On-the-Fly Array Initialization in Less Space*

Torben Hagerup[1] and Frank Kammer[2]

1 **Institut für Informatik, Universität Augsburg, Augsburg, Germany**
`hagerup@informatik.uni-augsburg.de`
2 **MNI, Technische Hochschule Mittelhessen, Gießen, Germany**
`frank.kammer@mni.thm.de`

Abstract

We show that for all given $n, t, w \in \{1, 2, \ldots\}$ with $n < 2^w$, an array of n entries of w bits each can be represented on a word RAM with a word length of w bits in at most $nw + \lceil n(t/(2w))^t \rceil$ bits of uninitialized memory to support constant-time initialization of the whole array and $O(t)$-time reading and writing of individual array entries. At one end of this tradeoff, we achieve initialization and access (i.e., reading and writing) in constant time with $nw + \lceil n/w^t \rceil$ bits for arbitrary fixed t, to be compared with $nw + \Theta(n)$ bits for the best previous solution, and at the opposite end, still with constant-time initialization, we support $O(\log n)$-time access with just $nw + 1$ bits, which is optimal for arbitrary access times if the initialization executes fewer than n steps.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Data structures, space efficiency, constant-time initialization, arrays

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2017.44

1 Introduction

Whereas the space used by an algorithm (measured in "memory units" such as words) is usually bounded by its running time, there may be exceptions if the memory offers random access, and it is occasionally useful to employ large arrays of which only a small part will ever be accessed. A case in point are adjacency matrices, which are a convenient representation of graphs if the algorithms to be executed issue adjacency queries (e.g., "does G contain an edge from u to v?") in an irregular pattern that cannot be served efficiently using adjacency lists. Even if one can afford the space needed by an adjacency matrix, it may be prohibitively expensive to clear all those entries in the matrix that do not correspond to edges in the graph. The problem does not occur if the memory cells allocated to hold the adjacency matrix can be assumed to be already initialized to some particular value (that can be taken to signify "no edge"), but in general this is not a realistic assumption. Therefore the problem of simulating an initialized array in an uninitialized memory has been considered since the early days of computing.

Additional motivation for our work comes from the fact that certain modern programming languages such as Java, VHDL and D stipulate that memory be initialized (e.g., cleared to zero) before it is allocated to application programs [8, 12] or have this as the default behavior [2]. The initialization is carried out for security reasons and to ease debugging by making faulty programs more deterministic. If it can be ensured that application programs access memory only through a well-defined interface, one may hope to let the interface provide conceptually cleared memory while avoiding the overhead of clearing the memory physically.

* A fuller version of this paper is available as [11], `https://arxiv.org/abs/1709.10477`.

For some $w \in \mathbb{N} = \{1, 2, \ldots\}$, our model of computation is a word RAM [3, 9] with a word length of w bits, where we assume that w is large enough to allow all memory words in use to be addressed. As part of ensuring this, in the context of an array of size n we always assume that $n < 2^w$. The word RAM has constant-time operations for addition, subtraction and multiplication modulo 2^w, division with truncation ($(x, y) \mapsto \lfloor x/y \rfloor$ for $y > 0$), left shift modulo 2^w ($(x, y) \mapsto (x \ll y) \bmod 2^w$, where $x \ll y = x \cdot 2^y$), right shift ($(x, y) \mapsto x \gg y = \lfloor x/2^y \rfloor$), and bitwise Boolean operations (AND, OR and XOR (exclusive or)). We also assume a constant-time operation to load an integer that deviates from $\sqrt{w}$ by at most a constant factor – this enables the proof of Lemma 3. The problem of central concern to us is to realize a clearable word array, defined as follows:

▶ **Definition 1.** A *clearable word array* is a data structure that can be initialized with an integer $n \in \mathbb{N}$ and subsequently maintains an element of $\{0, \ldots, 2^w - 1\}^n$, called its *client sequence* and initially $(0, 0, \ldots, 0)$, under the following operations:

read(ℓ) $(\ell \in \{0, \ldots, n-1\})$: If the client sequence before the call is $(x_0, \ldots, x_{n-1})$, returns x_ℓ without changing the client sequence.

write(ℓ, x) $(\ell \in \{0, \ldots, n-1\}$ and $x \in \{0, \ldots, 2^w - 1\})$: If the client sequence before the call is $(x_0, \ldots, x_{n-1})$, changes the client sequence to be $(x_0, \ldots, x_{\ell-1}, x, x_{\ell+1}, \ldots, x_{n-1})$.

The clearable word array is a special case of the *initializable array* of Navarro [15]. There are two differences. First, the data structure of Navarro is more general in that the initialization, in addition to n, receives a second parameter v that is taken to be the initial value of the array entries, i.e., the initial value of the client sequence is $(v, v, \ldots, v)$ rather than $(0, 0, \ldots, 0)$. As is easy to see and will be discussed in Section 3, however, the more general data structure reduces easily to the more restricted one. Second, Navarro does not specify the nature of the array entries, which is of no relevance to his approach, whereas we fix the array entries to be *words*, i.e., elements of $\{0, \ldots, 2^w - 1\}$. Again, this will turn out to be a restriction of little consequence.

Following the initialization of a clearable word array with an integer n, we call n the *universe size* of the data structure. We shall have occasion to consider restricted clearable word arrays that can be initialized only for certain specific universe sizes. Because the connection between the client sequence of an initializable array and an array used to hold it is often very close, it is easy to confuse the two. We may view the client sequence as an array, but then use the letter 'a' to denote this abstract array (which is initialized) and 'A' to denote the corresponding physical array (which is not initialized).

2 Previous Work

Fredriksson and Kilpeläinen [7] give a detailed overview of the known approaches to array initialization and compare them experimentally. In the discussion of their work, we assume that the task is to realize an initializable array of n entries of $b \leq w$ bits each. Define the *redundancy* of a data structure that solves this problem and occupies N bits to be $N - nb$, i.e., the number of bits used beyond the minimum of nb bits needed even without the requirement of initializability.

A number of the methods described by Fredriksson and Kilpeläinen can be viewed as special cases of a general *trie method*. Ignoring rounding issues, the trie method is parameterized by an integer $h \in \mathbb{N}$ and a *degree sequence* $(d_1, d_2, \ldots, d_h)$ of h positive integers with $\prod_{i=1}^{h} d_i = nb$. It uses a tree T of height h in which all nodes of height i have d_i children, for $i = 1, \ldots, h$. Each node in T has an associated bit, the bits of each maximal group of siblings are stored compactly, w bits to a word, and the nb bits at the leaves are identified with the nb bits of the abstract array a.

Let *processing* an inner node u in T be the following: If the bit associated with u has the value 0 (informally, u has been initialized, but its children have not), initialize the bits of all children of u, to 0 if the children are inner nodes and to the prescribed initial value v – within groups of b siblings in the obvious manner – if they are leaves. If u has d children, this can be done in $O(\lceil d/w \rceil)$ time. Finally set the bit associated with u to 1. If the value of that bit is 1 already prior to the processing of u, the processing of u terminates immediately after discovering this fact.

To initialize T, set the bit at its root to 0. In addition, it is permissible, as part of the initialization, to process the inner nodes in an upper part of T in a top-down fashion, i.e., so that no nonroot node is processed before its parent. We will say that such nodes are *preprocessed*. To read the ℓth entry of a, descend in T towards the ℓth group of b leaves. If an inner node is encountered whose associated bit has the value 0, return v. If not, return the value found in the ℓth group of b leaves. To write the ℓth entry of a, descend in the same manner towards the ℓth group of b leaves, process every inner node encountered on the way, and finally store the appropriate value in the bits of the ℓth group of b leaves. The total number of bits used by the data structure is the number of nodes in T that are not preprocessed, the initialization takes constant time plus time proportional to the sum of $\lceil d/w \rceil$ over all degrees d of preprocessed nodes, the worst-case time of *read* is $\Theta(h)$, and the worst-case time of *write* is the maximum over all leaves v in T of $\Theta(h + \sum_i \lceil d_i/w \rceil)$, where the sum ranges over those values of $i \in \{1, \ldots, h\}$ for which the ancestor of v of height i is not preprocessed.

Fredriksson and Kilpeläinen consider the following special cases of the trie method: Degree sequence (nb), preprocess the root (Plain); degree sequence (b, n), preprocess the root (Simple); degree sequence $(b, w, w, \ldots, w)$ (Hierarchic); degree sequence $(b, n/w, w)$ (Simple-H); and degree sequence $(b, w, n/w)$, preprocess the root (SHV). The redundancy is 0 for Plain and close to n (i.e., the number of nodes in T of height 1) for the other methods. The initialization time is $\Theta(1 + nb/w)$ for Plain, $\Theta(1 + n/w)$ for Simple, $\Theta(1 + n/w^2)$ for SHV and $\Theta(1)$ for the other methods. The worst-case time for *read* is $\Theta(1 + \log_w n)$ for Hierarchic and $\Theta(1)$ for the other methods. The worst-case time for *write*, finally, is $\Theta(1 + \log_w n)$ for Hierarchic, $\Theta(1 + n/w^2)$ for Simple-H and $\Theta(1)$ for the other methods.

None of the methods discussed above combines constant initialization time with constant access time, and it is easy to see that this is true of every instance of the trie method. Constant time for every operation is achieved by a folklore method that goes back at least to the early 1970s (see [1, Exercise 2.12]). The folklore method uses a physical array A with the index set $\{0, \ldots, n-1\}$ and assigns the codes $0, 1, \ldots$ to the indices of the abstract array a in the order in which the indices are first used in calls of *write*, x_ℓ is stored in $A[f(\ell)]$, where $f(\ell)$ is the code of ℓ, two tables are used to keep track of the encoding function f and its inverse f^{-1}, and finally the data structure remembers the number k of codes assigned. To access x_ℓ, first $f(\ell)$ is looked up in the table of f. Because the table is not initialized, the purported code j may not be correct, but j is the code of ℓ exactly if $0 \le j < k$ and the entry of j in the table of f^{-1} is ℓ. If not, the default initial value v is returned in the case of a read operation, and the next available code is assigned to ℓ in the case of a write operation. The remainder of the access is simply a reading or writing of $A[f(\ell)]$. The structure is initialized by setting k to 0. In addition to the space needed to hold the actual data in A, it needs space for the tables of f and f^{-1} and the counter k, so that its redundancy is $2n\lceil \log_2 n \rceil + \lceil \log_2(n+1) \rceil$.

A family of methods due to Navarro [15] combines the Hierarchic method above with the folklore method. The idea is, starting from Hierarchic, to replace the nodes of height $\ge h+2$, for some $h \ge 0$, by an instance of the folklore data structure. This achieves the same effect

as processing the nodes that were removed and obviates the need to descend through these nodes during an access. The initialization time is constant, the worst-case access time is $\Theta(h+1)$, and the redundancy is approximately $3n$ for $h=0$ and approximately n for $h \geq 1$.

3 Our Contribution

We give an upper-bound tradeoff that spans the entire range from minimal time to minimal space. Our main result is the following:

▶ **Theorem 2.** *There is a clearable word array that, for all given $n, t \in \mathbb{N}$, can be initialized for universe size n in constant time and subsequently occupies at most $nw + \lceil n(t/(2w))^t \rceil$ bits and supports read and write in $O(t)$ time.*

If w and hence (by assumption) n are bounded by constants, it is trivial to realize a clearable word array with constant initialization and access times and zero redundancy (initialize the array explicitly, i.e., use the Plain method of Fredriksson and Kilpeläinen). Given a constant $t \in \mathbb{N}$, we can therefore assume without loss of generality that $w \geq t^2$. Then $(t/w)^2 \leq 1/w$ and hence $(2t/(2w))^{2t} \leq 1/w^t$. Theorem 2 (used with t doubled) thus implies that for all constant $t \in \mathbb{N}$, there is a clearable word array that can be initialized in constant time, executes accesses in constant time and has redundancy $\lceil n/w^t \rceil$. The best previous constant-time solution, due to Navarro [15] and discussed above, has redundancy $n + o(n)$.

At the other end of the time-space tradeoff, for $t = \lceil \log_2 n \rceil$, the redundancy of Theorem 2 is 1, i.e., the constant-time initialization costs only a single bit and accesses are still supported in logarithmic time. If an initialization time of $\Theta(n)$ is acceptable, a clearable word array with constant-time access can obviously be realized with zero redundancy – this is again the Plain method of Fredriksson and Kilpeläinen. On the other hand, the redundancy cannot be reduced below our bound of 1 for any access times unless the initialization writes to at least n words, which needs at least n steps. To see this, assume that a clearable word array with universe size n is represented in N bits for some $N \in \mathbb{N}$. Because the client sequence can be in any one of 2^{nw} states, any two of which can be distinguished through *read* operations, whereas its representation can be in only 2^N states, we must have $N \geq nw$, irrespectively of all operation times. Moreover, if $N = nw$, every state of the client sequence is represented by exactly one bit pattern of its representation. Since the client sequence is in a well-defined state immediately after the initialization, this is impossible unless each of the nw bits of its representation is forced to one specific value during the initialization, i.e., unless the initialization writes to at least n words.

Note that it is a responsibility of the user of a clearable word array initialized for universe size n to ensure that $\ell < n$ in all calls of the form *read*(ℓ) or *write*(ℓ, x) issued to the data structure. Whereas the data structure can easily check the conditions $\ell \geq 0$ and $0 \leq x < 2^w$, when operated close to its minimum space it cannot afford to store the integer n. Thus illegal calls of its operations may go undetected and may lead to attempted accesses to memory words outside of the area assigned to the data structure.

Our result can be seen as a second application of the *light-path technique*, which was introduced (but not named) in [10] and used there to construct space-efficient nonsystematic choice dictionaries. From a technical perspective, the situation is simpler here, as there is no need to store data in a particular *compact representation* and to provide conversion to and from the compact representation. This gives us an opportunity to illustrate the light-path technique in a purer setting. At a more abstract level, the fundamental idea is to upset the structure of a simple table slightly in order to accommodate additional information in the

table. Whereas this principle has been used before [4, 5, 14], curiously, it has not so far been employed in the setting of initializable arrays even though it seems particularly natural there. It may be noted that the c-color choice dictionaries of [10] could be used directly as initializable arrays, but efficiently so only for arrays whose elements are drawn from a very small range $\{0,\ldots,2^b-1\}$. This is because each element of that range would be considered a separate color, i.e., we would have $c=2^b$.

Given the clearable word array of Theorem 2, it is easy to derive a more general data structure that, for some integer b with $1\le b\le w$, maintains a client sequence in $\{0,\ldots,2^b-1\}^n$, initially $(0,0,\ldots,0)$, under reading and writing of individual elements of the sequence. Simply pack the n elements of the client sequence tightly in $\lceil nb/w\rceil$ words of w bits each, initialize the used part of the last word to 0, maintain the other words in a clearable word array, inspect a b-bit element of the client sequence by reading the at most two words over which the b bits spread, picking out the relevant pieces of the words and concatenating the pieces, and update a b-bit element of the client sequence correspondingly by splitting the new value into at most two pieces and storing each piece appropriately in a word without disturbing the rest of the word. The execution times are within a constant factor of those of the clearable word array, and the number of bits needed is at most $nb+\lceil n(t/(2w))^t\rceil$.

We can also easily derive a data structure more general than that of Theorem 2 in that the client sequence is initialized to $(g(0),\ldots,g(n-1))$, where $g:\{0,\ldots,n-1\}\to\{0,\ldots,2^w-1\}$ is some function, rather than to $(0,0,\ldots,0)$. The simple idea is to swap the representations of the "internal" and "external" initial values. Both *read*(ℓ) and *write*(ℓ,x) then begin by evaluating $g(\ell)$. If reading the value associated with ℓ in a normal clearable word array yields the value 0, *read*(ℓ) returns $g(\ell)$. If the value read is $g(\ell)$, *read*(ℓ) returns 0, and every other value read is returned as it is. Similarly, if $x=g(\ell)$, *write*(ℓ,x) actually writes the value 0 to the normal clearable word array, $x=0$ causes the value $g(\ell)$ to be written, and every other value of x is written as it is. The initialization and access times are those of Theorem 2 plus whatever time is needed to initialize g and to evaluate it on one argument, respectively, and the space requirements are those of Theorem 2 plus those of g. It is easy to see that the generalizations described in this and the previous paragraph can be combined.

Very recently, giving a clever twist to the folklore method, Katoh and Goto [13] devised a clearable word array that executes every operation in constant time but, when the universe size is n, uses just $nw+1$ bits.

4 The Construction

In this section we prove Theorem 2. At a very low and technical level, we need the following staple of word-RAM computing.

▶ **Lemma 3** ([6, 10]). *Given a nonzero integer* $\sum_{i=0}^{w-1}2^ib_i$*, where* $b_i\in\{0,1\}$ *for* $i=0,\ldots,w-1$*, constant time suffices to compute* $\max I$ *and* $\min I$*, where* $I=\{i\mid 0\le i\le w-1$ *and* $b_i=1\}$*.*

Let a *colored tree* be an ordered outtree, each of whose leaves is either *white* or *black*. Given a colored tree T, we extend the colors at the leaves of T to its inner nodes as follows: If the leaf descendants of an inner node u all have the same color (white or black), then u has that same color. If u has both a white and a black leaf descendant, u is *gray*. Clearly every ancestor of a node v has the same color as v or is gray. In particular, every ancestor of a gray node is gray. Define the *navigation vector* of an inner node to be the sequence of the colors of its children in the order from left to right.

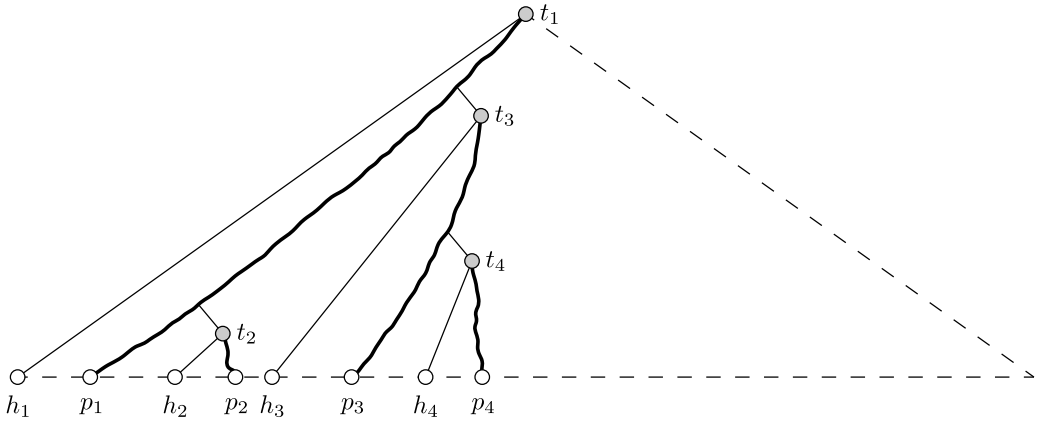

Figure 1 Example light paths (drawn thicker). Top nodes, historians and proxies are labeled "t", "h" and "p", respectively, and a subscript identifies the associated light path.

Recall that the *left spine* of a rooted ordered tree T is the maximal path in T that starts at the root of T and, whenever it contains an inner node u, also contains the leftmost child of u. Define the *preferred child* of a white or gray inner node in a colored tree T to be its leftmost gray child if it has at least one gray child, and its leftmost white child otherwise. Call an edge in T *light* if it leads from a gray inner node to its preferred child or lies on the left spine of a subtree of T whose root is white and has a gray parent of which it is the preferred child. In other words, every gray inner node picks the edge to its preferred child to be light, whereas a white inner node does so only if "prompted" by its parent. The light edges induce a collection of node-disjoint paths called *light paths*, each of which ends at a leaf in T. When P is a light path that starts at a (gray) node u and ends at a (white) leaf v, we call u the *top node*, v the *proxy* and the leftmost leaf descendant of u (that may coincide with v) the *historian* of P and of every node on P. These concepts are illustrated in Fig. 1. A gray node that is not the root of T is a top node exactly if it is not the preferred child of its parent, i.e., if it has at least one gray left sibling. No proper ancestor of a top node u can have a descendant of u as its leftmost leaf descendant, so a leaf is the historian of at most one light path. If h is the historian of a light path P, the top node and the proxy of P are also said to be the top node and the proxy, respectively, of h. A leaf ℓ cannot be the historian of one light path and the proxy of another, since otherwise the two corresponding top nodes would both be ancestors of ℓ and the path between them would contain only gray nodes and be part of a light path, a contradiction. A similar argument shows that in the left-to-right order of the leaves of T, no historian or proxy lies strictly between a historian and its proxy. Define the *history* of a light path that contains the nodes $u_1, \ldots, u_k$, in that order, to be the sequence $(q_1, \ldots, q_{k-1})$, where q_i is the navigation vector of u_i, for $i = 1, \ldots, k-1$ (u_k, as a leaf, has no navigation vector).

The following lemma describes the work-horse of our data structure.

▶ **Lemma 4.** *Let d and t be given positive integers with $2dt \leq w$ such that d is a power of 2. Then there is a clearable word array that can be initialized for universe size $n = d^t$ in constant time and subsequently occupies $nw + 2$ bits and, if given access to the parameters d and t, supports read and write in $O(t)$ time.*

Proof. Without loss of generality assume that $d \geq 2$. We use a conceptual colored tree T that is a complete d-ary tree of height t and identify the n leaves of T, in the order from left to right, with the integers $0, \ldots, n-1$. Let r be the root of T and, for each node u in

T, let T_u be the maximal subtree of T rooted at u. We represent a node u of height j in T through the pair (j, k), where k is the number of nodes in T of the same height as u and strictly to its left (in other words, the nodes on each level in T are numbered consecutively in the order from left to right, starting at 0). Then navigating in T is easy: If u is not the root r, its parent is (represented through) $(j+1, \lfloor k/d \rfloor)$, if u is not a leaf, its children are $(j-1, kd), \ldots, (j-1, kd+(d-1))$, u's leftmost leaf descendant is $(0, kd^j)$ (identified with the integer kd^j), and if u is not a leaf and ℓ is a leaf descendant of u, then $\mathit{viachild}(u, \ell)$, the child of u that is an ancestor of ℓ, is $(j-1, \lfloor \ell/d^{j-1} \rfloor)$. The assumption that d is a power of 2 ensures that we can compute the necessary powers of d in constant time by means of multiplication and left shift. This requires the availability of $\log_2 d$, which can be computed from d in constant time according to Lemma 3.

The actual data is stored in a word array A with index set $\{0, \ldots, n-1\}$ and in two additional *root bits*. The three colors white, gray and black are encoded in two bits, the navigation vector of an inner node in T is represented by the $2d$-bit concatenation of the representations of its d (color) elements, and the history of a k-node light path is represented by the $2d(k-1)$-bit concatenation of the representations of its $k-1$ (navigation-vector) elements. The relation $2dt \leq w$ ensures that every history fits in a w-bit word. Assume that a history of fewer than w bits is "right-justified" in the word so that the position in the word of the navigation vector of a node depends only on the height of the node.

With the aid of an algorithm of Lemma 3, the preferred child of a given white or gray inner node u in T can be computed in constant time from the navigation vector of u or a history that contains that navigation vector. This may need a couple of bit masks (informally, ones that correspond to all nodes having the same color) that can easily be obtained via multiplication with the integer $1_{dt,2} = (2^{2dt}-1)/3$, whose $(2dt)$-bit binary representation is $0101\cdots0101$. Because 2^{2dt} may not be representable in a w-bit word (namely if $2dt = w$), the computation of $1_{dt,2}$ needs a little care, but is still easy to do in constant time.

The client sequence $(x_0, \ldots, x_{n-1})$ is represented in $A[0], \ldots, A[n-1]$ and the two root bits according to the following storage invariants: First, the two root bits indicate the color of the root r of T. Second, for $\ell = 0, \ldots, n-1$,

- if ℓ is a historian, $A[\ell]$ stores the history of the proxy of ℓ (hence the term "historian"),
- if ℓ is black and not a historian, $A[\ell]$ stores x_ℓ,
- if ℓ is a proxy whose historian h is black, $A[\ell]$ stores x_h (as a "proxy" for h), and
- if ℓ is white and neither a historian nor a proxy whose historian is black, the value of $A[\ell]$ may be arbitrary.

Note that because every proxy is white, for each $\ell \in \{0, \ldots, n-1\}$ exactly one of the four cases above applies. In particular, although a proxy may coincide with its historian, this is not the case if the historian is black. The data structure is initialized by coloring r white (i.e., by setting the root bits accordingly).

In terms of the abstract array a, the leaf colors white and black signify "not yet written to, and therefore still containing the initial value 0" and "written to at least once", respectively. For the actual array A, this translates approximately into white and black meaning "not initialized" and "initialized to a meaningful value", respectively.

The data structure does not explicitly store the color of any node except r. Instead node colors must be deduced from histories. It turns out that the colors of all nodes other than r are implied by the histories of the light paths. A white leaf ℓ offers potential for storing a history (namely in its associated word $A[\ell]$), but we cannot know in advance where to find a white leaf. This motivates the introduction of historians and proxies. We actually need the history of a light path P when, during a descent in T from r to a leaf, we reach

the top node of P. The historian of P provides a fixed place (namely at the leftmost leaf descendant) at which to look for the history, but if the historian is black, then its own data must be accommodated somewhere else – this is the role of the (white) proxy. How this works is perhaps best illustrated by the following detailed description of the realization of the operation *read*, which basically carries out a descent in T. The call *leftmostleaf*(u) is assumed to return (the integer identified with) the leftmost leaf descendant of the node u.

read(ℓ):
  $u := r$; $(*$ start at the root $*)$
  while u is gray **do**
    if u is a top node **then** $(*$ switch to a new history $*)$
      $h :=$ *leftmostleaf*(u); $(*$ u's historian $*)$
      $H := A[h]$; $(*$ u's history $*)$
    $u :=$ *viachild*(u, ℓ); $(*$ continue towards ℓ $*)$
  if u is white **then return** 0; $(*$ the initial value $*)$
  $(*$ now ℓ is black $*)$
  if $u = r$ or $\ell \neq h$ **then return** $A[\ell]$; $(*$ ℓ is neither a historian nor a proxy $*)$
  $(*$ now ℓ is a black historian $*)$
  return $A[p]$, where p is the leaf at the end of the light path that contains u's parent;

The procedure discovers a white ancestor of ℓ and returns 0, determines that ℓ is black and not a historian and returns $A[\ell]$, or identifies ℓ as a black historian and returns $A[p]$, where p is the proxy of ℓ. In all cases, the return value is correct.

Whenever the color of a node u is queried, either $u = r$, in which case the color of u is given by the root bits, or the color of u can be deduced in constant time from the history stored in H, one of whose elements is the navigation vector of the parent of u. Similarly, if $u \neq r$, we can decide in constant time whether u is a top node by looking at the navigation vector of its parent. The light path that contains u's parent can be followed in constant time per node, again by inspection of H. Thus *read* can be executed in $O(t)$ time.

To execute *write*(ℓ, x), we carry out two phases. The purpose of the first phase is to take the data structure to a legal state in which ℓ is black and all values of the client sequence $(x_0, \ldots, x_{n-1})$ except possibly x_ℓ are correct, i.e., unchanged. The second phase concludes the writing by setting x_ℓ to x. In the description of the two phases, we leave to the reader details such as how to determine the color of a given node; in all cases, one can proceed similarly as in the implementation of *read*.

The first phase begins by following the path P in T from r to ℓ until encountering a node that is not gray. This can be done similarly as in the implementation of *read*: Each node visited is tested for being a top node, and at each top node a new history is fetched and subsequently used. This computation, in particular, can determine the color of ℓ. If ℓ is already black, the first phase terminates without modifying the data structure. Assume in the remaining discussion of the first phase that ℓ is white and consider the consequences of an *update* that changes the color of ℓ from white to black. We will use the terms "old" and "new" to describe the situation before and after the update, respectively.

Because the color of an inner node in T is a function of the colors of its children, only nodes on P can change their color as a result of the update. The first phase proceeds to find the first node v on P (i.e., the node on P of minimal depth) that changes its color. The following observations show that this can be done in a single traversal of P and characterizes the possible scenarios in a useful way. If some proper ancestor of ℓ is white (before the update), all proper ancestors of the first white node $\widetilde{v}$ on P are gray both before and after

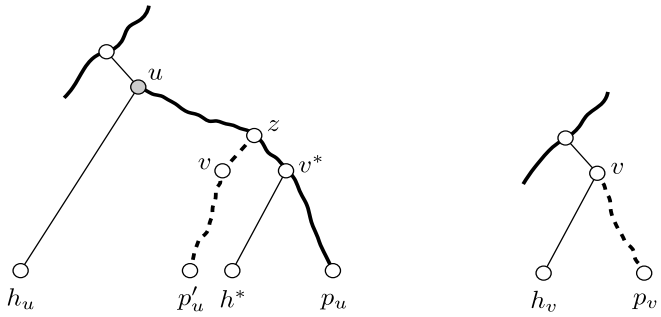

Figure 2 Left: The situation of Case 1 and, after a swap of the labels "p_u" and "p'_u", also of Case 2. Right: The situation of Cases 4 and 5.

the update, and all descendants of $\widetilde{v}$ on P other than ℓ are white before and gray after the update. Thus $v = \widetilde{v}$. In the opposite case, namely if all proper ancestors of ℓ are gray, let $\overline{v}$ be the last node on P that has a white or gray sibling if there is at least one such node, and take $\overline{v} = r$ otherwise. It is easy to see that all proper ancestors of $\overline{v}$ are gray both before and after the update and, by backwards induction on P, that all descendants of $\overline{v}$, including $\overline{v}$ itself, are black after the update. In this case, therefore, $v = \overline{v}$.

As can be seen from the observations above, no descendant of v has more than one gray child before or after the update under consideration. Therefore the only node in T_v that can be a top node before or after the update is v itself, the only node in T_v that can be a historian before or after the update is the leftmost leaf descendant h_v of v, and before as well as after the update at most one node in T_v is a proxy. Moreover, at most one node in T other than v can become or stop being a top node as a result of the update, and this node, if it exists, must be the leftmost gray sibling of v and to the right of v.

If $v = r$, change the root bits to reflect the new color of the root. Otherwise compute u as the top node of the light path that contains the (gray) parent of v, let h_u be the historian of u (before and after the update) and let p_u and p'_u be the proxies of u before and after the update, respectively, which can be found by following the old and new light paths that start at u. Store the new history of p'_u in $A[h_u]$. In particular, this registers the new color of v. To compute the history, it suffices to record the new navigation vectors encountered on the path in T from u to p'_u. Now consider five cases that together cover all possible situations and do not overlap. Even though every color change is irreversible, Cases 1 and 2 show some aspects of being reverses of each other, and so do Cases 4 and 5. These four cases are illustrated in Fig. 2.

Case 1*:* v has a parent z and is the preferred child of z after the update. Thus v changes its color from white to gray without becoming a top node. If h_u is black before the update, then execute $A[p'_u] := A[p_u]$, which moves x_{h_u} from the old to the new proxy of h_u. This overwrites no relevant information, as p'_u is white and neither a historian nor a proxy before the update unless p'_u coincides with h_u or p_u, in which case the assignment is not carried out or has no effect. Let v^* be the preferred child of z before the update and let h^* be the leftmost leaf descendant of v^*. If v^* is white before the update (this includes the case $v^* = v$), nothing more needs to be done. If v^* is gray (before and after the update), it is a right sibling of v, and it becomes a new top node whose historian h^* and proxy p_u must have their associated information updated accordingly. To this end first execute $A[p_u] := A[h^*]$ and subsequently store in $A[h^*]$ the history of the new light path that starts at v^* and ends at p_u. If $p_u = h^*$, the two assignments write to the same word, but then any relevant information present in $A[h^*]$ before the update was already copied to $A[p'_u]$.

Case 2*:* v has a parent z and is the preferred child of z before, but not after the update. After the update, v is black and no descendant of v is a historian or a proxy, except that h_v may coincide with h_u. Let v^* be the preferred child of z after the update and let h^* be the leftmost leaf descendant of v^*. If v^* is gray, it is a right sibling of v and a top node with historian h^* and proxy p'_u before the update, whereas after the update p'_u is the proxy of u and h^* is neither a historian nor a proxy unless $h^* = p'_u$. If h^* is black, then execute $A[h^*] := A[p'_u]$, which moves x_{h^*} to the correct place and overwrites a history that is no longer useful. Finally, independently of the color of v^* and as in Case 1, if h_u is black, then execute $A[p'_u] := A[p_u]$.

In the remaining cases 3–5 v is a preferred child neither before nor after the update, so there are no changes to light paths outside of T_v (i.e., the set of light edges outside of T_v remains the same). In particular, $p'_u = p_u$. Moreover, v is not a leftmost child.

Case 3*:* v is a leaf with at least one white left sibling. There are no changes to light paths, so nothing needs to be done.

Case 4*:* v is a top node after the update. Before the update, v is white, so no descendant of v is a historian or a proxy at that time (informally, no information is stored below v). Compute the proxy p_v of v after the update and store the new history of p_v in $A[h_v]$. This involves following the new light path that starts at v and recording the new navigation vectors encountered on the way.

Case 5*:* v is a top node before the update. Because v is black after the update, no descendant of v is a historian or a proxy at that time. Before the update, since ℓ is the only white descendant of v, it is its proxy. If h_v is black (i.e., if $h_v \neq \ell$), then copy the value of $A[\ell]$, namely x_{h_v}, to $A[h_v]$. This overwrites an old history that is no longer useful.

The second phase of the execution of *write*(ℓ, x) simulates the execution of *read*(ℓ) until the point when the routine is ready to return as its answer the value of $A[i]$ for some i (that is either ℓ or the proxy of ℓ). Instead of returning $A[i]$, it finishes by storing x in $A[i]$. Since i is not a historian, it is easy to see that a subsequent call of *read*(ℓ) will return x and that the update of $A[i]$ leaves the data structure in a legal state and does not change the value of any elements of the client sequence $(x_0, \ldots, x_{n-1})$ except x_ℓ. ◀

The next lemma and its proof show how to handle the case of an "incomplete tree" elegantly and, following the initialization, without any overhead to test for special cases.

▶ **Lemma 5.** *There is a clearable word array that, for all given* $n, d, t \in \mathbb{N}$ *with* $2dt \leq w$ *and* $n \leq d^t$ *such that* d *is a power of* 2*, can be initialized for universe size* n *in constant time and subsequently occupies* $nw + 2$ *bits and, if given access to* d *and* t*, supports read and write in* $O(t)$ *time.*

Proof. We use the construction of the previous proof for universe size d^t, but provide for its storage only a word array A with index set $\{0, \ldots, n-1\}$ in addition to two root bits. If $n = d^t$, nothing more needs to be said. If $n < d^t$, before executing any true *write* operation, we change the color of the root from white to gray (of course, by modifying the root bits) and store in $A[0]$ a history that corresponds to the leaves $0, \ldots, n-1$ being white and $n, \ldots, d^t - 1$ being black. Provided that only legal accesses are subsequently attempted, this prevents the data structure from ever choosing a proxy larger than $n-1$, and it will process the operations correctly without ever attempting to access one of the nonexisting array elements $A[n], \ldots, A[d^t - 1]$.

The computational steps just described are conceptually part of the initialization of the data structure, but the computation of the history to be stored in $A[0]$ may take more than constant time. In order to guarantee a constant initialization time, we postpone the steps

and execute them as an initial part of the first and only execution of a *write* operation that begins with a white root, until which point we remember n in $A[0]$. Since the steps are easily carried out in $O(t)$ time, the bound of $O(t)$ for the execution time of *write* remains valid. ◀

We now take the step to values of n larger than d^t.

▶ **Lemma 6.** *There is a clearable word array that, for all given $n, t \in \mathbb{N}$, can be initialized for universe size n in constant time and subsequently occupies at most $nw + \lceil n(t/(2w))^t \rceil$ bits and, if given access to n and t, supports read and write in $O(t)$ time.*

Proof. When $c \in \mathbb{N}$ is an arbitrary constant, we can assume without loss of generality that n is a multiple of c. This is because we can initialize up to $c-1$ "left-over" words in constant time. Moreover, a word RAM with a word length of w bits can simulate one with a word length of cw bits with constant slowdown, i.e., every instruction can be simulated in constant time. By these observations, we can essentially pretend to be working on a word RAM with a word length of cw bits (of course, the values communicated to and from a user of the data structure are still w-bit quantities). In particular, we view A as consisting of n/c *large words* of cw bits each, and the condition $2dt \le w$ of Lemma 5 can be relaxed to $2dt \le cw$. We use this with $c = 16$, for which choice the condition becomes $d \le 8w/t$.

Assume that $t \le w$. This entails no loss of generality because reducing larger values of t to w does not increase the space bound of the lemma (recall that $w \ge \lceil \log_2 n \rceil$). Compute d as the largest power of 2 no larger than $8w/t$ and note that $d \ge 4w/t \ge 2$. Dividing the universe $\{0, \ldots, n/c - 1\}$ into *ranges* of d^t consecutive elements each, except that the last range may be smaller, we store each subsequence of the client sequence corresponding to a range in an instance of the data structure of Lemma 5, called a *tree*, except that the root bits are handled slightly differently. Altogether we have $N = \lceil n/(cd^t) \rceil \le \lceil n(t/(2w))^t \rceil$ trees.

If $N = 1$, i.e., if there is only a single tree, we use a single root bit to distinguish between black and nonblack (i.e., white or gray). In order to indicate a white root, in addition to initializing the root bit to the value that denotes a nonblack color, we store in $A[0]$ a value that cannot be the history of a gray root, such as one in which all colors in the navigation vector of the root are white. The total redundancy is $1 = N \le \lceil n(t/(2w))^t \rceil$.

If $N > 1$, we solve the problem of initializing the N trees differently. Each tree has two root bits, and we must set these to indicate a white root. Assume, for convenience, that the root color white is represented through two bits with a value of zero. Then the task is to clear the $2N$ root bits, i.e., to set them to zero. Pack the $2N$ root bits tightly in $M = \lfloor 2N/w \rfloor$ fully occupied words and at most one partially occupied word. If there is an only partially occupied word, clear it explicitly. As for the M fully occupied words, maintain these, if $M \ge 1$, in a clearable word array implemented with the folklore method discussed near the end of Section 2. In addition to the M words, this needs space for two tables with altogether $2M$ entries and one counter that takes values in $\{0, \ldots, M\}$. Each table entry fits in a w-bit word, and except in the trivial case $w = 1$, the counter can be stored in N bits, so the redundancy is at most $3Mw + N \le 7N$. Since $N > 1$, we even have $8N \le 16(n/c)(t/(4w))^t = n(t/(4w))^t \le \lceil n(t/(2w))^t \rceil$. This slightly stronger bound is irrelevant here, but useful in a proof of Theorem 2. ◀

In order to derive Theorem 2 from Lemma 6 and its proof, we show in [11] how to "hide" the parameters n and t in the data structure essentially without additional space or how to make do without them.

It is interesting to note that we can add an additional operation to our clearable word array, namely an iteration that enumerates all first arguments of past *write* operations

(informally, the positions to which writing took place). For this we would iterate over the codes handed out by the folklore method and the associated trees, which is easy, enumerate all leaves of each tree whose root is black, and for each tree whose root is gray carry out a depth-first search (say) of its gray nodes and enumerate all leaf descendants of their black children. The time needed is proportional to the number k of leaves enumerated plus the total number of gray nodes, a quantity that is clearly bounded by $(t+1)k$ and never larger than $2n$. The iteration must be called with an argument that indicates n.

References

1 Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

2 Andrei Alexandrescu. *The D Programming Language*. Addison-Wesley, 2010.

3 D. Angluin and L. G. Valiant. Fast probabilistic algorithms for Hamiltonian circuits and matchings. *J. Comput. Syst. Sci.*, 18(2):155–193, 1979. doi:10.1016/0022-0000(79)90045-X.

4 Amos Fiat, J. Ian Munro, Moni Naor, Alejandro A. Schäffer, Jeanette P. Schmidt, and Alan Siegel. An implicit data structure for searching a multikey table in logarithmic time. *J. Comput. Syst. Sci.*, 43(3):406–424, 1991. doi:10.1016/0022-0000(91)90022-W.

5 Gianni Franceschini and Roberto Grossi. No sorting? Better searching! *ACM Trans. Algorithms*, 4(1):2:1–2:13, 2008. doi:10.1145/1328911.1328913.

6 Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993. doi:10.1016/0022-0000(93)90040-4.

7 Kimmo Fredriksson and Pekka Kilpeläinen. Practically efficient array initialization. *J. Softw. Pract. Exper.*, 46(4):435–467, 2016. doi:10.1002/spe.2314.

8 James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. Oracle America, 2015.

9 Torben Hagerup. Sorting and searching on the word RAM. In *Proc. 15th Annual Symposium on Theoretical Aspects of Computer Science (STACS 1998)*, volume 1373 of *LNCS*, pages 366–398. Springer, 1998. doi:10.1007/BFb0028575.

10 Torben Hagerup and Frank Kammer. Succinct choice dictionaries. *Computing Research Repository (CoRR)*, arXiv:1604.06058 [cs.DS], 2016. arXiv:1604.06058.

11 Torben Hagerup and Frank Kammer. On-the-fly array initialization in less space. *Computing Research Repository (CoRR)*, arXiv:1709.10477 [cs.DS], 2017. arXiv:1709.10477.

12 IEC/IEEE International Standard; Behavioural languages — Part 1–1: VHDL Language Reference Manual. IEC 61691–1–1:2011(E) IEEE Std 1076-2008, 2011. doi:10.1109/IEEESTD.2011.5967868.

13 Takashi Katoh and Keisuke Goto. In-place initializable arrays. *Computing Research Repository (CoRR)*, arXiv:1709.08900 [cs.DS], 2017. arXiv:1709.08900.

14 J. Ian Munro. An implicit data structure supporting insertion, deletion, and search in $O(\log^2 n)$ time. *J. Comput. Syst. Sci.*, 33(1):66–74, 1986. doi:10.1016/0022-0000(86)90043-7.

15 Gonzalo Navarro. Spaces, trees, and colors: The algorithmic landscape of document retrieval on sequences. *ACM Comput. Surv.*, 46(4):52:1–52:47, 2014. doi:10.1145/2535933.