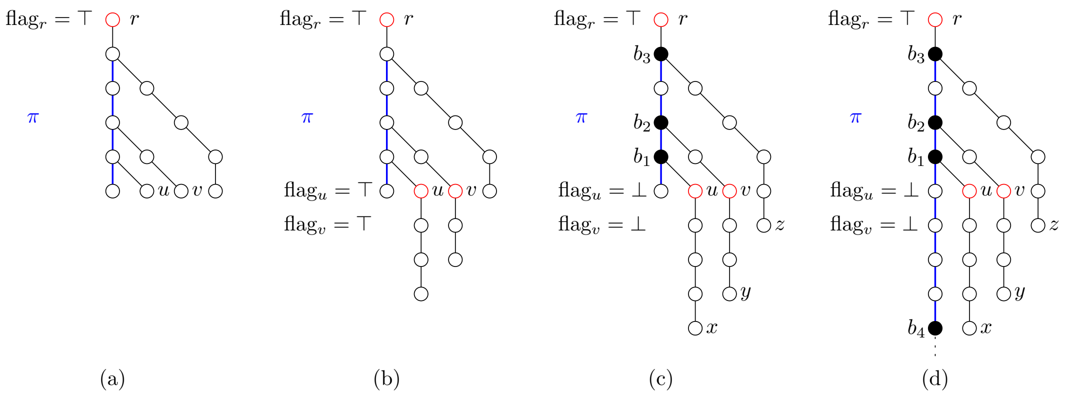


Figure 3 An example showing that an uncorrupted path π (depicted in blue) can exhibit an irregular pattern of black vertices (d). Situation (a) can be reached when the adversary corrupts r by setting $\text{flag}_r = \top$ before the insertions of the other nodes take place. To obtain (b), the adversary 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 $\perp$ 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).

is stated in Lemma 5. We first need to prove auxiliary properties. In Figure 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 Figure 4 (a).

▶ **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 18–24.

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 $P[y : w] = P'[y : w]$. Hence, y is at distance at most $\delta - 1$ from w in P (and in T) showing that z is a proper ancestor of y. Therefore all nodes in $P'[y : w]$ are uncorrupted, and the loop in in lines 18–24 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 x and its closest black proper ancestor. If $\ell \geq i$, then Algorithm 2 colors a vertex in $P(z : w] = \pi(z : w]$ black. Otherwise, if $\ell < i$, the discovery phase observed that the ℓ-parent y' of y was black. Since $\ell < \delta$, y' lies in $\pi(z : y]$. ◀

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

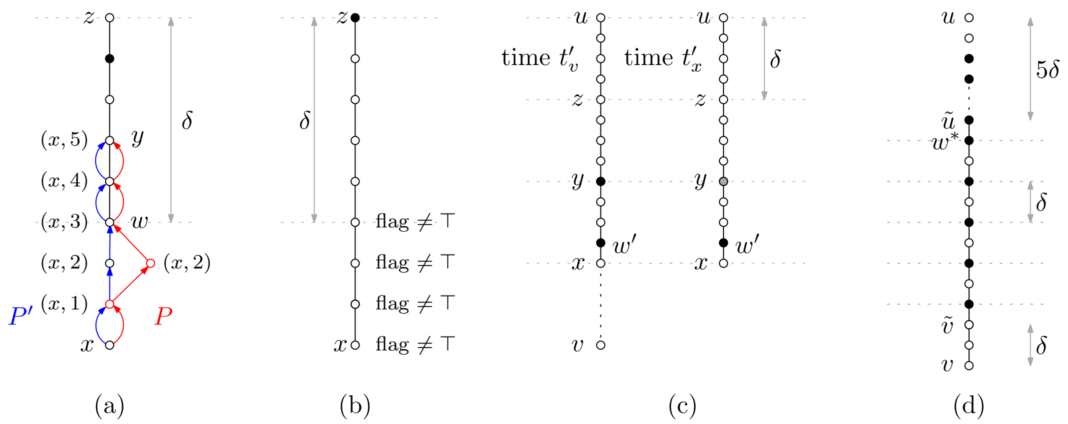


Figure 4 (a) Graphical representation of the proof of Lemma 1, for $\delta = 5$. (b), (c), (d): Representations of the statements of Lemma 3, Lemma 4, and Lemma 5, respectively.

▶ **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 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. ◀

Recall that we would like each uncorrupted path to contain a black vertex every δ levels. Consider an uncorrupted path π of length between δ and 2δ with a single black vertex z on top. Then, the vertex at distance δ from z is "overdue" to become black. Next lemma shows that all flags associated with descendants of the overdue vertex in π must 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 Figure 4 (b).

▶ **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 $\text{flag}_w \neq \top$.*

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 timing at which vertices of a long uncorrupted path become black. This will be instrumental to prove Lemma 5. See Figure 4 (c).

▶ **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). 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 Figure 4 (d).

▶ **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 8 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 $w^* = b$. ◀

Algorithm 3 Answers a level ancestor query LA(v, k) in dynamic trees.

if $k \leq 7\delta$ **then**
  return climb(v, k);
$\tilde{v} \leftarrow$ climb(v, δ);
Climb up the tree $\mathcal{T}$ from $\tilde{v}$ for up to δ nodes searching a black node;
if the previous procedure did not find a black node **then**
  return error;
$v' \leftarrow$ a black node found in the previous procedure;
$d \leftarrow$ distance between v' and v;
$k' \leftarrow k - d - 5\delta$;
$u' \leftarrow \mathtt{LA}_Q(q_{v'}, \lfloor k'/\delta \rfloor)$;
$k_{\text{rest}} \leftarrow k' - \lfloor k'/\delta \rfloor \cdot \delta + 5\delta$;
return climb(u', k_{rest}) ;

The above lemma 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 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 6.** *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 vector $\mathcal{A}$ 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 that requires linear space, takes constant time to answer each `LA` query and to perform each `AddLeaf` operation [2]. 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 $(n + \delta)/\delta = O(n/\delta)$. ◀

References

1. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. On finding lowest common ancestors in trees. *SIAM J. Comput.*, 5(1):115–132, 1976. doi:10.1137/0205011.
2. Stephen Alstrup and Jacob Holm. Improved algorithms for finding level ancestors in dynamic trees. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, ICALP '00, pages 73–84, Berlin, Heidelberg, 2000. Springer-Verlag.
3. Michael Bender and Martin Farach-Colton. The level ancestor problem simplified. *Theor. Comput. Sci.*, 321:5–12, January 2004.

4 Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In Gaston H. Gonnet, Daniel Panario, and Alfredo Viola, editors, *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000. doi:10.1007/10719839_9.

5 Omer Berkman and Uzi Vishkin. Finding level-ancestors in trees. *J. Comput. Syst. Sci.*, 48(2):214–230, April 1994. doi:10.1016/S0022-0000(05)80002-9.

6 Robert S. Boyer and J. Strother Moore. MJRTY: A fast majority vote algorithm. In Robert S. Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe*, Automated Reasoning Series, pages 105–118. Kluwer Academic Publishers, 1991.

7 Gerth Stølting Brodal, Pooya Davoodi, and S. Srinivasa Rao. Path minima queries in dynamic weighted trees. In Frank Dehne, John Iacono, and Jörg-Rüdiger Sack, editors, *Algorithms and Data Structures*, pages 290–301, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

8 Gerth Stølting Brodal, Rolf Fagerberg, Irene Finocchi, Fabrizio Grandoni, Giuseppe F. Italiano, Allan Grønlund Jørgensen, Gabriel Moruz, and Thomas Mølhave. Optimal resilient dynamic dictionaries. In Lars Arge, Michael Hoffmann, and Emo Welzl, editors, *Algorithms – ESA 2007*, pages 347–358, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

9 Xi Chen, Sivakanth Gopi, Jieming Mao, and Jon Schneider. Competitive analysis of the top-K ranking problem. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 1245–1264. SIAM, 2017. doi:10.1137/1.9781611974782.81.

10 Ferdinando Cicalese. *Fault-Tolerant Search Algorithms – Reliable Computation with Unreliable Information.* Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2013. doi:10.1007/978-3-642-17327-1.

11 Richard Cole and Ramesh Hariharan. Dynamic LCA queries on trees. *SIAM J. Comput.*, 34(4):894–923, 2005. doi:10.1137/S0097539700370539.

12 Erik D. Demaine, Gad M. Landau, and Oren Weimann. On cartesian trees and range minimum queries. *Algorithmica*, 68(3):610–625, 2014. doi:10.1007/s00453-012-9683-x.

13 Dariusz Dereniowski, Aleksander Łukasiewicz, and Przemysław Uznański. An efficient noisy binary search in graphs via median approximation. In Paola Flocchini and Lucia Moura, editors, *Combinatorial Algorithms*, pages 265–281, Cham, 2021. Springer International Publishing.

14 Paul F. Dietz. Finding level-ancestors in dynamic trees. In Frank Dehne, Jörg-Rüdiger Sack, and Nicola Santoro, editors, *Algorithms and Data Structures*, pages 32–40, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

15 Uriel Feige, Prabhakar Raghavan, David Peleg, and Eli Upfal. Computing with noisy information. *SIAM J. Comput.*, 23(5):1001–1018, 1994. doi:10.1137/S0097539791195877.

16 Irene Finocchi, Fabrizio Grandoni, and Giuseppe Italiano. Resilient dictionaries. *ACM Transactions on Algorithms*, 6, December 2009. doi:10.1145/1644015.1644016.

17 Irene Finocchi, Fabrizio Grandoni, and Giuseppe F. Italiano. Resilient search trees. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '07, pages 547–553, USA, 2007. Society for Industrial and Applied Mathematics.

18 Irene Finocchi, Fabrizio Grandoni, and Giuseppe F. Italiano. Optimal resilient sorting and searching in the presence of memory faults. *Theor. Comput. Sci.*, 410(44):4457–4470, 2009. doi:10.1016/j.tcs.2009.07.026.

19 Irene Finocchi and Giuseppe F. Italiano. 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, pages 101–110, New York, NY, USA, 2004. Association for Computing Machinery. doi:10.1145/1007352.1007375.

20 Barbara Geissmann, Stefano Leucci, Chih-Hung Liu, and Paolo Penna. Optimal sorting with persistent comparison errors. In Michael A. Bender, Ola Svensson, and Grzegorz Herman, editors, *27th Annual European Symposium on Algorithms, ESA 2019, September 9-11, 2019, Munich/Garching, Germany*, volume 144 of *LIPIcs*, pages 49:1–49:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ESA.2019.49.

21 Barbara Geissmann, Stefano Leucci, Chih-Hung Liu, Paolo Penna, and Guido Proietti. Dual-mode greedy algorithms can save energy. In Pinyan Lu and Guochuan Zhang, editors, *30th International Symposium on Algorithms and Computation, ISAAC 2019, December 8-11, 2019, Shanghai University of Finance and Economics, Shanghai, China*, volume 149 of *LIPIcs*, pages 64:1–64:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. `doi:10.4230/LIPIcs.ISAAC.2019.64`.

22 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984. `doi:10.1137/0213024`.

23 Giuseppe F. Italiano. Resilient algorithms and data structures. In Tiziana Calamoneri and Josep Díaz, editors, *Algorithms and Complexity, 7th International Conference, CIAC 2010, Rome, Italy, May 26-28, 2010. Proceedings*, volume 6078 of *Lecture Notes in Computer Science*, pages 13–24. Springer, 2010. `doi:10.1007/978-3-642-13073-1_3`.

24 Allan Grønlund Jørgensen, Gabriel Moruz, and Thomas Mølhave. Priority queues resilient to memory faults. In Frank K. H. A. Dehne, Jörg-Rüdiger Sack, and Norbert Zeh, editors, *Proceedings of the 10th International Workshop on Algorithms and Data Structures (WADS'07)*, volume 4619 of *Lecture Notes in Computer Science*, pages 127–138. Springer, 2007. `doi:10.1007/978-3-540-73951-7_12`.

25 Stefano Leucci, Chih-Hung Liu, and Simon Meierhans. Resilient dictionaries for randomly unreliable memory. In Michael A. Bender, Ola Svensson, and Grzegorz Herman, editors, *27th Annual European Symposium on Algorithms, ESA 2019, September 9-11, 2019, Munich/Garching, Germany*, volume 144 of *LIPIcs*, pages 70:1–70:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. `doi:10.4230/LIPIcs.ESA.2019.70`.

26 Andrzej Pelc. Searching games with errors – fifty years of coping with liars. *Theor. Comput. Sci.*, 270(1-2):71–109, 2002. `doi:10.1016/S0304-3975(01)00303-6`.

27 Umberto Ferraro Petrillo, Fabrizio Grandoni, and Giuseppe F. Italiano. Data structures resilient to memory faults: An experimental study of dictionaries. *ACM J. Exp. Algorithmics*, 18, 2013. `doi:10.1145/2444016.2444022`.