Fully persistent arrays with optimal worst-case complexity

Milan Straka*

Abstract: We describe a RAM implementation of fully persistent array with worst-case $\mathcal{O}(\log \log m)$ time complexity for update and lookup, where m is the number of modifications to the array. The space complexity of such persistent array with n elements is $\mathcal{O}(n + m)$.

This lookup complexity is shown to be asymptotically optimal even for partially persistent arrays in the cell probe model using recent results of Pătraşcu and Thorup. Hence there cannot exist any fully persistent array, persistent hashtable nor any persistent graph with constant-time operations.

We also improve the complexity of update and lookup to $\mathcal{O}(\log \log \min(n, m))$ and show how to augment the algorithm to cooperate with the garbage collector to free unreachable data.

1 Introduction

Arrays are without question the most frequently used data structure in imperative programming. But in functional setting, arrays have to be fully persistent unless single-threaded use is somehow forced, e.g. using monads or linear or unique types.

A persistent array a supports two operations:

- *lookup*(i, a_v) returns the value of $a_v[i]$, the i-th element of array a_v,
- *update*(i, x, a_v) returns new array a_u obtained from a_v by performing $a_v[i] \leftarrow x$.

We call a_v a *version* of the array and we will sometimes write only v when the array is clear from context. The array is *partially persistent*, if we can perform lookup with any array version and update only with the most recent array version, and *fully persistent*, if we can perform update with any array version as well. In the whole paper, n is always the size of the array and m is the number of modifications to the array.

Various implementations of fully persistent arrays have been proposed, both purely functional ones and imperative ones. The purely functional implementations such as Hughes [13] are based on binary trees and have $\mathcal{O}(\log n)$ worst-case time and space complexity for update and $\mathcal{O}(\log n)$ worst-case time complexity for lookup, which cannot be improved in pointer machine model.

*Department of Applied Mathematics, Malostranské nám. 25, 118 00 Praha, Czech Republic; fox@kam.mff.cuni.cz

The imperative implementations achieve more. Even when using binary trees, one can make the space complexity constant per update using Driscoll et al. [12]. Moreover, Dietz [9] outlines an implementation with $\mathcal{O}(\log \log m)$ amortized time complexity for updates and the same worst-case complexity for lookups.

Here we develop an implementation working in random access model which has $\mathcal{O}(\log \log m)$ worst-case time complexity per update and lookup and constant memory complexity per update. Moreover we show how to improve the complexities to $\mathcal{O}(\log \log \min(n, m))$. As Demaine and Langerman pointed out in [7], the recent result of Pătraşcu and Thorup [16] can be used to show that this is the optimal lookup time in cell probe model, see section 2 for details.

There have been many attempts to get $\mathcal{O}(1)$ complexity for update and lookup. Any fully persistent implementation can be augmented by explicitly storing the newest value of each element to achieve $\mathcal{O}(1)$ complexities when the array is used single-threadedly. Other attempts include shallow binding of Baker [2], which was extended in Chuang [5]. Here the update operations are always constant and the worst-case of lookup is $\mathcal{O}(n)$. But the lookups which are grouped in voluminous read sequences can be performed in amortized constant time. Using randomization Chuang [6] improved the worst-case complexity of this scheme. Another approach was taken by O'Neill and Burton [15], where an $\mathcal{O}(\log n)$ worst-case implementation supports constant lookup complexity under the condition of uniformity of access.

The imperative implementations usually do not consider freeing of unused versions. But in many algorithms the majority of the array versions becomes inaccessible and could be freed. We show how to augment both the amortized and worst-case implementation to take deletions into account, while preserving $\mathcal{O}(\log \log m)$ complexity.

Our persistent array implementation uses an algorithm for maintaining list order. This problem has been extensively studied, e.g. in Tsakalidis [20], Dietz and Sleator [8] and Bender et al. [3]. Here we give simple worst-case constant time implementation and we use rebuilding to perform deletions in both amortized and worst-case implementations.

The rest of the paper is organized as follows. In section 2 we give the lower bound for lookup operation in partially persistent arrays. In section 3 we describe a simplified amortized $\mathcal{O}(\log \log m)$ implementation of fully persistent arrays. We improve the complexities to worst-case ones in section 4. In section 5 we show how to improve the complexity to $\mathcal{O}(\log \log \min(n, m))$. In section 6 we use rebuilding in both amortized and worst-cast implementation to support deletions of unreachable versions.

2 Lower bound

We show that the lower bound of lookup complexity of a partial persistent array is $\Omega(\log \log n)$ in the case that $m = n^\gamma$, where γ is a constant such that $1 < \gamma \leq 2$. This lower bound holds in the cell probe model, assuming that the space complexity of the array is $\mathcal{O}(M \log^k M)$ for a constant k, where M is the number of array modifications. The time complexity of an update is not limited.

2.1 Predecessor search problem

In the Predecessor search problem we are given a set Y of n integers of ℓ bits each, and the goal is to answer predecessor queries, i.e. to evaluate, for a given integer x, *predecessor(x)*$= \max\{y \in Y | y \leq x\}$. The paper of Pătraşcu and Thorup [16] shows that if we use S w-bit words to represent Y and define $a = \lg \frac{S}{n} + \lg w$ then the lower bound is, up to constant factors:

$$\min \begin{cases} \log_w n \\ \lg \frac{\ell - \lg n}{a} \\ \frac{\lg(\ell/a)}{\lg(a/\lg n \cdot \lg(\ell/a))} \\ \frac{\lg(\ell/a)}{\lg(\lg(\ell/a)/\lg(\lg n/a))} \end{cases}$$

The paper also shows a matching upper bound for each case. This lower bound works only for deterministic algorithms, but the later paper [18] shows the same lower bound for probabilistic algorithms as well. In the important case when $w = \ell = \gamma \lg n$ for a constant $\gamma > 1$ and near linear space ($S = n \cdot \lg^{\mathcal{O}(1)} n$), the optimal search time is $\mathcal{O}(\lg \ell)$.

2.2 Reducing predecessor search problem to persistent array lookup

The predecessor search problem is tightly connected to persistent array lookup, because *lookup*(i, a_v) returns a value $a_u[i]$ stored by an update operation, where u is the closest predecessor of version v (inclusive) that modified index i. As Demaine and Langerman [7] mention in a footnote, the predecessor search problem can be used to bound the partially persistent array lookup complexity. For completeness, let us show a reduction of the predecessor search problem to persistent array lookup, as it is not present in the cited paper [7].

Suppose we are given a set Y of n integers of ℓ bits each in the case that $w = \ell = \gamma \lg n$ for $1 < \gamma \leq 2$. Let $m = n^{\gamma/2}$ and let t be an auxiliary (nonpersistent) array of $m + 1$ elements. We create a partially persistent array a of m elements, every element initially set to -1. For each $i \in \{0, \ldots, m-1\}$ perform the following:

- for each $y \in Y$ such that $y \bmod m = i$ modify the current version of a by storing a value i at the index $\lfloor y/m \rfloor$,
- store the current version of a in $t[i]$.

Finally let p be another auxiliary nonpersistent array of m elements and set $p[i] \leftarrow$ *predecessor*$(i \cdot m)$ for $1 \leq i \leq m$.

Excluding the partially persistent array operations the time complexity is $\mathcal{O}(n)$ and the space complexity is $\mathcal{O}(m)$. The partially persistent array has m elements and we perform n updates on it, so it fits in $\mathcal{O}(n \log^k n)$ space.

To answer the *predecessor(x)* query we perform a lookup at index $\lfloor x/m \rfloor$ and version $t[x \bmod m]$. If the result is not -1, then the predecessor is the result increased by $m\lfloor x/m \rfloor$. If the result is -1 then the predecessor is stored in $p[\lfloor x/m \rfloor]$. Therefore we can compute the predecessor using one lookup and $\mathcal{O}(1)$ extra work.

Consequently, the lower bound for predecessor search problem can be applied, showing that the time complexity of partially persistent array lookup operation is at least $\Omega(\lg \ell)$, which is $\Omega(\lg \lg n)$.

3 Amortized $\mathcal{O}(\log \log m)$ implementation

We develop an amortized implementation of fully persistent array. Our implementation is based on Dietz [9], but it is simplified and better suited for the worst-case variant in the next section.

3.1 Partially persistent arrays

We first consider the simple case of partially persistent arrays. The core of the implementation is a data structure of van Emde Boas [4] (hereafter *vEB tree* or *vEBT*). This data structure can be implemented in random access model, works with integers of size bounded by $U = \mathcal{O}(2^w)$ and supports insertions, deletions and finding predecessors and successors in time $\mathcal{O}(\log \log U)$. This data structure can be made space efficient (i.e. $\mathcal{O}(1)$ space per update) using dynamic perfect hashing [11] at the expense of requiring randomization.

We implement partially persistent array as an array of vEB trees. We label the array versions by consecutive integers starting from 1. The update operation with the latest version v is implemented as an insertion into appropriate vEBT with key $v+1$. The lookup operation at version v is implemented by predecessor search in appropriate vEBT with key v.

This implementation has $\mathcal{O}(\log \log m)$ worst-case time complexity for update and lookup and uses $\mathcal{O}(m + n)$ space.

3.2 Linearizing the version tree

To obtain full persistence, we need to be able to navigate in the version graph, which is a tree in the case of full persistence. As in Driscoll et al. [12] or Dietz [9] we solve this problem by linearizing the version tree and obtaining a *version list*. This list is the result of a DFS traversal of the version tree, where each array version v is recorded twice, first as v_+ when it is first visited and second as v_- when it is left. We can maintain this list incrementally: when we create a version u from a version v, we insert u_+ right after v_+ and u_- right after u_+.

When performing *update*(i, x, v), we first set $x' \leftarrow$ *lookup*(i, v). If x' is the same as x, then we do nothing and return v. Otherwise we create a new version u, a descendant of the version v, store u_+ and u_- in the version list, set $a_{u_+}[i] \leftarrow x$ and $a_{u_-}[i] \leftarrow x'$ and return u. The effect of u_- is to undo the change of u_+.

We shall call any member of the version list a *version*, even though only the $\bullet_+$ elements are real versions and are used to identify a version of the array. The $\bullet_-$ elements are internal versions that are present because of linearization and have only undo effect. Moreover, each version is assigned an index i, such that the value of the element at index i is changed in this version. For a given i we say that all such versions *correspond to the index* i.

To perform *lookup*(i, v) we find a predecessor of the version v corresponding to the index i. We now recall the list order problem, which will eventually allow us to use vEBT to perform these predecessor searches.

3.3 List order problem

The *list order problem* is a well studied problem of maintaining a list order:

- *insert(u, v)* inserts a new element u right after v to the list,
- *query(u, v)* returns true iff u precedes v in the list.

In this section we briefly describe the algorithm of Bender et al. [3], which performs *query* in worst-case constant time and *insert* in amortized constant time. Let N be the number of elements in the list. We first show an amortized $\mathcal{O}(\log N)$ solution. Then we reduce the complexity to $\mathcal{O}(1)$ in random access model.

The algorithm assigns each element of the list a label, an integer tag from $[0, U)$, where U is polynomial in N. Each *insert* assigns a label to the new element and possibly relabels some existing elements. After each *insert*, the order of the elements of the list must the same as the order of integer labels.

There is a close connection between list labelling and weight-balanced trees. When we write integer label in binary, we can consider it a path in a tree, where 0 means left and 1 means right. Obviously, a strategy for maintaining tags in the universe $[0, U)$ for U polynomial in N with amortized relabel cost $f(N)$ yields a strategy for maintaining a balanced binary tree of height $\mathcal{O}(\log N)$ with amortized insertion time of $\mathcal{O}(f(N))$.

The straightforward converse is false, because a single rotation of the root of the tree can be performed in $\mathcal{O}(1)$ time, and changes paths to all leaves, thus causing $\mathcal{O}(N)$ relabellings. But if we define a *weight* of a vertex to be the number its of descendants and a *weight cost* of an operation to be the sum of weights of modified nodes, we get following result: Any balanced tree structure of maximum degree d and height h with (amortized) weight cost $f(N)$ for insertions yields a strategy for list labelling with (amortized) cost $\mathcal{O}(f(N))$ with tags from universe $[0, d^h)$.

Many rotation-based trees like red-black trees or AVL trees have $\mathcal{O}(N)$ weight cost insertions and deletions. Yet some structures with $\mathcal{O}(\log N)$ weight cost insertions exist, like BB[α] trees [14], skip lists [17] or weight-balanced B-trees [1]. These structures could be used to get an $\mathcal{O}(\log N)$ solution to the list order problem.

The advantage of the algorithm of Bender et al. [3] is that it does not explicitly maintain any tree. For integers $i, j \geq 0$ we say that tags $j \cdot 2^i, \ldots, j \cdot 2^i + 2^i - 1$ form a *range* of size 2^i. If the tag universe size U were a power of two, these ranges would correspond to vertices of the perfect binary tree with U leaves.

Let $1 < \delta < 2$ be a constant. We say a range is *in overflow* if it contains more than δ^i labels. Whenever we want to insert a new element between elements with labels e and f, we assign the new element any label between e and f. If $e + 1 = f$, we first find (by traversing the version list in appropriate directions) the smallest range containing e which is not in overflow. Then we relabel the elements contained in this range evenly. Then range of size 2 containing e is not in overflow and we can insert the new element.

It is easy to see that the resulting labels have at most $\lceil \log_\delta N \rceil$ bits. Amortized complexity of insert is $\mathcal{O}(\log N)$: We can relabel a range of size 2^i in time $\mathcal{O}(\delta^i)$ and after such relabelling, both child subranges contain $\delta^i/2$ labels. Another relabelling of this range will occur when one of these children contains more than δ^{i-1} labels, which happens after at least $\delta^{i-1}(1 - \delta/2)$ inserts.

Therefore for a given range it is enough to charge a constant time to each insert belonging to it. As a label belongs to $\mathcal{O}(\log_\delta N)$ ranges, the result follows. For more detailed proofs and for experiments with δ see the original paper [3].

Now assume that we are given an algorithm solving the list order problem with (amortized) time complexity $\mathcal{O}(\log N)$ for insert. We use it to obtain an algorithm with amortized time complexity $\mathcal{O}(1)$ per insert. We partition the list into continuous $\mathcal{O}(N/\log N)$ sublists of size $\mathcal{O}(\log N)$ each and use the given algorithm on $\mathcal{O}(N/\log N)$ elements representing these sublists. When inserting a new element after x, we insert it to the sublist containing x. If the size of the sublist reaches $c \log N$, we split it in halves and insert the second half into the top-level structure. As there must be $c/2 \log N$ inserts before new sublist is created, there are at most $\mathcal{O}(N/\log N)$ sublists. Because each top-level insert takes $\mathcal{O}(\log N)$ time, we get amortized constant time per insert.

We order elements in a sublist using this simple algorithm: when inserting an element between labels e and f, we give it the label $\lfloor \frac{e+f}{2} \rfloor$. As in the random access model the integer words have $\Omega(\log N)$ bits, the algorithm works for at least $\Omega(\log N)$ elements. During sublist splitting we relabel all labels evenly, creating space for $\Omega(\log N)$ inserts in the resulting sublist.

To perform a *query*, we find out whether the two list elements are in the same sublist. If so, we compare the labels inside the sublist. If not, we ask the top-level structure to compare the order of the sublists.

3.4 Amortized fully persistent arrays

The described algorithm for list order problem works by assigning labels to versions, such that the order of the labels and the order of the versions is the same. That is convenient for us – for each index in the persistent array we could store corresponding labels in a vEBT and use its integer predecessor search to implement list predecessor search.

The problem is that the described algorithm does not assign labels directly to the elements of a list. The label of each list element is a catenation of the sublist label and the label in the sublist. Though the insert operation works in amortized constant time, it changes $\Theta(\log m)$ labels, as changing a sublist label changes the labels of all its elements.

We could hope in a better algorithm, that would change only a constant number of labels per insert. But it was shown by Dietz et al. [10] that there is a lower bound of $\Omega(N \log N)$ relabelling during N inserts. So our list order algorithm is optimal with respect to the number of relabellings and we have to deal with more than constant number of relabellings per insert.

As in Dietz [9] we use *bucketing* to overcome the problem of relabellings (with some modifications). For each persistent array index i we do not store all corresponding versions. Instead, we partition versions corresponding to the index i to *buckets* of size $\mathcal{O}(\log m)$ and call the first version of a bucket a *bucket leader*. For each index i we store only labels of bucket leaders in vEBT and each bucket leader points to a binary balanced tree containing versions in this bucket. When performing *lookup*, we first find the right bucket using vEBT and then search the bucket for the right version, both operations carried out in $\mathcal{O}(\log \log m)$ time.

We have to connect sublists and buckets. We do so by introducing the

following invariant: every sublist contains at most one bucket leader. When we maintain buckets by splitting only the buckets of size $\Omega(\log m)$, this invariant does not asymptotically change the number of sublists.

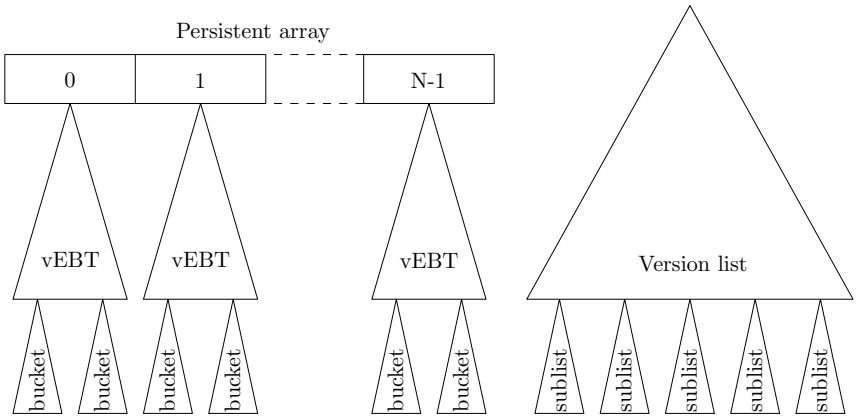

Figure 1: Fully persistent array implementation

We now describe the resulting implementation of a fully persistent array. We maintain a version list of versions. For each array index i we store versions corresponding to this index. The versions are split into continuous buckets of size $\mathcal{O}(\log m)$, and a bucket is split only if its size is $\Omega(\log m)$. The bucket leaders are stored in vEBT for this index i and every bucket forms a balanced binary tree pointed to by its bucket leader. The version list is split in sublists of size $\mathcal{O}(\log m)$, such that every sublist contains at most one bucket leader. A sublist is split only if its size is $\Omega(\log m)$ or if it contains two bucket leaders.

A *create(n, x_0)* operation creates initial version 0, at which all array elements have value x_0. This version is the only member of the version list and in each vEBT 0 is a bucket leader of a bucket containing exactly the version 0. We return a pointer to version 0 in the version list.

To perform *lookup*(i, v), we search the vEBT of index i for a predecessor of label of version v (inclusive). A bucket leader is found and we search corresponding balanced binary tree for predecessor of version v and return the value stored with that predecessor. These operations take $\mathcal{O}(\log \log m)$ time.

To perform *update*(i, x, v) we first perform $x' \leftarrow$ *lookup*(i, v) and if $x = x'$, then we just return v. Otherwise we insert versions u_+ and u_- in the version list. During this insert there is amortized constant number of sublist relabellings and sublist element relabellings. When changing a label of a sublist element, which is a bucket leader, we update the vEBT. As a sublist contains at most one bucket leader, change of the sublist label can be handled by updating one value in a vEBT as well. Now we insert version u_+ with value of x and version u_- with value of x' into the vEBT at index i. We do it by finding the right bucket and inserting the version to it. If the bucket gets bigger than $2 \log m$, we split the bucket in half and insert the bucket leader of the second half into the vEBT. Finally, we check that a sublist containing the new bucket leader does not contain any other bucket leader. If it does, we split this sublist into two.

Theorem 3.1 *The worst-case complexity of lookup is $\mathcal{O}(\log \log m)$ and the amortized complexity of update is $\mathcal{O}(\log \log m)$.*

Proof: The complexity of *lookup* is obvious. When not taking splitting of buckets into account, the amortized time of an *update* is $\mathcal{O}(\log \log m)$, because we perform amortized constant number of vEBT operations. Each bucket split takes $\mathcal{O}(\log \log m)$ time to split the balanced search tree and possibly the corresponding sublist. If the sublist is split, we spend $\mathcal{O}(\log m)$ time relabelling its elements and a new element must be inserted into top-level list order structure, which causes $\mathcal{O}(\log m)$ amortized relabellings of sublist labels. So a bucket split takes amortized $\mathcal{O}(\log m \log \log m)$ time. But there must be $\Omega(\log m)$ inserts into a bucket before it is split, hence it is enough to charge $\mathcal{O}(\log \log m)$ to each one of them.

4 Worst-case $\mathcal{O}(\log \log m)$ implementation

There are multiple places where we have to get rid of amortization. We have to maintain $\mathcal{O}(m/\log m)$ buckets of bounded size by spending only constant time per update on splitting, the list order structure must work in worst-case constant time per update and we must maintain the "one bucket leader per sublist" invariant.

4.1 Maintaining $\mathcal{O}(m/\log m)$ buckets of bounded size

We maintain exactly $\lfloor m/\log m \rfloor$ buckets. Whenever the number of allowed buckets increases, we create a new bucket by splitting the largest one. Obviously the next bucket will be created after at least $\log m$ steps.

It is clear that this algorithm maintains the right number of buckets. What we need to prove is the bound on the bucket sizes.

Theorem 4.1 *The size of buckets produced by this algorithm is* $\mathcal{O}(\log^2 m)$.

Proof: The proof of this theorem is based on a pebble game first introduced by Dietz and Sleator [8]. A pebble game starts with p empty piles, i.e. nonnegative real numbers $x_1, \ldots, x_p$, and proceeds by repeating following steps:

- the adversary chooses nonnegative real numbers $a_1, \ldots, a_p$ such that the some of them $\sum_i a_i = 1$ and sets $x_i \leftarrow x_i + a_i$,
- we find index i such that x_i is maximal and set $x_i \leftarrow 0$.

Dietz and Sleator (Theorem 5) proved that for any adversary no x_i ever exceeds $H_{n-1} + 1$, where $H_n = \sum_{i=1}^{n} i^{-1} \leq \ln n + 1$.

Raman [19] obtained a bound of $(1-\alpha)^{-1}(H_{n-1}+1)$ on all x_i if we can only perform $x_i \leftarrow \alpha x_i$ instead of zeroing, where $0 \leq \alpha < 1$. We present the proof here, as we need it. Let M_α be the yet unknown bound on all x_i. We define another pebble game with $y_i = \max(x_i - \alpha M_\alpha, 0)$. The key observation is that after performing $x_i \leftarrow \alpha x_i$ we have $y_i = 0$. Thus we play zeroing game on y_i, consequently all y_i are bounded by $H_{n-1} + 1$ and the result follows.

In our case the splitting of the buckets is just a pebble game with halving, where the adversary adds $\log m$ to the piles per turn. Although each split creates new nonempty bucket, from the mentioned proof we see that the bucket's y_i is zero. Thus we can imagine that the whole game took place on $m/\log m$ initially zeroed piles and we get the $\mathcal{O}(\log^2 m)$ bound on the size of the buckets.

4.2 List order problem in worst-case constant time

Amortized solution to the list order problem maintains $\mathcal{O}(m/\log m)$ sublists of size $\mathcal{O}(\log m)$. Our worst-case implementation will too maintain $\mathcal{O}(m/\log m)$ sublists, by splitting the largest sublist every $\log m$ updates. Similarly to Theorem 4.1 this results in sublists of size $\mathcal{O}(\log^2 m)$. We now describe how to work with sublists of this size and how to split them.

We start by describing a split of a sublist of size $\mathcal{O}(\log m)$. When the size of a sublist becomes equal to $2\log m$, we start the split. We fix the middle version v and create two empty sublists, one for versions preceding v and second for the rest of the versions. During following inserts we insert the elements to the old sublist, but we also copy elements to new sublists, three at a time. The copies are labelled during inserts, meaning a version has temporarily two labels, but we use the labels from the original sublist in queries. After $\log m$ such inserts we have copied all elements and we immediately switch to new labels and dispose the old sublist and old labels. The maximum size of the new sublists is $2\log m$, so $(2\log m)$-bit labels are enough during the whole algorithm.

Now we consider sublists of size $\mathcal{O}(\log^2 m)$. A sublist of this size is represented as a two-level tree with nodes of degree $\mathcal{O}(\log m)$, each node represented as a sublist of size $\mathcal{O}(\log m)$. When we insert an element, we insert it to the corresponding bottom node. When this node is of size $2\log m$, we use the described algorithm and start a split, inserting the newly created node to the top node. The label of an element is a catenation of the label in the top node and in the corresponding bottom node, so we can perform query in constant time.

To carry out a split of a sublists of size $\mathcal{O}(\log^2 m)$, we again find the middle version v. We split the bottom node containing v and then we split the top node according to v too. These two splits are splits of sublists of size $\mathcal{O}(\log m)$, so we can perform them during following $\mathcal{O}(\log m)$ inserts.

To complete the worst-case list order implementation, we need a top-level structure with $\mathcal{O}(\log m)$ worst-case time per insert: suppose we have such a structure. We maintain the sublists in the described way. Each $\log m$ inserts we start the split of the biggest sublist. We simultaneously perform an insert to the top-level structure, spreading the work during the following $\log m$ inserts. After these inserts both the top-level insert and the split of the biggest sublist is finished, so we can store the newly created sublist in the top-level structure. The only thing left to describe is the top-level structure.

4.3 Worst-case list order top-level structure

The top-level structure is a tree based on a *weight-balanced B-tree* [1], specialized for our needs. All its leaves are in the same depth and weight of a vertex of depth i is between 2^{i-1} and 2^{i+1}, where a *weight* of a node is the number of leaves in its subtree. Obviously a vertex of depth i has at most eight children, because its weight is at most 2^{i+1} and the weight of its children is at least 2^{i-2}.

Let this tree contain N elements. We show how an insert can be performed in $\mathcal{O}(\log N)$ worst-case time, such that we can update the labels of all elements.

We start by inserting a new leaf. Then we check its ancestors, starting from the lowest level, and if we find one with too big weight we split it. To split a node v of depth i we divide its children in two groups, the left children and the right children, such that the weights of those two groups are as close as possible.

Then we create nodes v_l with the left children and v_r with the right children, insert these nodes to the parent of v instead of v and continue by checking the weight of the parent of v.

If we split the children of v equally, the weight of both v_l and v_r would be 2^i. But there can be a child of weight 2^i, half of it missing in one group and half of it surplus in the other group, so the weight of v_l and v_r is in the range between $2^i - 2^{i-1} = 2^{i-1}$ and $2^i + 2^{i-1} = (3/2)2^i$. Therefore it is in the required range.

The children of a node are assigned 10-bit labels, such that the order of the children and the order of the labels is the same. The label of a leaf is a catenation of labels of the nodes on the path from the root to the leaf itself. We explicitly maintain both children labels and leaf labels. The children labels are updated immediately during split, but the leaf labels are updated gradually, though the order of the labels and the order of the leaves is the same all the time.

First consider the children labels. The first child gets the label 1000000000_2. When a child v is split into v_l and v_r, the label of v_l is the average of the label of the left sibling of v (or 0 if it does not exist) and the label of v; the label of v_r is the average of the label of v and the label of right sibling of v (or 1000000000_2 if it does not exist). As a node has at most 8 children, 8-bit labels are always enough. We elaborate on the two remaining bits later.

When the node v splits, we cannot update all the leaf labels in its subtree. Instead we start two relabelling processes. The *left relabel* relabels the leaves in the subtree of v_l and the *right relabel* relabels the leaves in the subtree of v_r. We consider here only the left relabel, the right is the same, only mirrored.

To relabel a leaf of v_l 20 bits of the label have to be changed – the 10 bits corresponding to v change to the label of v_l and the 10 bits corresponding to the label of the child of v containing this leaf change to the label of the child of v_l containing this leaf. When we split v, we associate the left relabel with v_l and during following insert into v_l we relabel 4 leftmost not relabelled leaves. As the weight of v_l is at most $(3/2)2^i$, there must be at least 2^{i-1} inserts into v_l before the next split, so we relabel all leaves in the subtree of v_l before the v_l splits.

During the left relabel a child w of v_l can split. If the leaves in its subtree are not yet relabelled, we cannot label the leaves in the subtree of w_l and w_r using the child label of w_l and w_r in v_l, as this would corrupt the ordering. Instead we must use the child labels w_l and w_r would get if they were children of v. As children of v_l can split at most twice before the left relabel finishes (they split after at least 2^{i-2} inserts), two spare bits of the child labels are enough to solve this.

Notice that when we change the label of a leaf, both the old and the new label preserve the ordering, as no labels between these two are used. Moreover, because the label changes are local, relabellings on different levels of a tree do not interfere.

We recapitulate the insert. After inserting the new leaf we check its ancestors, starting from the lowest level. If there is a relabelling process associated with an ancestor, we perform 4 its relabellings. If the weight of an ancestor v is too big, we split it and associate the left and the right relabels with v_l and v_r.

We conclude that the insert has a worst-case time complexity of $\mathcal{O}(\log N)$.

4.4 Implementation of worst-case fully persistent arrays

The resulting implementation of a worst-case fully persistent array is as follows. As in Figure 1 we maintain a version list, split into $\mathcal{O}(m/\log m)$ sublists of size $\mathcal{O}(\log^2 m)$. For each array index i we store versions corresponding to this index split into continuous buckets of size $\mathcal{O}(\log^2 m)$. The bucket leaders are stored in a vEBT for this index i and every bucket is represented as a balanced binary tree pointed to by its bucket leader. Every sublist contains at most one bucket leader.

Create and *lookup* operations are performed exactly as in amortized implementation. The *update* is also the same, but we do not split sublists and buckets when they get too full. Instead the splitting works in phases of length $2 \log m$. At the beginning of the phase we start the split of the largest sublist and we start the insert to the top-level structure, finishing both in $\log m$ steps and storing the new sublist in the top-level structure afterwards. Then we split the largest bucket, finishing immediately. If this split created second bucket leaders in a sublist, we start the split of this sublist, which finishes in following $\log m$ steps. Using the described list order data structure we conclude:

Theorem 4.2 *The worst-case complexity of lookup and update is* $\mathcal{O}(\log \log m)$.

5 Improving to $\mathcal{O}(\log \log \min(n, m))$

Improving the complexity to $\mathcal{O}(\log \log \min(n, m))$ is just a matter of splitting an array into independent pieces, each containing at most $\mathcal{O}(n)$ versions. We shall call these pieces *hunks*.

5.1 Amortized implementation

Whenever a hunk contains $2n$ versions, we perform a *split*. To peform a split, we find the version just in the middle of the version list and create two hunks, first containing versions preceding the middle version and second containing the rest of versions. Because we need the new hunks to be independent, we also create a snapshot of the whole array at the middle version and use this snapshot as the initial array value of the second hunk (the values associated with the version 0).

The actual implementation does not create two new hunks, but utilizes the existing hunk instead. Only the second hunk is created, with values of the middle version of the array, and then bigger versions are moved to this new hunk. The time complexity is linear, as we need to copy the initial n array values, n versions in the version list and we delete and subsequently insert $\mathcal{O}(n/\log\ n)$ values in vEBTs. As there are n updates between splits, it is enough to charge constant time to each insert, thus getting splits "for free".

5.2 Worst-case implementation

In the case of worst-case implementation, we cannot afford to do the whole *split* at once. Instead, we have to perform it step by step.

Whenever a hunk contains $2n$ versions, we start a split operation. We immediately identify the middle version and during the next $n/2$ updates we create a copy of the middle version of the array and create a new hunk from it. During

the following $n/2$ updates we move the versions bigger than middle version from the original hunk to the new hunk, four at a time. We finish on time, as we need to move at most n original versions and $n/2 + n/2$ recent versions. The same argument shows that the newly created hunks are of size at most $2n$.

The time needed by split is $\mathcal{O}(\log \log n)$ per one update to the hunk being split, so we once again get split "for free".

6 Supporting deletions

When a data structure is used in a functional environment, usually a garbage collector is used to free the unused data. Unlike purely functional implementations, imperative implementations must be augmented in order to facilitate this. We shall implement a function *delete*(v), which is supposed to be called by a garbage collector to indicate that the version v is no longer accessible (the so-called *finalizer*). A version which is not deleted is called *active*.

The *delete*(v) cannot immediately delete version v from version list and from vEBT, because the update which created version v_+ is visible until version v_-. So the version v can be really purged only when its effect is not observable between versions v_+ and v_- anymore. We call such a version *purgeable*.

Even if we are able to properly choose which versions to purge, another serious problem arises: it is not easy to maintain $\mathcal{O}(\log m)$-sized buckets/sublists. In the amortized implementation, we inspect only modified buckets, so if there are N elements in the list and then a lot of them are deleted, the untouched buckets can still be of size $\Theta(\log N)$, which can be unbounded in the current size. Likewise, the pebble game from worst-case implementation suffers from the same problem: if there are N elements in the list at one moment, even the optimal strategy results in a bucket of size $\Omega(\log\ N)$ *at some point in the future*, when $\log N$ can be also unbounded in the current size.

Because of these observations we do not handle *delete*s directly, as we still would have to tamper with all buckets even if only small number of them is accessed, and use a global rebuilding technique instead. If we denote the number of active versions by a, there can be $a \cdot n$ nonpurgeable versions, so the resulting implementations have $\mathcal{O}(\log \log(a \cdot n))$ time complexity and $\mathcal{O}(n + a \cdot n)$ space complexity.

6.1 Identifying purgeable versions

Let v be a version modifying index i. We call the version between v_+ and v_- *descendant* versions. A descendant version u is *captured*, if there exists a version w modifying index i, such that the order of versions is $v_+\ w_+\ u_+\ u_-\ w_-\ v_-$. The version v is purgeable, if there is no active uncaptured descendant version.

To recognize purgeable versions, we traverse the version list from left to right, counting for each v active descendant versions (na_v) and captured active descendant versions (nc_v). We use a stack and an array e of n elements. When we encounter a version v_+ modifying index i we push v and $e[i]$ to the stack and set $e[i] \leftarrow v$. When we encounter a version v_-, we pop v and reset $e[i]$ according to the stack. At this point we know na_v and nc_v, so we classify v as purgeable iff $na_v > nc_v$. We then add the number of active versions between v_+ and v_-

(inclusive, i.e. $na_v + 1$ if v is active) to the parent of v and to the $nc_{e[i]}$ if $e[i]$ is not empty.

This algorithm works in linear time and identifies all purgeable versions.

6.2 Amortized and worst-case implementation

In the amortized implementation we perform a *rebuild* after each $m/2$ inserts and deletes. The rebuild identifies purgeable versions and then creates a copy of the array containing only nonpurgeable versions. Such a rebuild can be performed in $\mathcal{O}(m)$ time, so its amortized complexity is $\mathcal{O}(1)$ per insert or delete.

In the case of worst-case implementation we cannot afford to do the whole *rebuild* at once. Instead we perform it step by step. Let the array contain N versions when we start rebuilding. We want to copy the nonpurgeable versions to a fresh empty array. We do so during following modifications to the array, copying three versions at a time. After at most $N/3$ steps all original versions are copied, and at most $N/3$ more have been created since. We copy these again in at most $N/9$ steps and iterate this copying algorithm, finishing after $\sum_{i=1}^{\infty} N \cdot 3^{-i} = N/2$ steps. At that moment both arrays are the same with respect to nonpurgeable versions, so we instantly switch to the just constructed one. When the rebuilding is finished, we immediately start a new rebuilding process.

7 Conclusion

We presented fully persistent array implementation with worst-case complexity $\mathcal{O}(\log\log\min(n, m))$ for lookup and update and shown that this lookup complexity is optimal in quite a general model. Thereby we fully analysed the potential of fully persistent arrays.

Fully persistent arrays are an important data structure as a functional analogue of arrays. But other data structures are based on arrays, e.g. any fully persistent graph data structure can be used as an array. On the other hand, one can implement fully persistent graph using fully persistent array. Thus we clarified the theoretical efficiency of important data structures in functional setting. We also studied how garbage collection of unused versions can be supported.

As mentioned by Dietz [9], any algorithm with worst-case complexities in random access model can be made fully persistent using a fully persistent array, with $\mathcal{O}(\log\log m)$ worst-case slowdown per operation.

The lower bound complexity of lookup applies only if we can access older versions of the array. If we are somehow able to limit the access to the latest version only, we can use an imperative array even in functional setting, thus being able to perform lookup and update in constant time.

Acknowledgements. I want to thank my supervisor Zdeněk Dvořák for his encouragement and a great deal of advice on this paper.

References

[1] L. Arge and J. Vitter. Optimal dynamic interval management in external memory. In *In FOCS*, 1996.

[2] H. G. Baker. Shallow binding makes functional arrays fast. *SIGPLAN Notices*, 26:1991–145, 1991.

[3] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *ESA '02: Proceedings of the 10th Annual European Symposium on Algorithms*, pages 152–164, London, UK, 2002. Springer-Verlag.

[4] P. V. E. Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Sys. Theory*, (10):99–127, 1977.

[5] T.-R. Chuang. Fully persistent arrays for efficient incremental updates and voluminous reads. In *4th European Symposium on Programming*, pages 110–129. Springer–Verlag, 1992.

[6] T.-R. Chuang. A randomized implementation of multiple functional arrays. In *In Proceedings of 1994 ACM Conference on Lisp and Functional Programming*, pages 173–184. ACM Press, 1994.

[7] E. D. Demaine, S. Langerman, and E. Price. Confluently persistent tries for efficient version control. In *SWAT '08: Proceedings of the 11th Scandinavian workshop on Algorithm Theory*, pages 160–172, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *STOC '87: Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 365–372, New York, NY, USA, 1987. ACM.

[9] P. F. Dietz. Fully persistent arrays (extended abstract). In *WADS '89: Proceedings of the Workshop on Algorithms and Data Structures*, pages 67–74, London, UK, 1989. Springer-Verlag.

[10] P. F. Dietz, J. I. Seiferas, and J. Zhang. A tight lower bound for online monotonic list labeling. *SIAM J. Discret. Math.*, 18(3):626–637, 2005.

[11] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994.

[12] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, 1989.

[13] J. Hughes. An efficient implementation of purely functional arrays. Technical report, Dept. of Computer Sciences, Chalmers University of Technology, Göteborg, 1985.

[14] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. In *STOC '72: Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 137–142, New York, NY, USA, 1972. ACM.

[15] M. E. O'Neill and F. W. Burton. A new method for functional arrays. *Journal of Functional Programming*, 7:1–14, 1997.

[16] M. Pătraşcu and M. Thorup. Time-space trade-offs for predecessor search. In *STOC '06: Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, pages 232–240, New York, NY, USA, 2006. ACM.

[17] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. In *In WADS*, 1989.

[18] M. Pătraşcu and M. Thorup. Randomization does not help searching predecessors. In *SODA '07: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 555–564, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[19] R. Raman. *Eliminating amortization: on data structures with guaranteed response time*. PhD thesis, Rochester, NY, USA, 1993.

[20] A. K. Tsakalidis. Maintaining order in a generalized linked list. *Acta Inf.*, 21(1):101–112, 1984.