Dynamic Dictionaries in Constant Worst-Case Time

Gonzalo Navarro

Dept. of Computer Science, Universidad de Chile, Chile.
gnavarro@dcc.uchile.cl

Abstract

We introduce a technique to maintain a set of n elements from a universe of size u with membership and indel operations, so that elements are associated r-bit satellite data. We achieve constant worst-case time for all the operations, at the price of spending $u + o(u) + O(nr + n \log\log\log u)$ bits of space. Only the variant where the space is of the form $O(nr + n \log u)$ was exhaustively explored before, yet in that case existing lower bounds prevent achieving constant worst-case times. As a byproduct, we improve a folklore data structure for initializing an array of n elements in constant time, by reducing its space requirement from $2n \log n$ to $n+o(n)$ bits.

Key words: Algorithms and data structures, succinct data structures, dynamic perfect hashing, dynamic dictionaries with satellite information.

1 Introduction and Related Work

One of the most basic algorithmic problems is that of maintaining a set of $(key, value)$ pairs, so as to retrieve the value associated to a key (or determine that the key is not present in the set), and be able of inserting and deleting pairs in/from the set (those two operations are collectively called indels or updates). In the literature this problem receives different names, such as dynamic perfect hashing, dynamic dictionary with satellite information, and dynamic dictionary with retrieval. In the absence of a consistent notation, in this paper we called this the *dynamic dictionary problem*, and distinguish it from the *dynamic membership problem* where only keys are stored. There is

⋆ Supported by a grant from Yahoo! Research Latin America

a huge literature on the subject, so here we only discuss the most recent and closest results.

To describe the state of the art, let us introduce some notation. We assume that the universe of key values is $[u] = \{1, 2, \ldots, u\}$, that the keys are unique, that the values require $r = O(\log u)$ bits to be represented [1], and that there are (currently) n elements in the set. We work on a RAM machine with word size $w = \Omega(\log u)$, measure the space in bits, and write log for $\log_2$. The operations permitted are called *retrieve*(*key*) (which retrieves the associated *value* or $\perp$ if the key is not in the set), *insert*(*key*, *value*), and *delete*(*key*). The dynamic membership problem considers query *member*(*key*) instead of *retrieve*, which returns a boolean value telling whether or not the key is present in the set.

Most of the research on the problem has focused on using space of the form $O(n \log u + nr)$ bits. Indeed, a good part of the research focuses on membership rather than retrieval queries. Existing lower bounds [3] establish that it is impossible to achieve constant worst-case time for membership and indels simultaneously using $O(n \log u)$ bits of space. Thus, the research has focused on randomized algorithms. The best results up to date, for the retrieval problem, are as follows. In [9], they use $(n \log \frac{u}{n} + nr)(1 + o(1))$ bits of space, which is within lower-order terms from the information-theoretic lower bound, answer queries in constant worst-case time and indels in constant expected amortized time. In [2] they use $O(n \log \frac{u}{n} + nr)$ bits of space, answer queries in constant time, and perform indels in constant time with high probability. In both cases r is arbitrary but the "constant time" ignores the $O(r/w)$ time needed to read/write the r data bits. Older references can be found within [9,2].

The variant of the problem where we are willing to spend u bits of extra space has received less attention, possibly because the membership plus indels problem becomes trivial in this case. However, if we wish to maintain satellite information as well, it is not immediate that retrieval and updates can be done in constant worst-case time. Only if we spend $u(r + 1)$ bits is the problem trivial.

In this paper, we explore what can be achieved in the worst case if we are allowed to spend u bits of space. We show that, by spending $O(n(r + \log\log\log u))$ $+u + o(u)$ bits, we can carry out the three operations in *worst-case* constant time (see the exact details in Theorem 4). The update times become amortized if the space is reduced to $nr + u + o(u) + O(n \log\log\log u)$ bits. This result contributes to understand the space/time tradeoff for dynamic dictionaries, showing that worst-case constant time is possible for this price.

As a byproduct, we obtain a succinct version of a folklore technique to initialize

[1] Our results apply to larger r values but the formulas are messier. We will anyway mention the more general results when appropriate, without giving all the details.

an array in constant time. Explicitly initializing a huge array might not pay off if we are not going to carry out many operations on it. This situation arises, for example, when using a (classical) hashing scheme and one has not sufficient knowledge of the amount of insertions that will occur. The folklore solution has the severe drawback of requiring $\Theta(n \log n)$ bits of extra space, on top of the array of n elements to initialize. We show that this space can be reduced to $n + o(n)$ bits, while retaining the same time complexities. This is necessary to obtain our main results in the paper, so we start with it and then move on to dynamic dictionaries.

2 Constant-Time Array Initialization in Little Space

A folklore solution, well-known at least since the seventies, permits initializing an array $A[1, n]$ in constant time (see [1, Ex. 2.12, page 71], or a complete description in [6, Section III.8.1]).

Definition 1 *An* initializable array *is a data structure $A[1, n]$ that supports the operations $init(A, n, v)$, $read(A, i)$ and $write(A, i, v)$. The first operation initializes $A[i] \leftarrow v$ for $1 \leq i \leq n$; the second obtains $A[i]$ and the third sets $A[i] \leftarrow v$. The size n is fixed at initialization time.*

Theorem 1 *(folklore). An array $A[1, n]$ can be enhanced with structures using $2n\lceil \log n \rceil$ additional bits of space, so that all its operations take constant time.*

This solution has the serious drawback of requiring too much extra space. For example, the space is tripled if A stores indexes in $[n]$.

In this section we largely reduce the extra memory required to initialize A in constant time. More precisely, we obtain the following result.

Theorem 2 *On a RAM machine with word size $w = \Omega(\log n)$, an array $A[1, n]$ can be enhanced with structures using $O(n)$ bits of extra space, so that all its operations init, read and write, take constant time. The $O(n)$ factor can be made $\left(\sum_{0 \leq d < h} n / \log^d n\right) + 3n / \log^h n$ for any constant $h \geq 0$. This is $n + o(n)$ already for $h = 1$.*

This result is interesting both from theoretical and practical standpoints, and can make the solution much more appealing in practical scenarios, especially if we consider that this problem arises when we do not want to afford the cost of initializing a large array. The extra space may make the difference between having the array in main memory or on disk, for example.

2.1 The Original Technique

The original folklore technique is as follows. Let $A[1,n]$ be the array we wish to initialize in constant time. We use a second array $B[1,n]$ and a stack $S[1,n]$, both storing indices in $[n]$. An additional variable $0 \le t \le n$ tells the current size of S, and variable V stores the initialization value.

Initialization of the whole structure, $init(A,n,v)$, consists of setting $t \leftarrow 0$ and $V \leftarrow v$. The invariant maintained by the structures is as follows:

$$A[i] \text{ is initialized} \iff (1 \le B[i] \le t \ \wedge \ S[B[i]] = i)\,,$$

which is immediately correct once we set $t = 0$. The idea is to distinguish initialized entries $A[i]$ because $B[i] = j$ and there is a back pointer $S[j] = i$. Let us take an uninitialized entry $A[i]$. Value $B[i]$ is not initialized either. If $B[i] < 1$ or $B[i] > t$, we know for sure that $A[i]$ is not initialized. Yet it could be that $1 \le B[i] \le t$. But then it is not possible that $S[B[i]] = i$ because entry $B[i]$ in S has been used to initialize another entry $A[i']$ and then $S[B[i]] = i' \ne i$.

Operation $read(A,i)$ is as follows. If $A[i]$ is initialized, then it returns $A[i]$, otherwise it returns V. Operation $write(A,i,v)$ is as follows. If $A[i]$ is not initialized, it first sets $t \leftarrow t+1$, $B[i] \leftarrow t$, and $S[t] \leftarrow i$. After the possible initialization, it sets $A[i] \leftarrow v$. It is interesting that one can even uninitialize $A[i]$, by setting $S[B[i]] \leftarrow S[t]$, $B[S[t]] \leftarrow B[i]$, $t \leftarrow t-1$.

2.2 Reducing Space

We use, instead of array B and stack S, a bit vector $C[1,n]$ so that $C[i] = 1$ iff $A[i]$ has been initialized. This way we require only n bits in addition to A. Of course the problem translates into initializing $C[i] \leftarrow 0$ for all i. We now take advantage of the unit-cost RAM model of computation, where $w \ge \log n$ must hold because we store numbers up to n in computer words.

As C is stored as a contiguous sequence of bits, let us interpret this sequence as an array $C'[1,n']$ of $n' = \lceil n/w \rceil$ entries, each entry holding a computer word of w bits of C. We can apply now the original solution of Section 2.1 to C', so that C' can be initialized in constant time (at value $C'[i] = 0$). The extra space on top of C' is $2n' \log n' \le 2n$ bits. Together with C', the space overhead of the solution is $3n$ bits. Now, in order to determine whether $A[i]$ is initialized, we just check $C[i]$: We compute $q = i \text{ div } w$ and $r = i \bmod w$ and check the bit number $r+1$ of $C'[q+1]$. If $C'[q+1]$ is not yet initialized, we know $C[i] = 0$ and thus $A[i]$ is not yet initialized. To initialize $A[i]$ we set

the $(r+1)$-th bit of $C'[q+1]$, previously initializing $C'[q+1] \leftarrow 0$ if needed.

This can be carried further. Instead of directly applying the solution of Section 2.1 to C', we could recursively use a bitmap of n' bits telling which entries of C' are initialized, compact them into a second array $C''[1,n'']$, $n'' = \lceil n'/w \rceil \approx n/w^2$, and then apply the solution of Section 2.1 to C''. This time the space overhead is $n + 3n/\log n$ bits. This can be repeated a constant number of times, to achieve any extra space of the form $\left(\sum_{0 \le d < h} n/\log^d n\right) + 3n/\log^h n$. The price in practice is that the time to access A grows linearly with h.

3 Extendible Arrays and Memory Allocation Model

A critical issue in dynamic data structures, which is often disregarded, is the memory allocation model. On a static data structure one may assume that all the data is conveniently packed in a memory area, but dynamic data structures that allocate and free chunks of memory may suffer from fragmentation, which has to be accounted for. Moreover, relying on system memory allocation may hide non-constant-time algorithms for allocation or freeing of memory chunks.

In [9], they consider two memory models called $\mathcal{M}_A$ and $\mathcal{M}_B$. In this paper we stick to $\mathcal{M}_B$ because it is the standard on the RAM model and it assumes the least from the system. In $\mathcal{M}_B$ there are no system calls for allocation and deallocation of memory, but the program must handle memory by itself. The memory is seen as an array of words of w bits, numbered 0 to $2^w - 1$. The amount of memory used by a program at a given moment is the length of the shortest prefix of the memory array that contains all the data currently allocated by the program.

A problem that is surprisingly difficult when one considers the details of memory allocation is the so-called *extendible array (EA)* problem, where we maintain an array of fixed-width cells and want to access it at random positions, as well as inserting and deleting cells at the end of the array. Maintaining a single EA is trivial even under model $\mathcal{M}_B$, but maintaining a *collection* of such arrays is not trivial anymore. More precisely, this problem is defined as follows.

Definition 2 *The* EA collection problem *is to maintain a collection of EAs, each of which can be created empty at some moment, destroyed, inspected at an arbitrary position, be grown by one position at the end, and shrink by one position at the end. Each EA maintains cells of fixed width r_i bits, possibly different from that of other EAs in the collection. If n_i is the current number of elements of each such EA, then the* nominal size *of the collection is $s = \sum n_i r_i$.*

A key result for EAs is as follows:

Theorem 3 *(simplified from Lemma 1 in [9]). A collection of a EAs of nominal size s bits can be represented using $s+O(aw+\sqrt{saw})$ bits of space, so that the operations of creation of an empty EA and access take constant worst-case time, whereas grow/shrink take constant amortized time. An EA of s' nominal bits can be destroyed in time $O(s'/w)$.*

In this paper the number of EAs a never decreases (we may delete an EA but immediately replace it). Thus, the current number of EAs a is also the largest number of EAs that ever existed in the collection.

The idea of the proof is to split the data of each EA (ignoring cell boundaries) into records of fixed size $\Theta(\sqrt{(s/a)w})$ bits (the records are resized when s or a double or halve). The records are stored contiguously, yet those of a single EA do not need to be contiguous nor ordered in the sequence. Since almost one such record can be wasted per EA, we waste $O(\sqrt{saw})$ bits overall. The records are managed in a *directory* spending one pointer (w bits) per record. As there are at most $\frac{s}{\sqrt{(s/a)w}} + a = \sqrt{sa/w} + a$ records, spending one pointer for each gives a directory size of $O(\sqrt{saw})$ bits. Additional $O(aw)$ bits are spent in mapping EA names to internal addresses within the directory. Those are also spent if the record sizes are smaller than w, in which case we have to make them of w bits to ensure that the r_i bits of a cell can be read/written in $O(r_i/w)$ time on the RAM machine [2]. In this case the space wasted in empty records also amounts to $O(aw)$. Indeed, any $O(w)$-bit sequence from the array can be read/written in constant time.

Note that it would be possible to simulate a different w when implementing an EA. If a^* and s^* are upper bounds to, respectively, the total number EAs and total number of bits we will ever have in the collection, we could use pointers of $w' = \Theta(\log(s^*a^*))$ bits within the collection, as they suffice to address the (at most) $\sqrt{s^*a^*/w + a^*}$ different records. Reducing w to w' makes the record size $\Theta(\sqrt{(s/a)w'})$ bits, and the wasted space across all the records $O(\sqrt{saw'})$ bits. The pointers in the directory are also of w' bits, so the directory needs $O(\sqrt{saw'})$ bits as well. The only detail is that now the time to read/write r bits of data is $O(r/w')$. If we wish to maintain this as $O(r/w)$, we must ensure that records are at least w bits long, which costs $O(aw)$ extra bits of wasted space within records.

Finally, we note that, under memory model $\mathcal{M}_B$, the space reported by the EA collection refers to the rightmost position of an active record. Moreover, the only memory management the EA collection needs are access, grow to

[2] This time is not accounted for in the complexities of the theorem.

the right, and shrink from the right. The directory is placed first and then a variable number of records are laid in compact form. This means that we could allocate an EA collection *within a larger EA* (this is indeed implied in Proposition 1 of [9]). The elements of the larger EA correspond to the records used to implement the EA collection ($\Theta(\sqrt{(s/a)w'})$ bits), and the directory is artificially split into records of the same size too. When the directory size changes the larger EA must be totally rewritten. This can be done in time proportional to the collection size by copying the larger EA onto a newly created one and deleting the old large EA.

Finally, the amortized time can be converted into worst case if one spends $O(s)$ extra space. The result of [9] builds over a technique given in [5], which maintains a collection of EAs with w-bit records in worst-case constant time per operation. Its limitations are that the address of the EAs may change upon operations, and that they need $O(nw)$ bits where n is *an a-priori upper bound* on the total number of elements in the collection. In [9], the technique is used to manage the dictionary of the collection, which must be rebuilt when s or a double or halve. At this point the entire collection can be rewritten to maintain the record size $\Theta(\sqrt{saw})$. As the only reason for the costs to be amortized instead of worst-case is the need to rewrite everything at those points, one can easily deamortize by usual means, that is, maintaining copies of the data structure corresponding to the next and previous value of $\lceil \log s \rceil$ or $\lceil \log a \rceil$ [8].

Corollary 1 *A collection of a (maximum a^*) EAs of nominal size s (maximum s^*) bits can be represented within a larger EA using $s + O(a \log(s^* a^*) + \sqrt{sa \log(s^* a^*)})$ bits of space, so that the operations of creation of an empty EA in the collection and accessing any $O(\log(s^* a^*))$ contiguous bits of an EA take constant worst-case time, whereas growing and shrinking an EA take constant amortized time. The amortized times can become fully worst-case by spending $O(s + a \log(s^* a^*) + \sqrt{sa \log(s^* a^*)}) = O(s + a \log(s^* a^*))$ bits of space.*

4 Dynamic Dictionaries: A First Approach

We extend the constant-time array initialization technique to achieve a dynamic dictionary implementation that requires $O(nr + u \log \log u)$ bits of space and constant worst-case time for all operations. Alternatively, we achieve $O(nr + u + n \log \log u)$ bits of space and $O(\log \log u)$ time for the operations. This is still not fully satisfactory, but forms the basis of our definitive solution depicted in the next section.

Definition 3 *Let $\mathcal{S} \subseteq [u]$ be a set of $|\mathcal{S}| = n$ elements, to which we associate*

r bits of satellite data. The dynamic dictionary problem *is to create $\mathcal{S} = \emptyset$ and then maintain $\mathcal{S}$ upon retrieval queries (which return the r bits associated to given index in $[u]$ if it is in $\mathcal{S}$, or $\perp$ otherwise), and insertions and deletions in $\mathcal{S}$. The size of the machine word is $w = \Omega(\log u)$. It is also usually assumed that $r = O(\log u)$, although we consider up to to the much more liberal $r = O(\text{poly}(u))$.*

A trivial implementation supporting retrieval and indels in $\mathcal{S}$ would be an array $A[1, u]$, so that $A[i]$ contained the r bits associated to the i-th element of the universe, plus a bit array $U[1, u]$ so that $U[i]$ tells whether $i \in \mathcal{S}$. This permits easily carrying out all the operations in constant time, but it takes $u(r + 1)$ bits of space, which might be too much. Actually, if one stores only keys, this reduces to the trivial solution to the dynamic membership problem using u bits of space.

In a first approach, one could store the data in a *compacted* array $A'[1, n]$, so that the data associated to $U[i] = 1$ is stored in $A'[rank(U, 1, i)]$, being $rank(U, x, y)$ the number of 1's in $U[x, y]$. This, however, would require solving the so-called *dynamic array problem* (insert, delete, and access cells anywhere in A'), which cannot be achieved in constant time [4], as well as solving the dynamic *rank* problem, which is also impossible in constant time [7].

A solution inspired in Section 2.2 is to store the nonempty values of A in extendible arrays (EAs, Section 3) associated to the stack cells of Section 2.2 (thus we will be able to create the set in constant time, even if our data structures require at least u bits). Array A will be divided into *chunks* of c entries, and for the nonempty chunks we will store the entries in compact form. In order to manipulate them in constant time, we will store the compacted cells in any order, so a permutation will be used to handle the ordering. More precisely, our data structures are as follows.

(1) We store the bit array $U[1, u]$ so that $U[i] = 1$ iff $i \in \mathcal{S}$.
(2) We logically divide A and U into chunks of c entries.
(3) We store arrays $S[1, \lceil u/c \rceil]$ and $S'[1, \lceil u/c \rceil]$. S uses all its entries, so that $S[i]$ corresponds to the entries $[(i - 1)c + 1, ic]$ of A and U. If all those entries are zero in U, then $S[i]$ is undefined, else $S[i] = j$ points to $S'[j] = i$. Therefore S' plays the role of the stack and S plays the role of the auxiliary array of Section 2.2. U does not need initialization to zero because we can know where it is uninitialized by using S and S'. To simplify matters we reserve the maximum possible space for S', although we could do better (yet not asymptotically). We nevertheless need to maintain the real size of S' in a variable nS'.
(4) We store, aligned to the array cells $S'[j]$, $1 \leq j \leq nS'$, EAs P_j and A_j. All those EAs are stored as a single collection of EAs.
   (a) $P_j[1, k]$ will be a permutation of $[k]$, where $0 \leq k \leq c$ is the number

of nonempty cells in the corresponding chunk of A.

(b) $A_j[1,k]$ will be an array of r-bit registers whose correspondence with the associated chunk in A will be given by P_j.

(5) We also store numbers nE_j to record the length of extendible arrays P_j and A_j (that is, the numbers k above).

We now describe the operations over this data structure. In the description, we use the following subroutines.

- $U[key]$ *is initialized*, or $S[i]$ *is initialized*, where $i = \lceil key/c \rceil$, means that $1 \leq S[i] \leq nS'$ and $S'[S[i]] = i$.
- *Initializing* $S[i]$ means incrementing $nS' \leftarrow nS' + 1$, creating a new cell in $S'[nS'] \leftarrow i$ and $S[i] \leftarrow nS'$, creating new empty EAs $P_{nS'}$ and $A_{nS'}$, setting $nE_{nS'} \leftarrow 0$, and finally setting $U[(i-1)c+1, ic] \leftarrow 0$. This last assignment can be done in time $O(c/w)$ on the RAM model.
- $rank(U, x, y)$ is computed in time $O(\frac{c}{\log u})$ via Four-Russians techniques: Use a universal table of $\sqrt{u}$ entries telling the number of bits set for every possible sequence of $\frac{1}{2}\log u$ bits. Such a table requires $O(\sqrt{u}\log u)$ bits of space. If $|y - x + 1|$ is not a multiple of $\frac{1}{2}\log u$, we compute the rank of the remainder bits by isolating them from the sequence and padding with zeros. This can be done in constant time if the RAM model allows shifting or multiplication/division, otherwise another Four Russians table would be needed to set to zero any prefix/suffix of a sequence of $\frac{1}{2}\log u$ bits. This second table would require $O(\sqrt{u}\log^2 u)$ bits of space.
- *Shifting cells in* P_j can be done in time $O(\frac{c\log c}{w})$ if the RAM model allows shifting or multiplication/division, otherwise a simple Four-Russians scheme achieves time $O(\frac{c\log c}{\log u})$.
- *Searching for a cell value* p *in* P_j, for a given p, can also be done in time $O(\frac{c\log c}{\log u})$. A Four-Russians table indexed by (i) a bit sequence of the maximum length[3] $k\log c \leq \frac{1}{2}\log u$ and (ii) the value p, stores the position of value p within the sequence, or zero if p is not within that sequence. This table requires $O(\sqrt{u}\log u \log\log u)$ bits.

The operations over this data structure are carried out as follows.

Create: Set $nS' \leftarrow 0$.

Retrieve(key)**:** Let $i \leftarrow \lceil key/c \rceil$. If $U[key] = 0$ or $U[key]$ is not initialized, return $\perp$. Otherwise, let $j \leftarrow S[i]$ and return $A_j[P_j[rank(U,(i-1)c+1, key)]]$. The time is $O(\frac{c}{\log u})$ plus that to read the r bits of data.

Insert$(key, value)$**:** Let $i \leftarrow \lceil key/c \rceil$. If $U[key]$ is initialized and $U[key] = 1$ then find the old value as above and replace the r bits with $value$. Otherwise, we have to insert a new cell of A. If $S[i]$ is not initialized, then initialize it, so that $S[i] = j$ has a value. Now set $U[key] \leftarrow 1$, increase $nE_j \leftarrow nE_j + 1$,

[3] If $k = 0$ then we can just scan the permutation values one by one.

and add *value* at the end of EA A_j (making it grow first). Now, update the permutation: Let $p \leftarrow rank(U,(i{-}1)c{+}1, key)$ be the relative position of the cell to insert within the nonempty cells of the chunk. Shift the values from p to the end of P_j one position to the right to make room for $P_j[p] \leftarrow nE_j$ (this implies making the EA grow by one position). This operation takes constant amortized time for the EAs, plus $O(\frac{c}{\log u})$ for *rank* and chunk initialization, $O(\frac{c \log c}{\log u})$ to shift the values of permutation P_j, and the time to write r bits.

Delete(key)**:** If $U[key]$ is not initialized or $U[key] = 0$ then report an error or do nothing (key is not present in $\mathcal{S}$). Otherwise obtain the j associated to key as for retrieval. Let $p \leftarrow rank(U,(i-1)c+1, key)$, so the data is at $A_j[P_j[p]]$. Copy $A_j[P_j[p]] \leftarrow A_j[nE_j]$ and make A_j shrink. Find the position p' of nE_j in P_j, and set $P_j[p']] \leftarrow P_j[p]$. Move the cells from $p+1$ to the end of P_j one position to the left and make P_j shrink. Finally set $nE_j \leftarrow nE_j - 1$ and $U[key] \leftarrow 0$. For simplicity, empty EAs are not destroyed, but just kept with size zero. The cost of this operation is similar to that of insertion.

4.1 Analysis

In the following we omit the ceilings that should surround all the non-integer values, as this does not affect the result. The space is u bits for U, at most $2\frac{u}{c}\log\frac{u}{c}$ bits for S and S', and $\log\frac{u}{c}$ for nS'. The space of the collection of EAs follows Theorem 3: the nominal size is $n\log c + nr$ (for all the P_j and A_j, as they have one cell per element in $\mathcal{S}$), and there are at most $a = 2u/c$ EAs, therefore the total space for the EAs, including the nominal size, is $nr + O(n\log c + \frac{u}{c}w + \sqrt{n(r+\log c)\frac{u}{c}w}) = nr + O(n\log c + \frac{u}{c}w + \sqrt{nr\frac{u}{c}w})$ bits. Other $O(\frac{u}{c}w)$ bits are spent in the pointers P_j and A_j. The nE_j counters require $\frac{u}{c}\log c$ bits and the Four-Russians tables require $O(\sqrt{u}\log^2 u)$ bits. Overall, the space complexity is $nr + O(\frac{u}{c}\log u + n\log c + \frac{u}{c}w + \sqrt{nr\frac{u}{c}w})$. Considering Corollary 1, we can replace w by $\log((2u/c)(u\log c + ur)) = \Theta(\log u)$, to obtain $nr + O(\frac{u}{c}\log u + n\log c + \sqrt{nr\frac{u}{c}\log u})$ bits of space.

If $r = O(\log u)$, the time complexity for the operations is $O(1 + \frac{c\log c}{\log u})$ amortized time. To achieve constant time we need $c = O(\frac{\log u}{\log\log u})$. Using this maximum value the space complexity becomes $nr + O(u\log\log u + \sqrt{nur\log\log u})$. To convert the amortized into true worst-case time, the extra space is $O(nr)$, adding up to $O(nr + u\log\log u)$ bits overall.

If, alternatively, we wish to have extra space overhead linear in u, we need to set $c = \Theta(\log u)$, obtaining $nr + O(u + n\log\log u + \sqrt{nur})$ bits. The time complexity, however, raises to $O(\log\log u)$. To make this time worst-case, the overall space is $O(nr + u + n\log\log u)$ bits.

If $r = \omega(\log u)$ (but still $O(\text{poly}(u))$), then we must add $O(\frac{r}{\log u})$ to the time complexities. This can be improved to the optimal $O(r/w)$, as long as we add $O(aw) = O(\frac{u}{c}w)$ bits to the space complexity. Then the choices for c made above must change accordingly.

5 Dynamic Dictionaries: The Definitive Solution

The problem in the first approach is that we need c to be small enough to manipulate permutations of $[c]$ in constant time, and large enough to make the overhead associated to chunks small enough. In this section we show that a two-level scheme achieves both goals simultaneously.

(1) Exactly as in Section 4, we store U, divide it into chunks of c entries, and store arrays S, S', and the counter nS'.
(2) Aligned to the entries $S'[j]$, we store EAs E_j. Those are "large" EAs that will contain EA collections inside. In turn, all those EAs E_j are stored as a single collection of EAs.
(3) We divide the chunks into *blocks* of length b, so that b divides c.
(4) Aligned to the cells $S'[j]$, we store arrays $B_j[1, c/b]$, $B'_j[1, c/b]$, $P_j[1, c/b]$, and $A_j[1, c/b]$. (All those arrays are of fixed size and thus can be concatenated into single arrays B, B', etc.). B_j and B'_j play the role of S and S' at the block level, and local to chunk $S'[j]$. The other arrays have their cells aligned to those of B'_j. P_j will store block-level permutations and A_j will store the block data in compact form. Each value of arrays P_j and A_j will be an EA identifier. The collection of all such EAs (up to $2c/b$) will be stored within the large EA E_j.
(5) The actual sizes nB'_j of the arrays B'_j are stored in array nB', and the actual sizes $nE_j[j']$ of arrays $P_j[j']$ and $A_j[j']$ are stored in arrays nE_j (which, again, can be concatenated into a single array nE).

The scheme is illustrated in Fig. 1. Before explaining the operations, we redefine some terms related to initialization of U.

- $S[i]$ *is initialized* means the same as in Section 4.
- $B_j[i']$ *is initialized*, where $S[i] = j$ is initialized and $i' = \lceil (key - (i-1)c)/b \rceil$, means $1 \le B_j[i'] \le nB'_j$ and $B'_j[B_j[i']] = i'$.
- $U[key]$ *is initialized* means that $S[i]$ is initialized (where $i = \lceil key/c \rceil$) and that $B_j[i']$ is initialized (see previous item). That is, both the chunk and the block containing key are initialized.
- $rank(U, x, y)$ and the other Four-Russians subroutines are as before but now will take time $O(\frac{b}{\log u})$ or $O(\frac{b \log b}{\log u})$ as they work on blocks, not chunks.
- *Initializing* $S[i]$ means incrementing $nS' \leftarrow nS' + 1$, creating a new cell in $S'[nS'] \leftarrow i$ and $S[i] \leftarrow nS'$, creating a new empty EA $E_{nS'}$, and setting

$nB'_{nS'} \leftarrow 0$.

- *Initializing* $B_j[i']$ means incrementing $nB'_j \leftarrow nB'_j + 1$, setting $B'_j[nB'_j] \leftarrow i'$, and $B_j[i'] \leftarrow nB'_j$, allocating empty EAs $P_j[nB'_j]$ and $A_j[nB'_j]$ within large EA E_j, setting $nE_j[nB'_j] \leftarrow 0$, and finally setting $U[(i-1)c + (i'-1)b + 1, (i-1)c + i'b] \leftarrow 0$. This last assignment can be done in time $O(b/w)$ on the RAM model.

Let us now explain how we carry out the operations over this data structure. We note that now we must use initializable arrays in the second-level structure as well, in order to achieve constant time.

Create: Set $nS' \leftarrow 0$.

Retrieve(key)**:** Let $i \leftarrow \lceil key/c \rceil$ and $i' \leftarrow \lceil (key - (i-1)c)/b \rceil$. If $U[key] = 0$ or $U[key]$ is not initialized, then return $\perp$. Otherwise, let $j \leftarrow S[i]$ and $j' \leftarrow B_j[i']$. Let D be the permutation (EA) at address $P_j[j']$ within EA E_j, and A be the EA at position $A_j[j']$ within EA E_j. Now, return the r bits of $A[D[rank(U, (i-1)c + (i'-1)b + 1, key)]]$. All this takes constant worst-case time, plus $O(\frac{b}{\log u})$ worst-case time to solve *rank*.

Insert($key, value$)**:** Let $i \leftarrow \lceil key/c \rceil$ and $i' \leftarrow \lceil (key - (i-1)c)/b \rceil$. If $U[key]$ is initialized and $U[key] = 1$ then find the old value as above and replace the r bits with *value*. Otherwise, we have to insert a new cell. If $S[i]$ is not initialized, then initialize it, so that $S[i] = j$ has a value. If $B_j[i']$ is not initialized, then initialize it, so that $B_j[i'] = j'$ has a value. Now, set $U[key] \leftarrow 1$ and update the permutation and data exactly as in Section 4 with $P_j[j']$, $A_j[j']$, and $E_j[j']$ instead of P_j, A_j, and E_j. This operation takes constant amortized time plus $O(\frac{b \log b}{\log u})$ time to manage the permutations.

Delete(key)**:** If $U[key]$ is not initialized or $U[key] = 0$ then report an error or do nothing (key is not present in $\mathcal{S}$). Otherwise locate the key as for retrieval. Let $p \leftarrow rank(U, (i-1)c + (i'-1)b + 1, key)$. Then do exactly as in Section 4 with $P_j[j']$, $A_j[j']$, and $E_j[j']$. The cost of this operation is similar to the case of insertion.

5.1 Analysis

The space is u bits for U, at most $2\frac{u}{c}\log\frac{u}{c}$ bits for S and S', and $\log\frac{u}{c}$ for nS', just as in Section 4, from where we also borrow the small $O(\sqrt{u}\log^2 u)$ space required for the Four-Russians tables. Now, for a given j, we have the arrays and counters B_j, B'_j, nB'_j, and nE_j, which require at most $2\frac{c}{b}\log\frac{c}{b} + \log\frac{c}{b} + \frac{c}{b}\log b = O(\frac{c}{b}\log c)$ bits each. Added over the u/c values j, these totalize $O(\frac{u}{b}\log c)$ bits. Also, we have all the u/b arrays $P_j[j']$ and $A_j[j']$, whose contents add up to $n\log b + nr$ nominal bits overall. These are spread across at most u/c EAs E_j, each of which stores at most $a^* = 2c/b$ entries ($P_j[j']$ and $A_j[j']$ for $1 \le j' \le c/b$) and handles at most $s^* = c\log b + cr$ bits. Therefore the pointers within

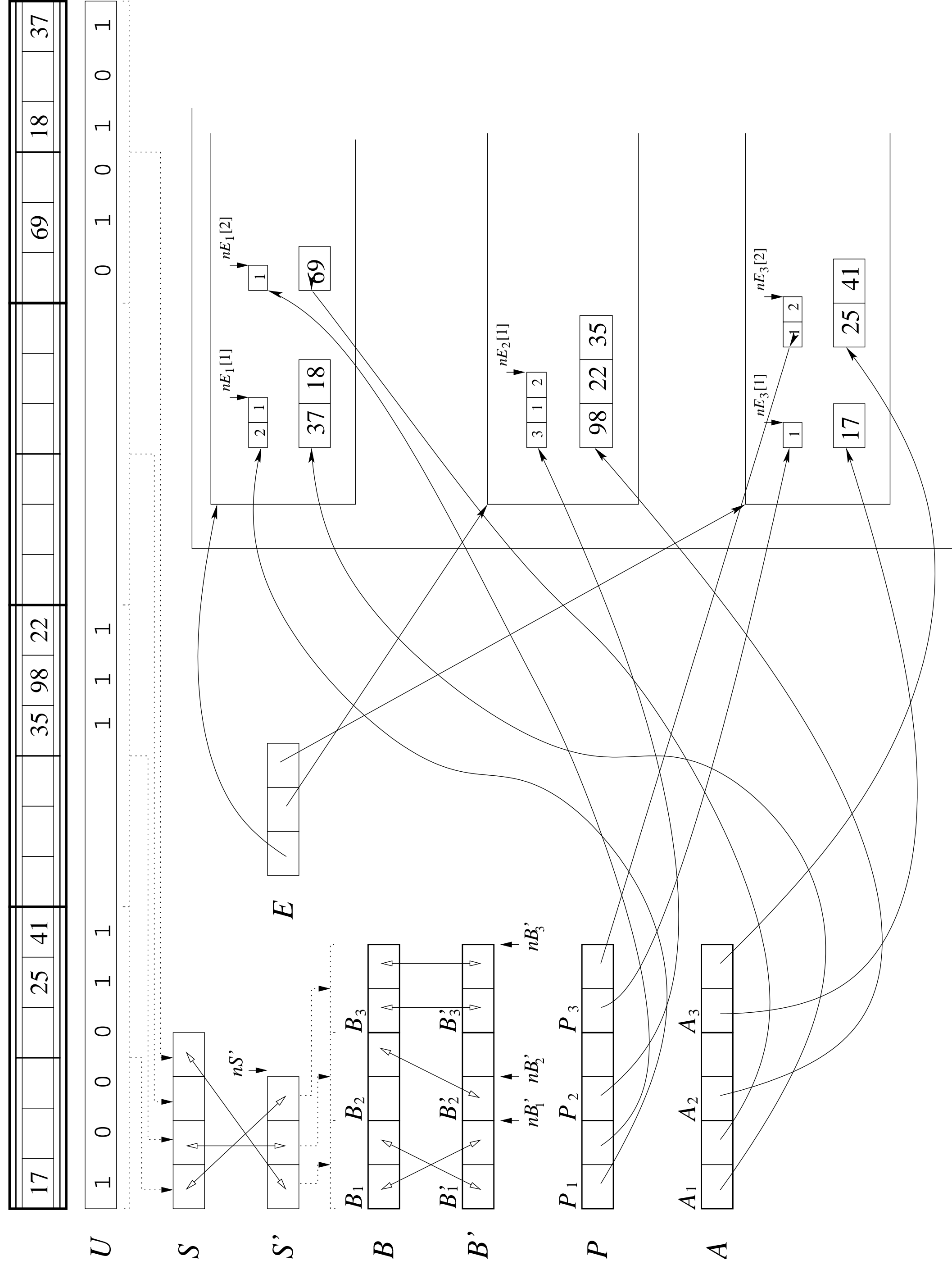

Fig. 1. The definitive scheme illustrated for $u = 24$, $c = 6$, $b = 3$, $n = 9$, and the data corresponding to the array on top (which is not stored).

the EAs E_j require $\log(s^*a^*) = \Theta(\log(cr))$ bits. On one hand, the $2u/b$ array identifiers require $O(\frac{u}{b}\log(cr))$ bits. As for their contents, since the extra space of storing an EA collection is convex, the worst case arises when each E_j handles $\frac{n}{u/c}$ entries (thus $s = \frac{n}{u/c}(\log b + r)$ for each E_j). In this worst situation, the total *extra* space of each E_j is $O(\frac{c}{b}\log(cr) + \sqrt{\frac{n}{u/c}(\log b + r)\frac{c}{b}\log(cr)})$. Added over all the u/c E_j's, this gives a *nominal size* (now including the true data and the EA overheads) of $n(\log b + r) + O(\frac{u}{b}\log(cr) + \sqrt{\frac{nu}{b}(\log b + r)\log(cr)})$. In turn, all the E_j's are maintained within a large EA collection where $a = a^* = u/c$ and $s = O(n(\log b + r) + \frac{u}{b}\log(cr))$ is precisely the nominal size for all E_j's just given. Thus $\log(a^*s^*) = \Theta(\log(ur)) = \Theta(\log u)$, so the identifiers E_j add up to $O(\frac{u}{c}\log u)$ bits and the *extra space* to store this EA collection is $O(\frac{u}{c}\log u + \sqrt{(n(\log b + r) + \frac{u}{b}\log(cr))\frac{u}{c}\log u})$. Adding up the $n(\log b + r) = nr + O(n\log b)$ nominal bits, the total space of the EA collection is $nr + O(\frac{u}{c}\log u + n\log b + \sqrt{(nr + \frac{u}{b}\log(cr))\frac{u}{c}\log u})$. Adding up all the space requirements, we achieve $nr + u + O(\frac{u}{c}\log u + \frac{u}{b}\log(cr) + n\log b + \sqrt{nr\frac{u}{c}\log u})$.

The time complexity is at most $O(1 + \frac{b\log b}{\log u})$. To achieve constant time we need $b = O(\frac{\log u}{\log\log u})$, while we are free to choose c. For example, if $r = O(\log u)$, we can choose $c = \Theta(\log^3 u)$ so that the space becomes $nr + u + O(\frac{u}{\log^2 u} + \frac{u}{b}\log\log u + n\log b + \sqrt{\frac{nu}{\log u}}) = nr + u + o(u) + O(\frac{u}{b}\log\log u + n\log b)$. Now we can use b to achieve different tradeoffs depending on n. For example, if we wish to minimize the dependence on n we can choose $b = \Theta((\log\log u)^k)$ (for any $k > 1$) so that the space is $nr + u + o(u) + O(n\log\log\log u)$. If we want a smaller $o(u)$ term, say at most $O(u\frac{\log\log u}{\log u})$, we can choose $b = \Theta(\frac{\log u}{\log\log u})$ to achieve space $nr + u + o(u) + O(n\log\log u)$.

Theorem 4 *The dynamic dictionary problem, with universe size u and current set size n, each element storing $r = O(\log u)$ bits of data, can be solved on the RAM model with word size $\Omega(\log u)$ using $nr + u + o(u) + O(n\log\log\log u)$ bits of space so that creation and querying can be done in worst-case constant time, and insertions and deletions in constant amortized time. Those can become constant worst-case time by spending $O(nr)$ additional bits of space.*

We can handle larger $r = O(\text{polylog}(u))$ values, spending $O(r/\log u)$ extra time per operation. In this case we have to choose $c = \Theta(r\log^2 u)$ to achieve the same result as above. It is possible to handle slightly larger r values, but still $\frac{u}{b}\log(cr)$ must be $o(u)$. For example, we can still obtain the same result if $\log r = O(\text{poly}(\log\log u))$, or $O(n\log\log u)$ extra bits of space if $\log r = O(\frac{\log u}{\log\log u})$.

If we wish to achieve optimal $O(r/w)$ time to access the r bits, then we must ensure records of size $\Theta(w)$ within all the EAs, which introduces some extra terms in the space complexity. Those vanish only if $w = \Theta(\log u)$.

6 Concluding Remarks

This paper contributes to the understanding of the space/time tradeoffs for one of the most basic data structures: a dictionary where one can insert, delete, and search for elements with r-bit associated satellite data. While most of the research has focused on using space linear on the set size, where lower bounds show that worst-case constant time cannot be achieved, we have shown that this complexity is indeed possible if a number of bits linear in the universe size are stored. It is not clear whether our scheme is optimal, as we are not aware of any lower bound that applies to this case. Such a lower bound would further contribute to our understanding of dynamic dictionaries. With regard to practicality, our structure is not complicated to implement (especially if one can resort to system-provided allocation instead of EAs), and can be interesting in practice for not very sparse sets (say, $n = \Omega(u/\log u)$), or when guaranteeing constant time is a must.

As a byproduct, we obtained a succinct version of a folklore technique to initialize an array in constant time, by reducing the extra space per data element from $2\log n$ to $1 + o(1)$ bits. Not only this result was necessary to obtain our main contribution, but it is also of independent theoretical and practical interest. We ran some experiments to illustrate its practicality. Empirical tests run on an Intel machine using an array of integers show that the folklore solution is faster than explicitly running the initialization as long as the number of operations done over the array does not exceed 8% of its size. Our $3n$-bit solution is faster only up to 2.5% due to the increased complexity. Yet, the folklore solution crashes when the array is so large that the extra structures (200% extra space) require swapping, whereas the extra space of our solution (less than 10%) goes unnoticed.

Acknowledgments

We thank Rajeev Raman for useful comments and for sharing a draft of the journal version of [9]. We also thank Alberto Apostolico for pointing us the earlier reference [1] for the folklore technique of constant-time array initialization. Last but not least, we thank our student Diego Arroyuelo for proofreading and debugging the paper.

References

[1] A. Aho, J. Hopcroft, and J. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.

[2] E. Demaine, F. Meter auf der Heide, R. Pagh, and M. Patrascu. De dictionariis dynamicis pauco spatio utentibus (lat. on dynamic dictionaries using little space). In *Proc. 7th Latin American Symposium on Theoretical Informatics (LATIN)*, LNCS 3887, pages 349–361, 2006.

[3] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, and F. Meter auf der Heide. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994. Earlier version in *Proc. ACM FOCS 1988*.

[4] M. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *Proc. 21st ACM Symposium on Theory of Computing (STOC)*, pages 345–354, 1989.

[5] T. Hagerup, K. Mehlhorn, and I. Munro. Maintaining discrete probability distributions optimally. In *Proc. 20th International Colloquium on Automata, Languages and Computation (ICALP)*, LNCS 700, pages 253–264, 1993.

[6] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1984.

[7] P. Miltersen. Cell probe complexity — a survey. In *Pre-Conference Workshop on Advances in Data Structures at the 19th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, 1999.

[8] M. Overmars. *The design of dynamic data structures*. LNCS 156. Springer-Verlag, 2nd edition, 1987.

[9] R. Raman and S. Srinivasa Rao. Succinct dynamic dictionaries and trees. In *Proc. 30th International Colloquium on Automata, Languages and Computation (ICALP)*, LNCS 2719, pages 357–368, 2003.