





Space-Efficient Functional Offline-Partially-Persistent Trees with Applications to Planar Point Location^{*}

Gerth Stølting Brodal , Casper Moldrup Rysgaard ,
Jens Kristian Refsgaard Schou , and Rolf Svenning 

Department of Computer Science, Aarhus University, Denmark
{gerth,rysgaard,jkrs,rolfsvenning}@cs.au.dk

Abstract. In 1989 Driscoll, Sarnak, Sleator, and Tarjan presented general space-efficient transformations for making ephemeral data structures persistent. The main contribution of this paper is to adapt this transformation to the functional model. We present a general transformation of an ephemeral, linked data structure into an offline, partially persistent, purely functional data structure with additive $\mathcal{O}(n \log n)$ construction time and $\mathcal{O}(n)$ space overhead; with n denoting the number of ephemeral updates. An application of our transformation allows the elegant slab-based algorithm for planar point location by Sarnak and Tarjan 1986 to be implemented space efficiently in the functional model using linear space.

Keywords: Data structures · Functional · Persistence · Point location

1 Introduction

The functional model has many well-known advantages such as modulation, shared resources, no side effects, and easier formal verification [4,14]. These advantages are given by restricting the model to only use functions and immutable data. As all data are immutable, side effects in functions are not possible, allowing modules to work independently of the context they are used in and reducing the complexity of formal verification [14].

In 1999 Okasaki [19] gave a seminal work on techniques for designing efficient (purely) functional data structures, and our result follows this line of research. Adapting existing data structures to the functional model is non-trivial since modifications are prohibited. However, it also means that updates do not destroy earlier versions of the data structure making functional data structures inherently persistent, but not necessarily space efficient. The focus of this paper is to adapt existing imperative techniques for persistence to the functional model in a space-efficient manner.

We introduce a purely functional framework that adapts classical tree structures to support offline partial persistence with an additive overhead. By offline

^{*} Work supported by Independent Research Fund Denmark, grant 9131-00113B.

Table 1: Previous and new results for planar point location, where \dagger are expected bounds and $*$ are results based on persistent data structures.

Reference	Construction	Query	Space	Model
David Kirkpatrick [17]	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	Imperative
Seidel [26]	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	Imperative \dagger
Dobkin and Munro [11]	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log^2 n)$	$\mathcal{O}(n \log n)$	Imperative*
Richard Cole [8]	$\mathcal{O}(n^2)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	Imperative*
Sarnak and Tarjan [24]	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	Imperative*
Sarnak and Tarjan [24]	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n \log n)$	Functional*
<i>New</i>	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	Functional*

partial persistence, we mean that all updates are made before queries. This restriction allows us to store update information without immediately being able to handle queries efficiently. A data structure that is not persistent, i.e., does not support queries and updates in previous versions, is said to be *ephemeral*. We show that by recording when an ephemeral tree structure is updated, we can build a query structure that can efficiently answer queries to previous versions of the structure. In the imperative paradigm, it is possible to efficiently interweave updates and queries [12,24], but in the functional paradigm, it incurs a multiplicative logarithmic space overhead, which we show how to circumvent, in the offline setting.

Planar point location is a classic computational geometry problem [13,17,24]. Given a planar straight-line graph with n edges (interchangeably line segments) and report the region containing a query point q . The task is to create a data structure that supports these queries while minimizing the construction time, query time, and space of the data structure. Sarnak and Tarjan [24] showed that the planar point location problem can be solved elegantly using *partially-persistent* sorted sets, that in the functional setting can be solved with balanced search trees using path copying resulting in a space usage of $\mathcal{O}(n \log n)$. As an application of our technique, we show how the algorithm of Sarnak and Tarjan can be implemented in the functional model to only use space $\mathcal{O}(n)$. For an overview of results for the planer point location problem see Table 1.

Below we state sufficient conditions for a functional or imperative data structure to be augmented with a functional support structure that supports offline partial persistence queries.

Definition 1 (TUNA conditions).

- T*: The data structure forms a rooted tree of constant degree d .
- U*: Updates create $\mathcal{O}(1)$ new edges and nodes.
- N*: No cycles are created by updates when considering the edges that have been created across all versions.
- A*: Attribute values of nodes are static, i.e., the information stored in a node is not changed after its insertion. This does not include fields pointing to the children.

This definition is not too restrictive, we show that binary search trees (BST), Treaps [3], Red/Black Trees [5], and Functional Random Access Arrays [18] all can be modified to satisfy Definition 1, without asymptotically significant overhead. In Section 5 we discuss how some of these requirements can be relaxed or generalized further.

Theorem 1. *For any ephemeral, linked data structure that satisfies the TUNA conditions, an equivalent, functional, offline-partially-persistent data structure preserving the asymptotic update and query times, can be created, with an additive construction overhead of time $\mathcal{O}(n \log n)$ and space $\mathcal{O}(n)$ for a series of n updates.*

The main idea behind Theorem 1 is to store the update information in a list, including edge insertion and deletion timestamps. After all the updates have been applied, a bottom-up topological sort produces a directed acyclic graph (DAG) that supports queries to any previous version of the data structure. This is essentially implementing the node copying approach of [24], while carefully avoiding the creation of cycles.

Building upon [24], Theorem 1 immediately implies a state-of-the-art functional planar point location solution, summarised in the following corollary.

Corollary 1. *There exists a purely functional solution to the planar point location problem with construction time $\mathcal{O}(n \log n)$, query time $\mathcal{O}(\log n)$, and space $\mathcal{O}(n)$.*

We implemented unbalanced binary search trees in the purely functional programming language Haskell and report on some experiments in Section 6.

1.1 Persistence

A data structure is said to be *persistent* if it is possible to query previous versions of it and *ephemeral* if it is only the current version that is available. A *partially-persistent* data structure is *persistent* and allows updates only to the latest version. The stronger notion of *full persistence* implies that any version can be both queried and updated. An update to a persistent data structure never changes an existing version, but instead creates a new version derived from the version the update is applied to. In this way, the different versions of the partial persistent structure form a version list, whereas full persistence forms a version tree. General transformations to make data structures persistent were studied by Driscoll et al. [12] and Overmars [21,22]. In this paper, we focus on *offline* partial persistence for linked data structures, where all updates are performed before all queries, which was also explored in [11].

A naive idea to achieve partial persistence is to store a copy of every previous version. If the underlying structure is a list, then this approach generates an overhead of $\Omega(n^2)$ space for n insertions into an initially empty list. To improve upon this, the crucial observation is that when structures have a large overlap between updates, it is possible to reuse large parts of the previous versions and

greatly reduce space usage. To achieve linear space, the notions of *fat nodes* and *timestamps* were introduced by Driscoll et al. [12], where each pointer field in a node is replaced by a list of pointers and each pointer in the list has an associated timestamp, denoting when the pointer was updated. Specifically, for a binary search tree, each node will, instead of containing a pointer to the `left` and `right` subtree, contain a list of timestamped `left` pointers and a list of timestamped `right` pointers. An update to a given pointer now adds a new pointer to the pointer list with the current timestamp, resulting in $\mathcal{O}(1)$ space overhead per update. As the pointer in the version is the last in the list, there is no overhead in finding the active pointer in the current version. To locate the correct pointer at a given older timestamp, a binary search on the list of pointers can be performed, which then imposes a multiplicative $\mathcal{O}(\log n)$ overhead on queries.

In [24] the notion of *path copying* was introduced for BSTs, where all nodes on the path to the node being updated are copied. Any existing pointer along the path can then point freely to parts of the old structure. For BSTs, this has a space overhead of the length of the path, i.e., for balanced BSTs an $\mathcal{O}(\log n)$ space overhead per update. It does however impose no overhead on the query time, apart from initially finding the correct root to query.

Thus, the fat node technique has a query time overhead, whereas path copying has a space overhead. By combining these two techniques, it is possible to have no overhead on query time and space. In [12,24] the authors achieve this by introducing the *node copying* technique for partially-persistent general pointer-based data structures. Here nodes are allowed to hold a constant number of additional time-stamped pointers. When some operation requires a node to update a pointer, the timestamp of the old pointer is updated to end at the current time. We denote a pointer that has ended as *expired*. If the pointer is replaced by a new pointer, then the new pointer is placed in one of the free extra pointer slots of the node, with a timestamp starting at the current time. If there is no free pointer slot *node copying* is performed, where the non-expired pointers and the new pointer are copied to a new node. Any pointer in another node to the copied node at the current time must be split, which may cause node copying to cascade up the structure. However, an amortization argument [24] shows that this technique only has an additive $\mathcal{O}(n)$ overhead in space, when the indegree of every node in the underlying structure is constant. Finally, as the number of pointers in each node is constant the overhead on the query time is also constant.

1.2 Persistence and the functional model

The functional programming paradigm is well suited for persistence as stated by Okasaki: “*A distinctive property of functional data structures is that they are always persistent*” [19]. In purely functional programming, there are no side effects and variables are immutable, meaning that any modifications to a structure S are obtained by creating a new structure S' without altering S . In this new structure S' , substructures may be references to (immutable) old substructures from S .

Purely functional data structures are consequently particularly interesting when persistence is critical, and it is natural that they are less efficient than imperative data structures, since they inherently solve a more general problem. We note that there has been some work in describing to what degree mutability increases the capabilities (efficiency) of a language, and Pippenger [23] gave an example of a problem with a logarithmic-factor separation under certain conditions. However, for many data structure problems, purely functional solutions have been developed that match their imperative counterparts with only constant overhead. Examples include optimal *confluently* persistent deques which were developed over a number of papers [7,9,16] and optimal priority queues [6].

1.3 Functional vs. space efficient imperative persistence

The fat node and node copying persistence techniques mentioned in Section 1.1 rely upon the imperative paradigm’s ability to modify pointers in the nodes in a graph structure where nodes can have multiple ingoing edges. As the functional model cannot mutate pointers, directly translating these solutions leads to significant overhead in the update time, as, even with path copying, all ancestors of an updated node must be remade to point to the newly created node(s). For this reason, we focus our attention on data structures with a tree structure where ancestors appear on a single path to the root.

1.4 Planar point location

Dobkin and Lipton [10] solved the planar point location problem by drawing vertical lines through every node resulting in vertical slabs. For every slab, the line segments spanning the slab are stored in a BST. This method allows efficient queries by performing a binary search horizontally for the slab, and then a binary search vertically in the slab for the region. This gives an overall query time of $\mathcal{O}(\log n)$. The drawback of this method is that each line could potentially be stored in almost every slab, resulting in $\Theta(n^2)$ space. A number of different results [8,13,17,24], show that the space can be reduced to $\mathcal{O}(n)$ (non-functional) without affecting the query time. The solution by Cole [8] is particularly interesting as it exploits that neighboring slabs are very similar, meaning that the problem can be reduced to creating persistent, sorted sets. This observation is vital to the work on persistent search trees by Sarnak and Tarjan [24]. On the other hand, the approach by Kirkpatrick [17] is completely different and is based on repeatedly triangulating the graph and removing a constant fraction of the nodes with degree at most 11. Then a DAG is created bottom-up based on the overlap between two consecutive triangulations. Since the DAG is created bottom-up, similarly to our approach described in Section 3.2, we conjecture that this approach could be adapted to the purely functional model and leave it as an open problem.

2 Initial analysis of binary search trees

In this section, we introduce ephemeral (i.e., non-persistent) unbalanced binary search trees (BSTs) and illustrate how the fat node technique can be adapted to the functional paradigm. The main obstacle in making the adaptation is to avoid cycles between nodes. We solve this by making new copies of nodes that should be moved above their parent. In Section 5 these techniques are extended to cover balanced search trees.

2.1 Ephemeral binary search trees

Any BST node is either an empty leaf or a node containing an element and two pointers to a left and right sub-tree, which are in turn also BSTs. Forthwith, these pointers will be denoted as *edges*. Likewise, let T and T' denote BSTs over a totally ordered set of elements X and let $x \in X$ denote an element. BSTs are ordered such that all elements in the left subtree are smaller than the element in the node, and all elements in the right subtree are larger. BSTs support many operations; we focus on the following three basic operations:

- **Insert**(T, x): Insert x into T and return the resulting tree T' .
- **Delete**(T, x): Delete x from T and return the resulting tree T' .
- **Search**(T, x): Return the smallest element x' in T , such that $x \leq x'$.

All operations can be implemented in time linear in the height of the tree. We call operations that modify the data structure *updates* (**Insert** and **Delete**). We call operations that query the data structure without changing it *queries* (**Search**). When constructing the data structure, we consider a sequence of n updates (u_1, \dots, u_n) and for version $0 \leq t \leq n$ updates u_1, \dots, u_t have been applied.

2.2 Fat node binary search trees

In this subsection, we describe how to adapt imperative unbalanced BSTs to adhere to the TUNA conditions (Definition 1), using fat nodes. Most importantly, the **Delete** update is changed slightly from the classic behavior since it can cause nodes to be reordered in the tree.

When performing **Insert**, assuming that x is not present in T , a path to the correct leaf position is found, and a new node, containing x is created at the position of that leaf. The difference between the old and the new tree is a single edge at the bottom of the tree. By adding *creation timestamps* on the edges to represent creation time, queries can detect if a particular edge should be considered to exist in a specific version or not.

When performing **Delete**, if the deleted element x is at the bottom of the tree, then, in effect, the edge e from its parent p to the deleted node ceases to exist. We record this by adding an *ending timestamp* to the edge. If the deleted element x is in some internal node, see Fig. 1 (Left), then a predecessor or

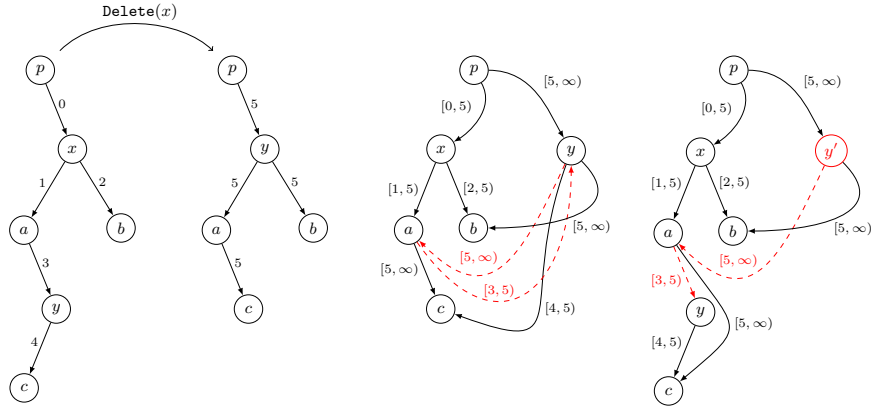


Fig. 1: We let a single number above an edge denote its creation time. In the graphs, the interval above an edge is the living span of the edge. Recall that a persistent search tree query is a timestamp/version and a value. Left: a deletion in a BST. Middle: the resulting graph contains the dashed cycle between y and a . Right: by creating a new copy y' of y the resulting graph remains a DAG and is TUNA compliant.

successor y (depending on implementation) in the subtree rooted at x is found and moved to the deleted spot. This introduces up to four new edges.

When transforming the BST into the functional model, the predecessor cannot be directly moved and reused, as it might be moved above its parent, which would create a cycle in the structure when also considering expired edges, as seen in Fig. 1 (Middle). We avoid creating cycles by replacing x with a copy of its predecessor which breaks the cycle as seen in Fig. 1 (Right). However, it remains a crucial issue that b (and c) still have two parents, when also considering the expired edges, which cannot be updated efficiently in the functional model (see Section 1.3).

3 Freezer and query structure construction

In this section we present our main contribution: the concept of a *Freezer*, which stores update information as it comes in, and how to use it to build a query DAG after all updates have occurred, using fat nodes and node copying. We give an amortized analysis argument that the total number of node copies is linear in the number of updates and from the proof we deduce that the TUNA conditions 1 are sufficient to prove Theorem 1.

3.1 The Freezer

The underlying ephemeral structure forms a tree, but when performing updates using the TUNA-compliant fat node method (see Fig. 3) to get persistence, then

the data structure forms a DAG. This is problematic since in the functional model a DAG cannot be maintained efficiently using path copying, as any node in the data structure which can reach an updated node would also have to be copied to reflect the change. Thus, we store all edges not present in the most recent version of the data structure separately in a list which we call the *freezer*. That is, only the most recent version of the data structure is explicitly maintained. This is sufficient for partial persistence since updates are only allowed in the most recent version. Finally, after all the updates have been applied, the DAG is built bottom-up. For this reason, persistence is restricted to offline.

To store the edges in the freezer, a unique *id* is assigned to each node and each edge in the freezer is stored as a 5-tuple $(id_{\text{from}}, id_{\text{to}}, \text{field}, t_{\text{start}}, t_{\text{end}})$. The ids are used to identify the nodes the edge connects, the field denotes which field of the node with id id_{from} the edge originates from (in the BST this is either **left** or **right**), and the time stamps t_{start} and t_{end} denote the *living span* of the edge as the half-open interval $[t_{\text{start}}, t_{\text{end}})$.

The freezer in addition stores which node is the root at any given time, and a map from ids to the value contained in the node with the given id. Further, note that storing the deletion time of an edge can be omitted and instead be read from the starting time of the next edge in the corresponding field of the same node, with the modification that empty fields have an edge to a special Nil leaf.

3.2 Offline construction of the fat node query DAG

In this section, we describe how to construct the fat node DAG structure from the expired edges in the freezer and the final non-expired structure.

Any edge that ends in the freezer must have been created in some version of the tree as the result of an update operation. An **Insert** operation creates one new edge, and a **Delete** operation creates at most four new edges. Thus, after all n updates, the freezer contains $\mathcal{O}(n)$ edges. Similarly, each **Insert** and **Delete** creates at most one new node, so the number of nodes is $\mathcal{O}(n)$.

Using Kahn's algorithm [15], which topologically sorts the nodes by repeatedly extracting nodes with outdegree zero, we build the DAG bottom-up. The while loop of the imperative algorithm is replaced by a recursive function in the functional algorithm, for each iteration calling with the new values needed. As mentioned, by copying nodes when they were moved around by the updates we avoid introducing cycles (see Figs. 1 and 3), which is required for Kahn's algorithm. By having a map from node ids to the values contained in the nodes, it is possible to explicitly build the DAG over the edges of the freezer. This creates the DAG of fat nodes.

The construction time of, a non-functional implementation of, Kahn's algorithm is linear in the number of nodes and edges. This however relies on being able to effectively fetch the ingoing and outgoing edges of nodes, and reduce the outdegree of nodes in time $\mathcal{O}(1)$. The construction relies on efficient maps from ids to nodes. As random access is not part of the functional model, maps with $\mathcal{O}(1)$ lookup time and update times do not exist. We instead use balanced trees

which introduces an overhead of $\mathcal{O}(\log n)$ for each of the operations, yielding a DAG construction time of $\mathcal{O}(n \log n)$. The space usage remains $\mathcal{O}(n)$.

3.3 Bounding node outdegree of the query DAG

Having arbitrary outdegree of fat nodes affects the query time, as stated in Section 1.1. However, by limiting the number of extra pointers in each node, limited node copying similar to [24] can be implemented to remove the query overhead, while still maintaining linear space. The difference here is that in [24] the copy is performed during the update phase as soon as there are too many pointers in a node. We restrict ourselves to the offline setting and as such do not need to be able to handle queries before all updates have been applied, this means that we can allow nodes to become arbitrarily fat in the update phase.

Let d be the degree of the underlying ephemeral structure. After the update phase, we handle the high degree by recursively splitting the fat nodes into multiple nodes, each with degrees $\mathcal{O}(d)$ by interleaving the node splitting idea with Kahn’s algorithm. As discussed in Section 3.2 we visit the nodes of the freezer bottom-up in topological order. Recall that the freezer contains edges, so nodes are inferred. When visiting a node v , that has an outdegree larger than $d + e$, for a parameter $e = \mathcal{O}(d)$ indicating the extra edges we allow every node to hold, we split it into a left node v_l to represent “the past” and a right node v_r to represent “the future”, essentially performing node copying. Note that some edges will be active in both of these nodes and thus are duplicated, similar to regular node copying. We perform the split such that the outdegree of the left node is at most $d + e$, and that the left and right nodes in total represent the original node. If the outdegree of v_r is larger than $d + e$ we recursively split it until v has been split into a number of nodes each of outdegree at most $d + e$. By requiring $1 \leq e \leq cd$, for some constant c , we ensure that the query time is proportional to that of the underlying ephemeral structure since the new nodes will contain $\Theta(d)$ edges. We call this procedure *node copying* and the following lemma shows that the space remains linear in the number of updates n .

Lemma 1. *The number of nodes introduced by node copying is $\mathcal{O}(n)$ when $1 \leq e \leq cd$ for some constant c .*

Proof. We define the potential function

$$\Phi = \sum_v \max\{0, O_v - (d + e)\},$$

where O_v is the number of outgoing edges from the node v , and the terms in the sum indicate the number of outgoing edges above the threshold $d + e$. Observe that the potential is nonnegative and that $\Phi = \mathcal{O}(n)$ when initializing *node copying* due to property U.

We now analyze how the potential changes when we split a node v , which must have $O_v > d + e$, into a left node v_l and a right node v_r . We consider the edges (consisting of a start and end timestamp) in nondecreasing order by

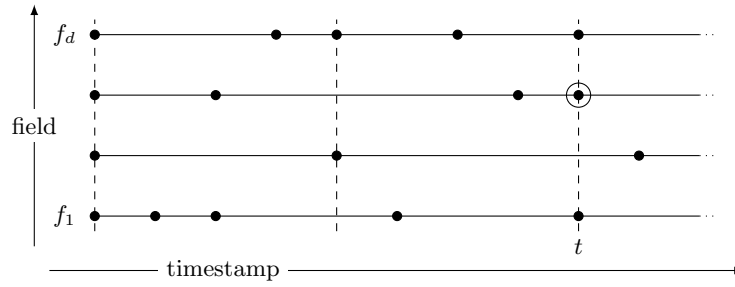


Fig. 2: Illustration of the fields of a node over time. The number of fields is $d = 4$ and $e = 4$. Each horizontal line represents one of the fields, with the dots denoting times when the value of the field change. In other words, the segments between dots represent time intervals where the field is unaltered. The vertical dashed lines are the times when the node is split. The dot with the circle around it is the start of the $(d + e + 1)$ th edge (when counting only edges after the previous split) introducing the split at time t . This results in $s_1 = 1$, $s_2 = 2$ and $a = 1$.

their start timestamp. The split is performed at the $(d + e + 1)$ th smallest start timestamp t . We include all edges with start timestamps strictly less than t in v_l . In v_r we include all edges with an end timestamp strictly larger than t .

Expanding on this, at time t we let s_1 denote the number of edges with start timestamp t among the first $d + e$ edges, and let s_2 be the number of edges with start timestamp t not among the first $d + e$ edges. Together $s_1 + s_2$ is the number of edges changing at time t , or equivalently the number of fields updated at time t . Furthermore, we call an edge *active* if its living span contains the splitting time, that is $t_{start} < t < t_{end}$, and let a denote the number of active edges at time t . See Fig. 2 for an example. Since all nodes have degree d exactly¹, we have $d = s_1 + s_2 + a$, where $s_2 \geq 1$ from the edge where we perform the split. The number of outgoing edges in v_l is $O_{v_l} = d + e - s_1 \leq d + e$. Likewise, the number of outgoing edges in v_r is $O_{v_r} = O_v - O_{v_l} + a = O_v - d - e + s_1 + d - s_1 - s_2 \leq O_v - (e + 1)$, where the inequality follows from $s_2 \geq 1$.

Finally, at time t there is at most one incoming edge to v from a parent v_p , since the underlying structure forms a tree. This edge must potentially be split in two which increases the outdegree of the parent by one. The difference in potential before and after this split is then at most:

$$\begin{aligned} \Delta\Phi &= \Delta\Phi_{v_p} + \Phi_{v_l} + \Phi_{v_r} - \Phi_v \\ &\leq 1 + 0 + \max\{0, O_v - (e + 1) - (d + e)\} - (O_v - (d + e)) . \end{aligned}$$

Now there are two cases depending on which term is larger in the maximum. We first look at the case when $0 \leq O_v - (e + 1) - (d + e)$. We call this case A

¹ Each node has exactly d fields and each field always holds an edge to either another node or to Nil.

and here we get $\Delta\Phi \leq -e$. The other case we call case B and here we get $\Delta\Phi \leq 0$ since $\Phi_v \geq 1$. Furthermore, since $O_v - (e + 1) - (d + e) < 0$ implies $O_{v_r} \leq O_v - (e + 1) < e + d$ then no further splits of v_r are needed. We can now upper bound the number of splits by the number of times these two cases occur. Since the potential never increases, case A can occur at most $\mathcal{O}(n/e)$ times. Next, since we split nodes bottom-up in the topological order, we never add edges to a node after it has been split. Thus, as case B only happens when we perform the last split of a node, it can occur at most $\mathcal{O}(n)$ times, once for each node in the freezer. Combining the two cases we see that the number of splits is $\mathcal{O}(n)$. \square

4 Proof of Theorem 1

Let D be a data structure that satisfies the TUNA conditions (Definition 1). We identify each node by a unique id and store timestamps with each edge denoting their living span. The freezer, see Section 3.1, stores edges on the form $(id_{\text{from}}, id_{\text{to}}, field, t_{\text{start}}, t_{\text{end}})$, where $field$ denotes the outgoing field of the edge from the node with id id_{from} , and the edge was live in the version interval $[t_{\text{start}}, t_{\text{end}}]$. The freezer further records what id the root of the data structure has for each time step, as well as what static value is associated with each id.

Applying an update to the structure can be done without saving the old structure, as condition A ensures that all nodes still present contain the same value. Furthermore, condition T ensures that the outgoing number of edges is d , leading to a constant number of updates to the freezer for each node updated, resulting in updates having unaltered asymptotic running time.

After applying all updates, recording the relevant information in the freezer, and obtaining tree T , condition U ensures that the freezer contains $\mathcal{O}(n)$ edges. For ease of argument, we then enter all edges from T into the freezer. Note that the number of elements in the freezer remains $\mathcal{O}(n)$.

To build the query DAG, apply a modified version of Kahn's algorithm [15] as described in Section 3.2, to perform a bottom-up topological sort of the graph induced by the edges in the freezer in time $\mathcal{O}(n \log n)$. Kahn's algorithm requires that the graph is acyclic, which condition N ensures. First, give each node, defined by id, the value stored in the freezer and the edges with matching id_{from} . Second, we employ the fat node technique [12] by allowing nodes to have $d + e$ edges. A fat node can overflow, if it gets more than $d + e$ edges. When an overflow is encountered the node is split into two nodes with the same value and the same parent but only a subset of the edges. Lemma 1 guarantees that splits only cause limited cascading while keeping nodes of degree $\mathcal{O}(d)$ and thus within a constant factor of their degree in the ephemeral structure as promised by condition T.

We now have a list of roots, sorted by time, that can be used to access previous versions of D . For queries, we assume the search starts from the relevant root. Otherwise, the relevant root can be found in $\mathcal{O}(\log r)$ time, where r is the total number of roots. Timestamps on edges represent the versions in which the edge was present, so it is easy to adapt queries to the DAG to only take into

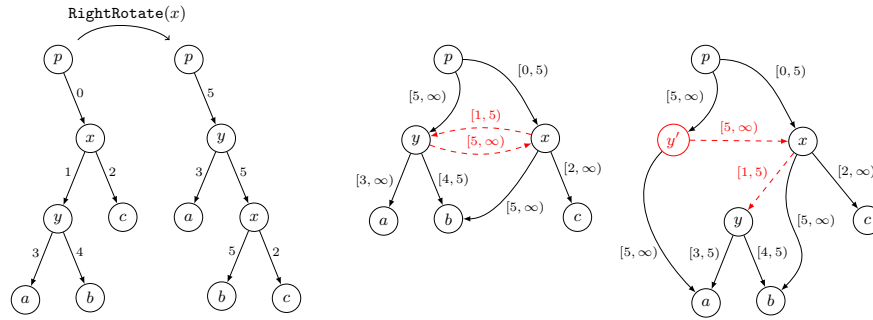


Fig. 3: Left: a rotation in a BST. Middle: the resulting graph contains the dashed cycle between x and y . Right: by creating a new copy y' of y the resulting graph remains a DAG and is TUNA compliant. We use the same notation as in Fig. 1.

account the relevant edges among the $\Theta(d)$ present edges in the node, leading to the same asymptotic running time for queries. \square

5 Further applications

Okasaki [18] introduces functional *random access arrays*, achieving, for an array of size n , worst-case lookup and update time $\mathcal{O}(\min\{i, \log n\})$, where i is the index of the queried element. This data structure easily satisfies the TUNA conditions, and can therefore be made offline partially persistent with only a constant factor space overhead.

Condition U of Definition 1 ensures that each of the n updates makes $\mathcal{O}(1)$ edges, which then totals to $\mathcal{O}(n)$ edges in the final structure. In the analysis of the space complexity, it is however not important exactly how many edges each update adds, and in fact, the more general property holds that the space usage is linear in the number of edges created by the ephemeral structure. Condition U can therefore be relaxed to each update producing for example amortized or expected $\mathcal{O}(1)$ new edges. This allows for balanced *Red-Black trees* [5] to fulfill the TUNA conditions even if the colors should be stored for persistent queries, as they make amortized $\mathcal{O}(1)$ color changes for each update [27]. The simpler functional implementation of Red-Black trees by Okasaki [20] makes amortized $\mathcal{O}(1)$ changes per insertion by a similar argument. See Fig. 3 on how rotations can be handled. Similarly, *Treaps* [3] fulfill the relaxed TUNA conditions, as each update makes expected $\mathcal{O}(1)$ rotations and therefore $\mathcal{O}(1)$ new edges.

Condition A ensures that all nodes can be reused and that storing edges is sufficient to produce the query DAG. The underlying tree is always represented explicitly, and as updates always operate on the current structure, it is possible to relax condition A, to allow for dynamic update information in each node. This information can be altered during the update phase and is not needed to produce the query DAG. Recall that our structure imposes additive $\mathcal{O}(n \log n)$

construction time independent of original update complexity. This allows for *AVL-trees* [1] to fulfill the TUNA conditions, as the balance value (the height of the subtree rooted at the node) is only used for balancing during the updates, and updates only perform amortized $\mathcal{O}(1)$ rotations [2].

6 Implementation and experiments

The construction described in this chapter has been implemented and tested in Haskell². Experiments were performed on WSL Ubuntu 20.04.6 on Windows 10.0.19044, with Processor 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz, 4 Core(s), 8 Logical Processor(s), 16 GB RAM, running `ghc` haskell compiler version 8.6.5, without any compiler flags. Each runtime measurement is the average of a constant number of runs on the same input. A random input is generated by providing the algorithm with a seed to a pseudo-random number generator.

For the following, let the version of a data structure made in our offline-partially-persistent framework be denoted as *persistent* and the version made in regular Haskell without necessarily saving the root of the structure be denoted *ephemeral*. The implementation uses time intervals for the edges in the freezer, and not only a start timestamp. Edges in the current structure are therefore not recorded in the freezer before they are removed from the live structure. Therefore, if any larger part of the structure is to be removed, it must be done so recursively, to correctly enter all of the edges into the freezer. Note that this does not alter the amortized running time of updates.

To test the *correctness* of the implementation, we apply various deterministic and random sequences of updates to both an ephemeral and a persistent unbalanced binary search tree, saving the roots of the ephemeral versions. We then construct the query DAG as described in Section 3.2 based on the persistent version. Then, for each timestamp, we tested if both versions produced the same tree. The test did not reveal any errors.

To measure space usage of a data structure, Haskell provides the function `recursiveSizeNF` which recursively measures the size of the object in bytes, i.e., traverses the whole pointer structure on the heap. Note that parts of the structure reachable from multiple places only are counted once. The space of the persistent data structure is measured after building the query DAG. Note that due to [25], a sequence of uniformly random insertions in an unbalanced binary search tree produces expected depth $\Theta(\log n)$.

The experiments for space usage, runtime of updates, and queries follow the expected result from the theory, see Figs. 4 and 5a–d. The experiment for the DAG building runtime appeared to be more than the theoretical $\mathcal{O}(n \log n)$ from the plot, see Fig. 5e. We are unable to find an explanation for the overhead. To ascertain the source of the overhead, we ran a sanity experiment to test if this overhead occurred on simpler problems. We inserted 1 to n into an unbalanced ephemeral BST where the insertion order created an almost perfectly balanced

² Available at <https://github.com/Crowton/Persistent-Functional-Trees>.

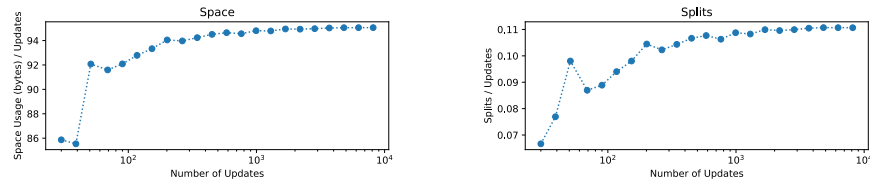


Fig. 4: Space usage experiments. Elements 1 to n are inserted into a persistent BST in order to create a path. Element $n + 1$ was then inserted and deleted n times, to introduce cascading node splits up the path, for a total of $3n$ updates.

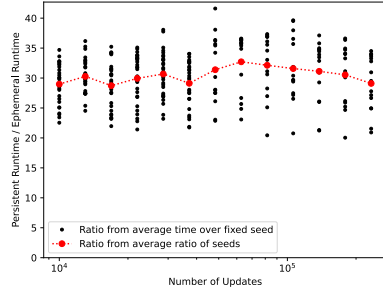
tree, via careful insertion of elements. Then all elements were queried in random order. This simple experiment has the clean theoretical runtime of $\mathcal{O}(n \log n)$ but similarly turned out not to produce an $\mathcal{O}(n \log n)$ plot in practice, see Fig. 5f, leading to the conjecture that the extra overhead in runtime is not from the program itself, but the Haskell compiler, specific implementations of underlying structures, and/or the environment the code is executed in.

The runtime experiments of updates and queries showed no issues with extra logarithmic factors, as only the relative runtime of the ephemeral and persistent implementation is compared. Here we found a constant factor difference, and thus the experiments did not disprove Theorem 1.

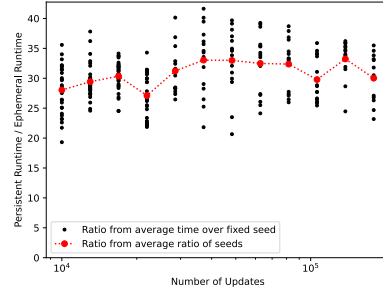
References

1. Adel’son-Vel’skii, G.M., Landis, E.M.: An algorithm for the organization of information. *Soviet Math. Doklady* **3**, 1259–1263 (1962)
2. Amani, M., Lai, K.A., Tarjan, R.E.: Amortized rotation cost in AVL trees. *Inf. Process. Lett.* **116**(5), 327–330 (2016). <https://doi.org/10.1016/j.ipl.2015.12.009>
3. Aragon, C.R., Seidel, R.: Randomized search trees. In: 30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989. pp. 540–545. IEEE Computer Society (1989). <https://doi.org/10.1109/SFCS.1989.63531>
4. Backus, J.: Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Commun. ACM* **21**(8), 613–641 (aug 1978). <https://doi.org/10.1145/359576.359579>
5. Bayer, R.: Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica* **1**, 290–306 (1972). <https://doi.org/10.1007/BF00289509>
6. Brodal, G.S., Okasaki, C.: Optimal purely functional priority queues. *Journal of Functional Programming* **6**(6), 839–857 (1996). <https://doi.org/10.1017/S095679680000201X>
7. Buchsbaum, A.L., Tarjan, R.E.: Confluently persistent dequeues via data-structural bootstrapping. *Journal of Algorithms* **18**(3), 513–547 (1995). <https://doi.org/10.1006/jagm.1995.1020>
8. Cole, R.: Searching and storing similar lists. *Journal of Algorithms* **7**(2), 202–220 (1986). [https://doi.org/10.1016/0196-6774\(86\)90004-0](https://doi.org/10.1016/0196-6774(86)90004-0)

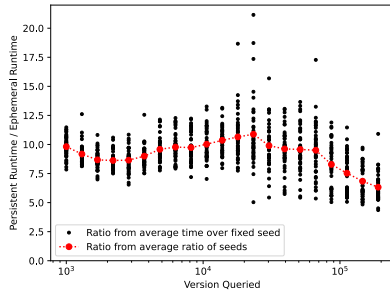
9. Demaine, E.D., Langerman, S., Price, E.: Confluently persistent tries for efficient version control. In: Scandinavian Workshop on Algorithm Theory. pp. 160–172. Springer (2008). <https://doi.org/10.1007/s00453-008-9274-z>
10. Dobkin, D., Lipton, R.J.: Multidimensional searching problems. *SIAM Journal on Computing* **5**(2), 181–186 (1976). <https://doi.org/10.1137/0205015>
11. Dobkin, D.P., Munro, J.I.: Efficient uses of the past. In: Proceedings of the 21st Annual Symposium on Foundations of Computer Science. pp. 200–206. SFCS '80, IEEE Computer Society, USA (1980). <https://doi.org/10.1109/SFCS.1980.18>
12. Driscoll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E.: Making data structures persistent. In: Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing. pp. 109–121. STOC '86, Association for Computing Machinery, New York, NY, USA (1986). <https://doi.org/10.1145/12130.12142>
13. Edelsbrunner, H., Guibas, L.J., Stolfi, J.: Optimal point location in a monotone subdivision. *SIAM Journal on Computing* **15**(2), 317–340 (1986). <https://doi.org/10.1137/0215023>
14. Hughes, J.: Why functional programming matters. In: Turner, D.A. (ed.) *Research Topics in Functional Programming*. pp. 17–42. The UT Year of Programming Series, Addison-Wesley (1990). <https://doi.org/10.1093/comjnl/32.2.98>
15. Kahn, A.B.: Topological sorting of large networks. *Commun. ACM* **5**(11), 558–562 (nov 1962). <https://doi.org/10.1145/368996.369025>
16. Kaplan, H., Tarjan, R.E.: Persistent lists with catenation via recursive slow-down. In: Proceedings of the twenty-seventh annual ACM Symposium on Theory of Computing. pp. 93–102 (1995). <https://doi.org/10.1145/225058.225090>
17. Kirkpatrick, D.: Optimal search in planar subdivisions. *SIAM Journal on Computing* **12**(1), 28–35 (1983). <https://doi.org/10.1137/0212002>
18. Okasaki, C.: Purely functional random-access lists. In: Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture. pp. 86–95. FPCA '95, Association for Computing Machinery, New York, NY, USA (1995). <https://doi.org/10.1145/224164.224187>
19. Okasaki, C.: *Purely Functional Data Structures*. Cambridge University Press, USA (1999)
20. Okasaki, C.: Red-black trees in a functional setting. *Journal of Functional Programming* **9**(4), 471–477 (1999). <https://doi.org/10.1017/s0956796899003494>
21. Overmars, M.H.: Searching in the past II: general transforms. Tech. rep., Tech. Rep. RUU (1981)
22. Overmars, M.H.: *The Design of Dynamic Data Structures*, Lecture Notes in Computer Science, vol. 156. Springer (1983). <https://doi.org/10.1007/BFb0014927>
23. Pippenger, N.: Pure versus impure lisp. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 104–109. POPL '96, Association for Computing Machinery, New York, NY, USA (1996). <https://doi.org/10.1145/237721.237741>
24. Sarnak, N., Tarjan, R.E.: Planar point location using persistent search trees. *Commun. ACM* **29**(7), 669–679 (jul 1986). <https://doi.org/10.1145/6138.6151>
25. Sedgewick, R., Flajolet, P.: *An Introduction to the Analysis of Algorithms*. Addison-Wesley Longman Publishing Co., Inc., USA (1996)
26. Seidel, R.: A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Computational Geometry* **1**(1), 51–64 (1991). [https://doi.org/10.1016/0925-7721\(91\)90012-4](https://doi.org/10.1016/0925-7721(91)90012-4)
27. Tarjan, R.E.: Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods* **6**(2), 306–318 (1985). <https://doi.org/10.1137/0606031>



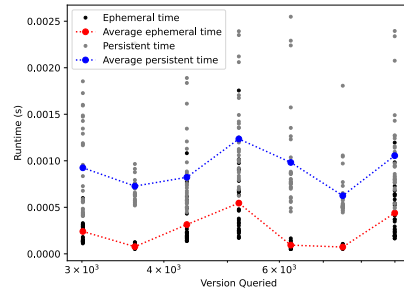
(a) Inserting elements 1 to n in random order.



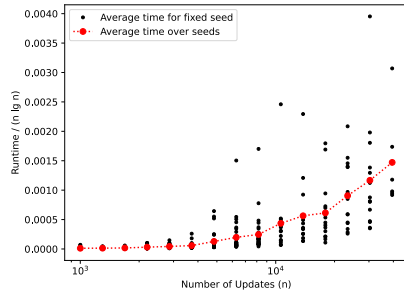
(b) Inserting and deleting elements 1 to n in random order.



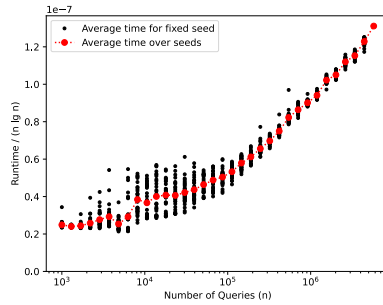
(c) Relative running time over querying all elements of a persistent BST in order at different timestamps. The tree is created by inserting elements 1 to 200000 in random order.



(d) Elements 1 to n are inserted to make a path. Element $n + 1$ is inserted and deleted n times, for $n = 3000$. Time measured for querying element $n + 1$ at different timestamps.



(e) DAG building running time experiment. Elements 1 to n were inserted and deleted from a persistent BST in random order. Time was measured on the DAG building alone.



(f) Sanity experiment. Querying all elements of an ephemeral perfectly balanced BST of size n in random order.

Fig. 5: Running time experiments.