An Efficient Quasidictionary

Torben Hagerup[1] and Rajeev Raman[2*]

[1] Institut für Informatik, Johann Wolfgang Goethe-Universität Frankfurt, D–60054 Frankfurt am Main. hagerup@ka.informatik.uni-frankfurt.de

[2] Department of Maths and Computer Science, University of Leicester, Leicester LE1 7RH, UK. r.raman@mcs.le.ac.uk

Abstract. We define a *quasidictionary* to be a data structure that supports the following operations: check-in(v) inserts a data item v and returns a positive integer tag to be used in future references to v; check-out(x) deletes the data item with tag x; access(x) inspects and/or modifies the data item with tag x. A quasidictionary is similar to a dictionary, the difference being that the names identifying data items are chosen by the data structure rather than by its user. We describe a deterministic quasidictionary that executes the operations check-in and access in constant time and check-out in constant amortized time, works in linear space, and uses only tags bounded by the maximum number of data items stored simultaneously in the quasidictionary since it was last empty.

1 Introduction

Many data structures for storing collections of elements and executing certain operations on them have been developed. For example, the red-black tree [14] supports the operations insert, delete, and access, among others, and the Fibonacci heap [10] supports such operations as insert, decrease, and extractmin. An element is typically composed of a number of fields, each of which holds a value. Many common operations on collections of elements logically pertain to a specific element among those currently stored. E.g., the task of a delete operation is to remove an element from the collection, and that of a decrease operation is to lower the value of a field of a certain element. Therefore the question arises of how a user of a data structure can refer to a specific element among those present in the data structure.

The elements stored in a red-black tree all contain a distinguished key drawn from some totally ordered set, and the tree is ordered with respect to the keys. In some cases, it suffices to specify an element to be deleted, say, through its key. If several elements may have the same key, however, this approach is not usable, as it may not be immaterial which of the elements with a common key gets deleted. Even when keys are unique, specifying an element to be deleted from a Fibonacci heap through its key is not a good idea, as Fibonacci heaps

* Supported in part by EPSRC grant GR L/92150.

do not support (efficient) searching: Locating the relevant key may involve the inspection of essentially all of the elements stored, an unacceptable overhead. In still other cases, such as with finger trees [5, 17], even though reasonably efficient search is available, it may be possible to execute certain operations such as deletions faster if one already knows "where to start".

For these reasons, careful specifications of data-structure operations such as delete state that one of their arguments, rather than being the element to be operated on, is a pointer to that element. Several interpretations of the exact meaning of this are possible. In their influential textbook, Cormen et al. use the convention that elements are actually stored "outside of" the data structures, the latter storing only pointers to the elements [6, Part III, Introduction]. Thus insert and delete operations alike take a pointer to an element as their argument. In contrast, the convention employed by the LEDA library is that (copies of) elements are stored "inside" the data structures, and that an insertion of an element returns a pointer (a "dependent item", in LEDA parlance) to be used in future references to the element [19, Section 2.2].

The method employed by Cormen et al. can have its pitfalls. Consider, e.g., the code proposed for the procedure BINOMIAL-HEAP-DECREASE-KEY in [6, Section 20.2] and note that the "bubbling up" step repeatedly exchanges the *key* fields and all satellite fields of two nodes y and z, in effect causing y and z to refer to different elements afterwards. Because of this, repeated calls such as BINOMIAL-HEAP-DECREASE-KEY(H, x, k_1); BINOMIAL-HEAP-DECREASE-KEY (H, x, k_2), despite their appearance, may not access the same logical element. Indeed, as binomial heaps do not support efficient searching, it will not be easy for a user to get hold of valid arguments x to use in calls of the form BINOMIAL-HEAP-DECREASE-KEY(H, x, k). One might think that this problem could be solved by preserving the *key* and satellite fields of the nodes y and z and instead exchanging their positions within the relevant binomial tree. Because of the high degrees of some nodes in binomial trees, however, this would ruin the time bounds established for the BINOMIAL-HEAP-DECREASE-KEY operation.

Presumably in recognition of this problem, in the second edition of their book, Cormen et al. include a discussion of so-called *handles* [7, Section 6.5]. One kind of handle, which we might call an *inward handle*, is a pointer from an element of an application program to its representation in a data structure in which, conceptually, it is stored. The inward handle has a corresponding *outward handle* that points in the opposite direction. Whenever the representation of an element is moved within a data structure, its outward handle is used to locate and update the corresponding inward handle. While this solution is correct, it is not modular, in that the data structure must incorporate knowledge of the application program that uses it. It can be argued that this goes against current trends in software design.

A solution that does not suffer from this drawback can be obtained by employing the LEDA convention described above and introducing one more level of indirection: The pointer supplied by the user of a binomial heap would then point not into the heap itself, but to an auxiliary location that contains the actual pointer into the heap and is updated appropriately whenever the element designated moves within the heap.

In the following, we adopt the LEDA convention and assume that every insertion of an element into a data structure returns a pointer to be used in future references to the element. The pointer can be interpreted as a positive integer, which we will call the *tag* of the element. At every insertion of an element, the data structure must therefore "invent" a suitable tag and hand it to the user, who may subsequently access the element an arbitrary number of times through its tag. Eventually the user may delete the element under consideration, thus in effect returning its tag to the data structure for possible reuse. During the time interval in which a user is allowed to access some element through a tag, we say that the tag is *in use*. When the tag is not in use, it is *free*. Fig. 1 contrasts the solution using handles with the one based on tags.

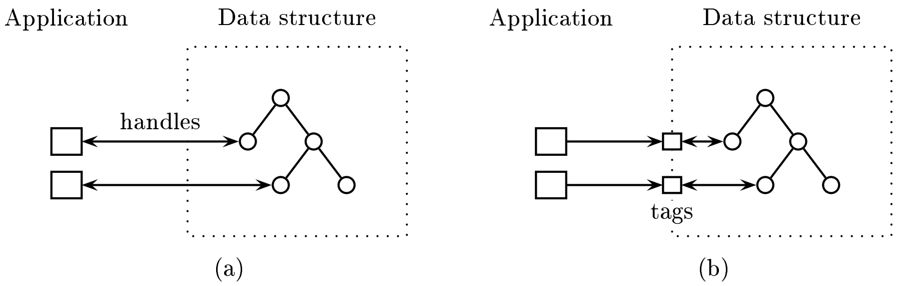

Fig. 1. The interplay between an application and a data structure based on (a) handles and (b) tags.

It might seem that the only constraint that a data structure must observe when "inventing" a tag is that the tag should not already be in use. This constraint can easily be satisfied by maintaining the set of free tags in a linked list: When a new tag is needed, the first tag on the free list is deleted from the list and put into use, and a returned tag is inserted at the front of the list. In general, however, a data structure has close to no control over which tags are in use. In particular, it may happen that at some point, it contains only few elements, but that these have very large tags. Recall that the purpose of a tag is to allow efficient access to an element stored in the data structure. The straightforward way to realize this is to store pointers to elements inside the data structure in a *tag array* indexed by tag values. Now, if some tags are very large (compared to the current number of elements), the tag array may have to be much bigger than the remaining parts of the data structure, putting any bounds established for its space consumption in jeopardy.

In his discussion of adding the `delete` operation to priority queues that lack it, Thorup [23, Section 2.3] proposes the use of tags (called "identifiers") and a tag array (called "D"). He recognizes the problem that identifiers may become too big for the current number of elements. The solution that he puts forward to solve this problem places the burden on the user of a data structure: Whenever the number of elements stored has halved, the user is required to extract all elements from the data structure and subsequently reinsert them, which gives the data structure an opportunity to issue smaller tags. Our goal in this paper is to provide an algorithmic solution to the problem.

Even when no tag arrays are needed, small tags may be advantageous, simply because they can be represented in fewer bits and therefore stored more compactly. This is particularly crucial in the case of *packed* data structures, which represent several elements in a single computer word and operate on all of them at unit cost. In such a setting, tags must be stored with the elements that they designate and follow them as they move around within a data structure. Therefore it is essential that tags can be packed as tightly as the corresponding elements, which means that they should consist of essentially no more bits. The investigation reported here was motivated by an application to a linear-space packed priority queue.

We formalize the problem of maintaining the tags of a data structure by defining a *quasidictionary* to be a data structure that stores an initially empty set S of elements and supports the following operations:

check-in(v), where v is an element (an arbitrary constant-size collection of fields), inserts v in S and returns a positive integer called the tag of v and distinct from the tag of every other element of S.

check-out(x), where x is the tag of an element in S, removes that element from S.

access(x), where x is the tag of an element in S, inspects and/or modifies that element (but not its tag); the exact details are of no importance here.

The term "quasidictionary" is motivated by the similarity between the data structure defined above and the standard *dictionary* data structure. The latter supports the operations insert(x, v), delete(x), and access(x). The access operations of the two data structures are identical, and delete(x) has the same effect as check-out(x). The difference between insert(x, v) and check-in(v) is that in the case of insert(x, v), the tag x that gives access to v in later operations is supplied by the user, whereas in the case of check-in(v), it is chosen by the data structure. In many situations, the tags employed are meaningful to the user, and a dictionary is what is called for. In other cases, however, the elements to be stored have no natural a priori names, and a quasidictionary can be used and may be more convenient.

Quasidictionaries have potential applications to storage management. A storage allocator must process online a sequence of requests for contiguous blocks of memory of specified sizes, interspersed with calls to free blocks allocated earlier. It is known [21, 22] that if N denotes the maximum total size of the blocks in use simultaneously, the storage allocator needs $\Omega(N \log N)$ words of memory to serve the requests in the worst case. As proved in [18], randomization does not help very much. A decision version of the corresponding offline problem, in which all requests are known initially, is NP-complete, even when all block sizes are 1 or 2 [12, Section A4.1], and the best polynomial-time approximation algorithm known has an approximation ratio of 3 [13]. Thus the storage-allocation problem is difficult in theory. As argued in detail in [24], despite the use of a number of heuristics, it also continues to constitute a challenge in practice.

Conventional storage allocators are not allowed to make room for a new block by *relocating* other blocks while they are still in use. This is because of the difficulty or impossibility of finding and updating all pointers that might point into such blocks. Relocating storage allocators, however, can achieve a superior memory utilization in many situations. This is where quasidictionaries come into the picture: Equipped with a "front-end" quasidictionary, a relocating storage allocator can present the interface of a nonrelocating one. An application requesting a block of memory then receives a tag that identifies the block rather than a pointer to the start of the block, and subsequent accesses to the block are made by passing the tag of the block and an offset within the block to a privileged operation that accesses the quasidictionary. This has a number of disadvantages. First, the tag cannot be used as a "raw" pointer into memory, and address arithmetic involving the tag is excluded. This may not be a serious disadvantage. Indeed, the use of pointers and pointer arithmetic is a major source of difficult-to-find software errors and "security holes". Second, memory accesses are slowed down due to the extra level of indirection. However, in the interest of security and programmer productivity, considerable run-time checks (e.g., of array indices against array bounds) were already made an integral part of modern programming languages such as Java.

Our model of computation is the unit-cost *word RAM*, a natural and realistic model of computation described and discussed more fully in [15]. Just as a usual RAM, a word RAM has an infinite memory consisting of cells with addresses $0, 1, 2, \ldots$, not assumed to be initialized in any particular way. For a positive integer parameter w, called the *word length*, every memory cell stores an integer in the range $\{0, \ldots, 2^w - 1\}$, known as a *word*. We assume that the instructions in the so-called *multiplication instruction set* can be executed in constant time on w-bit operands. These include addition, subtraction, bitwise Boolean operations, left and right bit shifts by an arbitrary number of positions, and multiplication. The word length w is always assumed to be sufficiently large to allow the memory needed by the algorithms under consideration to be addressed; in the context of the present paper, this will mean that if a quasidictionary at some point must store n elements, then $w \geq \log_2 n + c$ for a suitable constant $c > 0$; in particular, we will have $n < 2^w$. An algorithm for the word RAM is called *weakly nonuniform* if it needs access to a constant number of precomputed constants that depend on w. If it needs only the single constant 1, it is called *uniform*. When nothing else is stated, uniformity is to be understood.

Since the space requirements of data structures are of central interest to the present paper, we provide a careful definition, which is somewhat more involved than the corresponding definition for language-recognition problems. Informally, the only noteworthy points are that the space consumption is related not to an input size, but to the number of elements currently stored, and that space is counted in units of w-bit words.

First, a *legal input* to a data structure D is a sequence of operations that D is required to execute correctly, starting from its initial state. We will say that D *uses space* s during the execution of a legal input σ_1 to D if, even if the contents of all memory cells with addresses $\geq s$ are altered arbitrarily at arbitrary times during the execution of σ_1, subsequently every operation sequence σ_2 such that $\sigma_1\sigma_2$ is a legal input to D is executed correctly, and if s is the smallest non-negative integer with this property. Assume now that D stores sets of elements and let $\mathbb{N} = \{1, 2, \ldots\}$ and $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$. The *space consumption* of D is the pointwise smallest function $g : \mathbb{N}_0 \to \mathbb{N}_0$ such that for all legal inputs σ to D, if D contains n elements after the execution of σ, then the space used by D during the execution of σ is bounded by $g(n)$. If the space consumption of D is $O(n)$, we say that D works in linear space.

Our main result is the following:

Theorem 1. *There is a uniform deterministic quasidictionary that can be initialized in constant time, executes the operations* check-in *and* access *in constant time and* check-out *in constant amortized time, works in linear space, and uses only tags bounded by the maximum number of tags in use simultaneously since the quasidictionary was last empty.*

A result corresponding to Theorem 1 is not known for the more powerful dictionary data structure. Constant execution times in conjunction with a linear space consumption can be achieved, but only at the expense of introducing randomization [9]. A deterministic solution based on balanced binary trees executes every operation in $O(\log n)$ time, where n is the number of elements currently stored, whereas a data structure of Beame and Fich [4], as deamortized by Andersson and Thorup [3], yields execution times of $O(\sqrt{\log n / \log\log n})$. If we insist on constant-time access, the best known result performs insertions and deletions in $O(n^{\epsilon})$ time, for arbitrary fixed $\epsilon > 0$ [16]. We are not aware of any previous work specifically on quasidictionaries.

2 Four Building Blocks

We will compose the quasidictionary postulated in Theorem 1 from four simpler data structures. One of these is a quasidictionary, except that it supports an additional operation scan that lists all tags currently in use, and that it is initialized with a set of pairs of (distinct) tags already in use and their corresponding elements, rather than starting out empty; we call such a data structure an *initialized quasidictionary.* The three other data structures are *static dictionaries*, which are also initialized with a set of (tag , element) pairs, and which subsequently support only the operation access. The four building blocks are characterized in the following four lemmas.

Lemma 1. *There is an initialized quasidictionary that, provided that every tag initially in use is bounded by N and that the number of tags simultaneously in use is guaranteed never to exceed N, can be initialized in $O(N)$ time and* *space, uses $O(N)$ space, and executes* check-in, check-out *and* access *in constant time and* scan *in $O(N)$ time. Moreover, every tag used is bounded by a tag initially present or by the maximum number of tags in use simultaneously since the quasidictionary was initialized.*

Proof. Use a tag array storing for each of the integers $1, \ldots, N$ whether it is currently in use as a tag and, if so, its associated element. Moreover, maintain the set of free tags in $\{1, \ldots, N\}$ in a linked list as described in the introduction. By assumption, no tag will be requested when the list is empty, and it is easy to carry out the operations within the stated time bounds. Provided that the initial free list is sorted in ascending order, no check-in operation will issue a tag that is too large.

We will refer to the integer N that appears in the statement of the previous lemma as the *capacity* of the data structure. A data structure similar to that of the lemma, but without the need for an initial specification of a capacity, was implemented previously by Demaine [8] for use in a simulated multiprocessor environment.

Lemma 2. *There is a static dictionary that, when initialized with $n \geq 4$ tags in $\{1, \ldots, N\}$ and their associated elements, can be initialized in $O(n + N/\log N)$ time and space, uses $O(n+N/\log N)$ space, and supports constant-time accesses.*

Proof. Assume without loss of generality that the tag N is in use. The data structure begins by computing a quantity b as a power of two with $2 \leq b \leq \log N$, but $b = \Omega(\log N)$, as well as $\log b$, with the help of which divisions by b can be carried out in constant time. Similarly, it computes l as a power of two with $\log b \leq l \leq b$, but $l = O(\log b)$, as well as $\log l$.

Let S be the set of tags to be stored. For $j = 0, \ldots, r = \lfloor N/b \rfloor$, let $U_j = \{x \mid 1 \leq x \leq N \text{ and } \lfloor x/b \rfloor = j\}$ and take $S_j = S \cap U_j$. For $j = 0, \ldots, r$, the elements with tags in S_j are stored in a separate static dictionary D_j, the starting address of which is stored in position j of an array of size $r + 1$. For $j = 0, \ldots, r$, D_j consists of an array B_j of size $|S_j|$ that contains the elements with tags in S_j and a table L_j that maps every tag in S_j to the position in B_j of its associated element. L_j is organized in one of two ways.

If $|S_j| \geq l$, L_j is an array indexed by the elements of U_j. Since $|S_j| \leq b \leq 2^l$, every entry in L_j fits in a field of l bits. Moreover, by assumption, the word length w is at least $\log N \geq b$. We therefore store L_j in $l \leq |S_j|$ words, each of which contains b/l fields of l bits each. It is easy to see that accesses to L_j can be executed in constant time.

If $|S_j| < l$, L_j is a list of $|S_j|$ pairs, each of which consists of the remainder modulo b of a tag $x \in S_j$ (a *tag remainder*) and the corresponding position in B_j. Each pair can be represented in $2l$ bits, so the entire list L_j fits in $2l^2 = O((\log \log N)^2)$ bits. We precompute a table, shared between all of $D_0, \ldots, D_r$, that maps a list L_j with $|S_j| < l$ and a tag remainder to the corresponding position in B_j. The table occupies $o(r)$ words and is easily computed in $o(r)$ time. Again, accesses to L_j can be executed in constant time.

It is simple to verify that the overall dictionary uses $O(r+\sum_{j=0}^{r}(1+|S_j|)) = O(n+N/\log N)$ space, can be constructed in $O(n+N/\log N)$ time, and supports accesses in constant time.

Lemma 3. *There is a static dictionary that, when initialized with $n \geq 2$ tags in $\{1,\ldots,N\}$ and their associated elements, can be initialized in $O(n \log n)$ time and $O(n)$ space, uses $O(n)$ space, and supports accesses in $O(1+\log N/\log n)$ time.*

Proof. We give only the proof for $N = n^{O(1)}$, which is all that is needed in the following. The proof of the general case proceeds along the same lines and is hardly any more difficult.

The data structure begins by sorting the tags in use. Viewing each tag as a k-digit integer to base n, where $k = O(1+\log N/\log n) = O(1)$, and processing the tags one by one in sorted order, it is then easy, in $O(n)$ time, to construct a *trie* or *digital search tree* T for the set of tags. T is a rooted tree with exactly n leaves, all at depth exactly k, each edge of T is labeled with a digit, and each leaf u of T corresponds to a unique tag x in the sense that the labels on the path in T from the root to u are exactly the digits of x in the order from most to least significant. If the element associated with each tag is stored at the corresponding leaf of T, it is easy to execute access operations in constant time by searching in T, provided that the correct edge to follow out of each node in T can be found in constant time.

T contains fewer than n *branching nodes*, nodes with two or more children, and these can be numbered consecutively starting at 0. Since the total number of children of the branching nodes is bounded by $2n$, it can be seen that the problem of supporting searches in T in constant time per node traversed reduces to that of providing a static dictionary for at most $2n$ keys, each of which is a pair consisting of the number of a branching node and a digit to base n, i.e., is drawn from a universe of size bounded by n^2. We construct such a dictionary using the $O(n \log n)$-time procedure described in [16, Section 4].

Remark. The main result of [16] is a static dictionary for n *arbitrary* keys that can be constructed in $O(n \log n)$ time and supports constant-time accesses. The complete construction of that dictionary is weakly nonuniform, however, which is why we use only a uniform part of it.

Lemma 4. *There is a constant $\nu \in \mathbb{N}$ and functions $f_1,\ldots,f_\nu : \mathbb{N}_0 \to \mathbb{N}_0$, evaluable in linear time, such that given a set S of $n \geq 2$ tags bounded by N with associated elements as well as $f_1(b),\ldots,f_\nu(b)$ for some integer b with $\log N \leq b \leq w$, a static dictionary for S that uses $O(n)$ space and supports accesses in constant time can be constructed in $O(n^5)$ time using $O(n)$ space.*

Proof. For $\log N \leq n^3$, we use a construction of Raman [20, Section 4], which works in $O(n^2 \log N) = O(n^5)$ time.

For $\log N > n^3$, the result can be proved using the *fusion trees* of Fredman and Willard [11] (see [15, Corollary 8]). Reusing some of the ideas behind the fusion trees, we give a simpler construction.

Number the bit positions of a w-bit word from the right, starting at 0. Our proof takes the following route: First we show how to compute a set A of fewer than n bit positions such that two arbitrary distinct tags in S differ in at least one bit position in A. Subsequently we describe a constant-time transformation that, given an arbitrary word x, clears the bit positions of x outside of A and compacts the positions in A to the first n^3 bit positions. Noting that this maps S injectively to a universe of size 2^{n^3}, we can appeal to the first part of the proof.

Write $S = \{x_1, \ldots, x_n\}$ with $x_1 < \cdots < x_n$. We take

$$A = \{\mathrm{MSB}(x_i \oplus x_{i+1}) \mid 1 \le i < n\},$$

where $\oplus$ denotes bitwise exclusive-or and $\mathrm{MSB}(x) = \lfloor \log_2 x \rfloor$ is the *most significant bit* of x, i.e., the position of the leftmost bit of x with a value of 1. In other words, A is the set of most significant bit positions in which consecutive elements of S differ. To see that elements x_i and x_j of S with $x_i < x_j$ differ in at least one bit position in A even if $j \ne i+1$, consider the digital search tree T for S described in the previous proof, but for base 2, and observe that $\mathrm{MSB}(x_i \oplus x_j) = \mathrm{MSB}(x_r \oplus x_{r+1})$, where x_r is the largest key in S whose corresponding leaf in T is in the left subtree of the lowest common ancestor of the leaves corresponding to x_i and x_j.

By assumption, $\oplus$ is a unit-time operation, and Fredman and Willard have shown how to compute $\mathrm{MSB}(x)$ from x in constant time [11, Section 5]. Their procedure is weakly nonuniform in our sense. Recall that this means that it needs to access a fixed number of constants that depend on the word length w. It is easy to verify from their description that the constants can be computed in $O(w)$ time (in fact, in $O(\log w)$ time). We define ν and $f_1, \ldots, f_\nu$ so that the required constants are precisely $f_1(b), \ldots, f_\nu(b)$. For this we must pretend that the word length w is equal to b, which simply amounts to clearing the bit positions $b, b+1, \ldots$ after each arithmetic operation. It follows that A can be computed in $O(n \log n)$ time.

Take $k = |A|$ and write $A = \{a_1, \ldots, a_k\}$. Multiplication with an integer $\mu = \sum_{i=1}^{k} 2^{m_i}$ can be viewed as copying each bit position a to positions $a + m_1, \ldots, a + m_k$. Following Fredman and Willard, we plan to choose μ so that no two copies of elements of A collide, while at least one *special copy* of each element of A is placed in one of the bit positions $b, \ldots, b+k^3-1$. The numbers $m_1, \ldots, m_k$ can be chosen greedily. For $i = 1, \ldots, k$, none of the k copies of a_i may be placed in one of the at most $(k-1)k$ positions already occupied, which leaves at least one position in the range $b, \ldots, b+k^3-1$ available for the special copy of a_i. Thus a suitable multiplier μ can be found in $O(k^4) = O(n^4)$ time.

An operation $\mathsf{access}(x)$ is processed as follows: First the bits of x outside of positions in A are cleared. Then the product μx is formed. It may not fit in one word. Since $b + k^3 \le w + n^3 \le w + \log N \le 2w$, however, it fits in two words and can be computed in constant time using simulated double-precision arithmetic. Shifting μx right by b bits and clearing positions $n^3, n^3+1, \ldots$ completes the transformation of x, and the transformed tag is accessed in a data structure constructed using Raman's method.

The static dictionaries of Lemmas 2, 3 and 4 can easily be extended to support deletions in constant time and scans in $O(n)$ time. For deletion, it suffices to mark each tag with an additional bit that indicates whether the tag is still in use. For scan, step through the list from which the data structure was initialized (which must be kept around for this purpose) and output precisely the tags not marked as deleted.

3 The Quasidictionary

In this section we describe a quasidictionary with the properties claimed in Theorem 1, except that check-in operations take constant time only in an amortized sense. After analyzing the data structure in the next section, we describe the minor changes needed to turn the amortized bound for check-in into a worst-case bound. Since we can assume that the data structure is re-initialized whenever it becomes empty, in the following we will simplify the language by assuming that the data structure was never empty in the past after the execution of the first (check-in) operation.

Observe first that for any constant $c \in \mathbb{N}$, the single semi-infinite memory at our disposal can emulate c separate semi-infinite memories with only a constant-factor increase in execution times and space consumption: For $i \in \mathbb{N}_0$, the real memory cell with address i is simply interpreted as the cell with address $\lfloor i/c \rfloor$ in the $((i \bmod c)+1)$st emulated memory. Then a space bound of $O(n)$ established for each emulated memory translates into an overall $O(n)$ space bound. In the light of this, we shall feel free to store each of the main components of the quasidictionary in its own semi-infinite array.

We divide the set of potential tags into *layers*: For $j \in \mathbb{N}$, the jth layer U_j consists of those natural numbers whose binary representations contain exactly j bits; i.e., $U_1 = \{1\}$, $U_2 = \{2,3\}$, $U_3 = \{4,\ldots,7\}$ and so on. In general, for $j \in \mathbb{N}$, $U_j = \{2^{j-1},\ldots,2^j - 1\}$ and $|U_j| = 2^{j-1}$. Let S be the current set of tags in use and for $j \in \mathbb{N}$, take $S_j = S \cap U_j$. For $j \in \mathbb{N}$, we call $|S_j| + |S_{j-1}|$ the *pair size* of j (take $S_0 = \emptyset$), and we say that S_j is *full* if $S_j = U_j$.

Let $J = \{j \in \mathbb{N} \mid S_j \neq \emptyset\}$. For all $j \in J$ and all tags $x \in S_j$, the element with (external) tag x is stored with an (internal) tag of $x - (2^{j-1} - 1)$ in a separate data structure D_j called a *data stripe* and constructed according to one of Lemmas 1–4. The offset $o_j = 2^{j-1} - 1$ applied to tags in D_j is called for by our convention of letting all tag ranges start at 1. For $j \in J$ and $i = 1,\ldots,4$, if D_j is constructed as described in Lemma i, we will say that D_j is (stored) in *Representation* i. All data stripes are stored in an initial block of a semi-infinite array $\mathcal{D}$. The first cell of $\mathcal{D}$ following this block is pointed to by a variable *top*. When stating that a data stripe D_j is *moved to* $\mathcal{D}$, what we mean is that it is stored in $\mathcal{D}$, starting at the address *top*, after which *top* is incremented by the size of D_j. *Converting* a data stripe D_j *to* Representation i means extracting all (tag, element) pairs stored in D_j by means of its scan operation, abandoning the old D_j (which may still continue to occupy a segment of $\mathcal{D}$), initializing a new D_j in Representation i with the pairs extracted, and moving (the new) D_j

to $\mathcal{D}$. Every data stripe is *marked* or *unmarked*, the significance of which will be explained shortly.

Another main component of the quasidictionary is a dictionary H that maps each $j \in J$ to the starting address in $\mathcal{D}$ of (the current) D_j. All changes to $\mathcal{D}$ are reflected in H through appropriate calls of its operations; this will not be mentioned explicitly on every occasion. Following the first check-in operation, we also keep track of $b = \lfloor \log_2 N \rfloor + 1$, where N is the maximum number of tags simultaneously in use since the initialization. By assumption, we always have $b \le w$, and b never decreases. The main components of the quasidictionary are illustrated in Fig. 2.

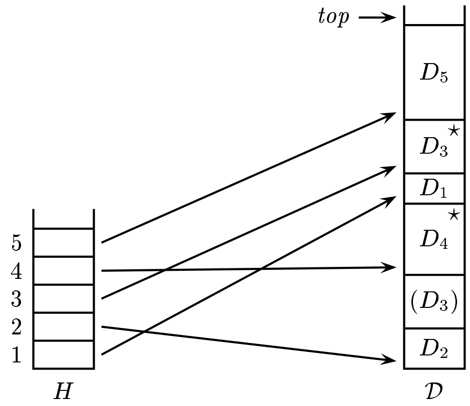

Fig. 2. The array $\mathcal{D}$ that holds the data stripes and a symbolic representation of the dictionary H that stores their starting addresses. A star denotes a marked data stripe, and an abandoned data stripe is shown with parentheses.

3.1 access

To execute access(x), we first determine the value of j such that $x \in U_j$, which is the same as computing $\mathrm{MSB}(x)+1$. Then the starting address of D_j is obtained from H, and the operation access$(x - o_j)$ is executed on D_j.

As explained in the proof of Lemma 4, $\mathrm{MSB}(x)$ can be computed from x in constant time, provided that certain quantities $f_1(b), \ldots, f_\nu(b)$ depending on b are available. We assume that $f_1(b), \ldots, f_\nu(b)$ are stored as part of the quasidictionary and recomputed at every change to b.

3.2 check-in

To execute check-in(v), we first compute the smallest positive integer j such that S_j is nonfull. For this purpose we maintain a bit vector q whose bit in position $b - i$ is 1 if and only if S_i is nonfull, for $i = 1, \ldots, b$, so that $j = b - \mathrm{MSB}(q)$.

Note that, by assumption, $|S|$ remains bounded by $2^b - 1$, which implies that q is always nonzero just prior to the execution of a check-in operation.

If $j \notin J$, we create a new, empty, and unmarked data stripe D_j with capacity 2^{j-1} in Representation 1, move it to $\mathcal{D}$ (recall that this means placing it at the end of the used block of $\mathcal{D}$, recording its starting address in H, and updating *top*), execute the operation check-in(v) on D_j, and return the tag obtained from D_j plus o_j to the caller.

If $j \in J$, we locate D_j as in the case of access. If D_j is not stored in Representation 1, we convert it to that representation with capacity 2^{j-1}, execute the operation check-in(v) on (the new) D_j, and return the tag obtained from D_j plus o_j to the caller.

3.3 check-out

To execute check-out(x), we first determine the value of j such that $x \in U_j$ as in the case of access(x). Operating on D_j, we then execute check-out($x - o_j$) if D_j is in Representation 1 and delete($x - o_j$) otherwise. For $i \in \{j, j+1\} \cap J$, if the pair size of i has dropped below $1/4$ of its value when D_i was (re)constructed, we mark D_i. Finally we execute a *global cleanup* (see below) if the number of check-out operations executed since the last global cleanup (or, if no global cleanup has taken place, since the initialization) exceeds the number of tags currently in use.

3.4 Global Cleanup

A global cleanup clears $\mathcal{D}$ after copying it to temporary storage, sets *top* $= 0$, and then processes the data stripes one by one. If a data stripe D_j is empty (i.e., $S_j = \emptyset$), it is discarded. Otherwise, if D_j is not marked, it is simply moved back to $\mathcal{D}$. If D_j is marked, it is unmarked and moved to $\mathcal{D}$ after conversion to Representation i, where

$$i = \begin{cases} 2, & \text{if } 2^j/j \le |S_j|, \\ 3, & \text{if } 2^{j/5} \le |S_j| < 2^j/j, \\ 4, & \text{if } \qquad\quad |S_j| < 2^{j/5}. \end{cases}$$

3.5 The Dictionary H

The dictionary H that maps each $j \in J$ to the starting address h_j of D_j in $\mathcal{D}$ is implemented as follows: The addresses h_j, for $j \in J$, are stored in arbitrary order in the first $|J|$ locations of a semi-infinite array $\mathcal{H}$. When a new element j enters J, h_j is stored in the first free location of $\mathcal{H}$, and when an element j leaves J, h_j is swapped with the address stored in the last used location of $\mathcal{H}$, which is subsequently declared free. This standard device ensures that the space used in $\mathcal{H}$ remains bounded by $|S|$.

What remains is to implement a dictionary H' that maps each $j \in J$ to the position d_j of h_j in $\mathcal{H}$, a nonnegative integer bounded by $b - 1$. We do this in a way reminiscent of Representation 2. Let l be a power of two with $\log b < l \leq b$, but $l = O(\log b)$. We assume that l and its logarithm are recomputed at every change to b.

When $|J|$ grows beyond $2l$, H' is converted to a representation as an array, indexed by $1, \ldots, b$ and stored in at most $2l \leq |J|$ words, each of which contains $\lfloor b/l \rfloor$ fields of l bits each.

When $|J|$ decreases below l, H' is converted to a representation in which the elements of J are stored in the rightmost $|J|$ fields of $l + 1$ bits each in a single word X (or a constant number of words), and the corresponding positions in $\mathcal{H}$ are stored in the same order in the rightmost $|J|$ fields of $l + 1$ bits in another word Y. Given $j \in J$, the corresponding position d_j can be computed in constant time using low-level subroutines that exploit the instruction set of the word RAM and are discussed more fully in [1]. First, multiplying j by the word 1_{l+1} with a 1 in every field yields a word with j stored in every field. Incrementing this word by 1_{l+1}, shifted left by l bits, changes to 1 the leftmost bit, called the *test bit*, of every field. Subtracting X from the word just computed leaves in the position of every test bit a 1 if and only if the value i of the corresponding field in X satisfies $i \leq j$. A similar computation carries out the test $i \geq j$, after which a bitwise AND singles out the unique field of X containing the value j. Subtracting from the word just created, which has exactly one 1, a copy of itself, but shifted right by l positions, we obtain a mask that can be used to isolate the field of Y containing d_j. This field still has to be brought to the right word boundary. This can be achieved through multiplication by 1_{l+1}, which copies d_j to the leftmost field, right shift to bring that field to the right word boundary, and removal of any spurious bits to the left of d_j. It can be seen that H' uses $O(|S|)$ space and can execute access, insert, and delete in constant time.

The description so far implicitly assumed b to be a constant. Whenever b increases, the representation of H and the auxiliary quantity 1_{l+1} are recomputed from scratch.

4 Analysis

The data structure described in the previous section can clearly be initialized in constant time. We will show that it works in linear space, that it executes access in constant time and check-in and check-out in constant amortized time, and that every tag is bounded by the maximum number of tags in use since the initialization.

4.1 Space Requirements

In this subsection we show the space taken up by every component of the quasidictionary to be linear in the number of tags currently in use. As concerns the dictionary H, this was already observed in the discussion of its implementation, so we can restrict attention to $\mathcal{D}$. The size of the used part of $\mathcal{D}$ is given by the value of *top*, so our goal is to show that $top = O(|S|)$ holds at all times.

Define an *epoch* to be the part of the execution from the initialization or from just after a global cleanup to just after the next global cleanup or, if there are no more global cleanups, to the end of the execution. Let $n = |S|$ and, for all $j \in \mathbb{N}$, denote $|S_j|$ by n_j and the pair size $|S_j| + |S_{j-1}|$ of j by p_j.

Let $j \geq 2$ and $i \in \{1, \ldots, 4\}$ and consider a (re)construction of D_j in Representation i at the time at which it happens. If $i = 1$, the size of D_j is $O(2^j)$ and $p_j \geq 2^{j-2}$. If $i = 2$, the size of D_j is $O(n_j + 2^{j-1}/(j-1))$, which is $O(n_j) = O(p_j)$ because Representation 2 is chosen only if $n_j \geq 2^j/j$. If $i \in \{3, 4\}$, the size of D_j is $O(n_j) = O(p_j)$. Thus, in all cases, when D_j is (re)constructed, its size is $O(p_j)$. Moreover, since D_j is marked as soon as p_j drops below $1/4$ of its value at the time of the (re)construction of D_j, the assertion that the size of D_j is $O(p_j)$ remains true as long as D_j is unmarked. This shows that at the beginning of every epoch, where every data stripe is unmarked, we have $top = O(\sum_{j=1}^{\infty} p_j) = O(n)$.

Consider a fixed epoch, let σ be the sequence of operations executed in the epoch, and let σ' be the sequence obtained from σ by removing all check-out operations. We will analyze the execution of σ' starting from the same situation as σ at the beginning of the epoch and use primes ($'$) to denote quantities that pertain to the execution of σ'. Note that n' and $p'_1, p'_2, \ldots$ never decrease.

During the execution of σ', the used part of $\mathcal{D}$ grows only by data stripes D_j in Representation 1, and there is at most one such data stripe for each value of j. Consider a particular point in time t during the execution of σ'. For each $j \geq 2$ for which a data stripe D_j was moved to $\mathcal{D}$ before time t in the epoch under consideration, the size of this newly added D_j is within a constant factor of the value of p'_j at the time of its creation and also, since p'_j is nondecreasing, at time t. At time t, therefore, the total size of all data stripes added to $\mathcal{D}$ during the execution of σ' is $O(\sum_{j=1}^{\infty} p'_j) = O(n')$. By the analysis above, the same is true of the data stripes present in $\mathcal{D}$ at the beginning of the epoch, so $top' = O(n')$ holds throughout.

Consider now a parallel execution of σ and σ', synchronized at the execution of every operation in σ'. It is easy to see that check-out operations never cause more space on $\mathcal{D}$ to be used, i.e., $top \leq top'$ holds at all times during the epoch. We may have $n < n'$. Since the epoch ends as soon as $n' - n > n$, however, we always have $n' \leq 2n + 2$. At all times, therefore, $top \leq top' = O(n') = O(n)$.

4.2 Execution Times

access operations clearly work in constant time and leave the data structure unchanged (except for changes to elements outside of their tags), so they will be ignored in the following. We will show that check-in and check-out operations take constant amortized time by attributing the cost of maintaining the quasidictionary to check-in and check-out operations in such a way that every operation is charged for only a constant amount of computation.

Whenever b increases, various quantities depending on b as well as the representation of H are recomputed. This incurs a cost of $O(b)$ that can be charged to $\Omega(b)$ check-in operations since the last increase of b or since the initialization.

When $|J|$ rises above $2l$ or falls below l, H is converted from one representation to another. The resulting cost of $O(l)$ can be charged to the $\Omega(l)$ check-in or check-out operations since the last conversion of H or since the initialization.

For $j \in \mathbb{N}$, define a *j-phase* as the part of the execution from just before a (re)construction of D_j in Representation 1 to just before the next such (re)construction or, if there is none, to the end of the execution. Consider a particular j-phase for some $j \geq 2$. The cost of the (re)construction of D_j in Representation 1 at the beginning of the phase is $O(2^j)$. Since the pair size of j at that point is bounded by 2^j and D_j is not reconstructed until the pair size of j has dropped below 1/4 of its initial value, Lemma 2 shows the total cost of all reconstructions of D_j in Representation 2 in the j-phase under consideration to be

$$O\Big(\sum_i \big(4^{-i} \cdot 2^j + 2^{j-1}/(j-1)\big)\Big),$$

where the sum extends over all $i \geq 1$ with $4^{-i} \cdot 2^j \geq 2^j/j$. The number of such i being $O(\log j)$, the sum evaluates to $O(2^j)$. Likewise, by Lemmas 3 and 4, the total cost of all reconstructions of D_j in Representations 3 and 4 are

$$O\Big(\sum_{i=0}^{\infty} \big(4^{-i} \cdot (2^j/j) \cdot \log(2^j/j)\big)\Big) = O(2^j) \quad \text{and}$$

$$O\Big(\sum_{i=0}^{\infty} \big(4^{-i} \cdot 2^{j/5}\big)^5\Big) = O(2^j),$$

respectively. Thus the total cost of all (re)constructions of D_j in a j-phase is $O(2^j)$. For the first j-phase, we can charge this cost to 2^{j-2} check-in operations that filled S_{j-1}. For every subsequent j-phase, the pair size of j dropped below 1/4 of its initial value during the previous phase, hence to at most $(1/4)(2^{j-1} + 2^{j-2}) = (3/4) \cdot 2^{j-2}$. Since S_{j-1} was full again at the beginning of the phase under consideration, we can charge the cost of $O(2^j)$ to the intervening at least $(1/4) \cdot 2^{j-2}$ check-in operations that refilled S_{j-1}. The arguments above excluded the case $j = 1$, but the cost incurred in (re)constructing D_1 once is $O(1)$ and can be charged to whatever operation is under execution.

At this point, the only execution cost not accounted for is that of copying unmarked data stripes back to $\mathcal{D}$ during a global cleanup. By the analysis of the previous subsection, this cost is $O(|S|)$, and it can be charged to the at least $|S|$ check-out operations executed since the last global cleanup.

4.3 Tag Sizes

Consider the point in time just after the quasidictionary issues a tag x and suppose that no tag of x or larger was issued earlier and that $x \in U_j$. Then D_j is in Representation 1 and just issued the tag $x - o_j$, and neither D_j nor an earlier data stripe with index j (an "earlier incarnation" of D_j) ever before issued a tag of $x - o_j$ or larger. The last sentence of Lemma 1 can be seen to imply that

D_j has $x - o_j$ tags in use. Moreover, by the implementation of check-in, all of $S_1, \ldots, S_{j-1}$ are full, so that they contribute an additional $1+2+\cdots+2^{j-2} = o_j$ tags in use. Thus the overall quasidictionary has x tags in use after issuing x. It is now easy to see that the quasidictionary never uses a tag larger than the maximum number of tags in use since its initialization.

As an aside, we note that the binary representation of every tag issued by the quasidictionary consists of exactly as many bits as that of the smallest free tag.

5 Deamortization

In this section we show how the constant amortized time bound for check-in operations can be turned into a constant worst-case time bound. Since the deamortization techniques used are standard, we refrain from giving all details.

In addition to computation that is covered by a constant time bound, a check-in operation, in our formulation to this point, may carry out one or more of the following:

- After an increase in b, an $O(b)$-time computation of various quantities depending on b;
- After an increase in $|J|$ beyond $2l$, an $O(l)$-time conversion of the dictionary H to its array representation;
- A (re)construction of a data stripe in Representation 1.

The computation triggered by an increase in b is easy to deamortize. For concreteness, consider the calculation of $f_1(b), \ldots, f_\nu(b)$. Following an increase in b, rather than calculating $f_1(b), \ldots, f_\nu(b)$, we anticipate the next increase in b and start calculating $f_1(b+1), \ldots, f_\nu(b+1)$. The calculation is interleaved with the other computation and carried out piecemeal, $O(1)$ time being devoted to it by each subsequent check-in operation. Since the entire calculation needs $O(b)$ time, while at least 2^b check-in operations are executed before the next increase in b, it is easy to ensure that $f_1(b+1), \ldots, f_\nu(b+1)$ are ready when they are first needed.

The conversion of H to its array representation is changed to begin as soon as $|J|$ exceeds l. While the conversion is underway, queries to H are answered using its current representation, but updates are executed both in the current representation and in the incomplete array representation, so that an interleaved computation that spends constant time per check-in operation on the conversion can deliver an up-to-date copy of H in the array representation when $|J|$ first exceeds $2l$—because of the simplicity of the array representation, this is easily seen to be possible. Provided that any incomplete array representation of H is abandoned if and when $|J|$ decreases below l, a linear space bound continues to hold.

For $j \geq 2$, the analysis in Section 4 already charged the cost of (re)constructing a data stripe D_j in Representation 1 to at least $\frac{1}{4} \cdot 2^{j-2}$ check-in operations that (re)filled S_{j-1}. We now let an interleaved construction actually take place as part of the execution of these check-in operations. In other words, every check-in operation that issues a tag from a data stripe D_{j-1} that is at least 3/4 full also spends constant time on the (re)construction of D_j in Representation 1, in such a way as to let the new D_j be ready before S_{j-1} is full. As above, updates to D_j that occur after the (re)construction has begun (these can only be deletions) are additionally executed on the incomplete new D_j. Provided that global cleanups remove every partially constructed data stripe D_j for which $|S_{j-1}| < \frac{3}{4} \cdot 2^{j-2}$, it is easy to see that a linear space bound and a constant amortized time bound for check-out operations continue to hold.

6 Conclusions

We have described a deterministic quasidictionary that works in linear space and executes every operation in constant amortized time. However, our result still leaves something to be desired.

First, we would like to deamortize the data structure, turning the amortized time bound for deletion into a worst-case bound. Barring significant advances in the construction of static dictionaries, it appears that techniques quite different from ours would be needed to attain this goal. To see this, consider an operation sequence consisting of a large number of calls of check-in followed by as many calls of check-out. During the execution of the check-out operations, the need to respect a linear space bound presumably requires the repeated construction of a static dictionary for the remaining keys. The construction cannot be started much ahead of time because the set of keys to be accommodated is unknown at that point. Therefore, if it needs superlinear time, as do the best constructions currently known, some calls of check-out will take more than constant time.

Second, one might want to eliminate the use of multiplication from the operations. However, by a known lower bound for the depth of unbounded-fanin circuits realizing static dictionaries [2, Theorem C], this cannot be done without weakening our time or space bounds or introducing additional unit-time instructions.

Finally, our construction is not as practical as one might wish, mainly due to the fairly time-consuming (though constant-time) access algorithm. Here there is much room for improvement.

References

1. A. Andersson, T. Hagerup, S. Nilsson, and R. Raman, Sorting in linear time?, *J. Comput. System Sci.* **57** (1998), pp. 74–93.
2. A. Andersson, P. B. Miltersen, S. Riis, and M. Thorup, Static dictionaries on AC^0 RAMs: Query time $\Theta(\sqrt{\log n/\log\log n})$ is necessary and sufficient, Proc., 37th Annual IEEE Symposium on Foundations of Computer Science (FOCS 1996), pp. 441–450.
3. A. Andersson and M. Thorup, Tight(er) worst-case bounds on dynamic searching and priority queues, Proc., 32nd Annual ACM Symposium on Theory of Computing (STOC 2000), pp. 335–342.

4. P. Beame and F. E. Fich, Optimal bounds for the predecessor problem, Proc., 31st Annual ACM Symposium on Theory of Computing (STOC 1999), pp. 295–304.
5. M. R. Brown and R. E. Tarjan, A representation for linear lists with movable fingers, Proc., 10th Annual ACM Symposium on Theory of Computing (STOC 1978), pp. 19–29.
6. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms* (1st edition), The MIT Press, Cambridge, MA, 1990.
7. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (2nd edition), The MIT Press, Cambridge, MA, 2001.
8. E. D. Demaine, A threads-only MPI implementation for the development of parallel programs, Proc., 11th International Symposium on High Performance Computing Systems (HPCS 1997), pp. 153–163.
9. 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** (1994), pp. 738–761.
10. M. L. Fredman and R. E. Tarjan, Fibonacci heaps and their uses in improved network optimization problems, *J. ACM* **34** (1987), pp. 596–615.
11. M. L. Fredman and D. E. Willard, Surpassing the information theoretic bound with fusion trees, *J. Comput. System Sci.* **47** (1993), pp. 424–436.
12. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, New York, 1979.
13. J. Gergov, Algorithms for compile-time memory optimization, Proc., 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1999), pp. 907–908.
14. L. J. Guibas and R. Sedgewick, A dichromatic framework for balanced trees, Proc., 19th Annual IEEE Symposium on Foundations of Computer Science (FOCS 1978), pp. 8–21.
15. T. Hagerup, Sorting and searching on the word RAM, Proc., 15th Annual Symposium on Theoretical Aspects of Computer Science (STACS 1998), Lecture Notes in Computer Science, Springer-Verlag, Berlin, Vol. 1373, pp. 366–398.
16. T. Hagerup, P. B. Miltersen, and R. Pagh, Deterministic dictionaries, *J. Algorithms* **41** (2001), pp. 69–85.
17. S. Huddleston and K. Mehlhorn, A new data structure for representing sorted lists, *Acta Inf.* **17** (1982), pp. 157–184.
18. M. G. Luby, J. Naor, and A. Orda, Tight bounds for dynamic storage allocation, Proc., 5th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1994), pp. 724–732.
19. K. Mehlhorn and S. Näher, *LEDA: A Platform for Combinatorial and Geometric Computing,* Cambridge University Press, 1999.
20. R. Raman, Priority queues: Small, monotone and trans-dichotomous, Proc., 4th Annual European Symposium on Algorithms (ESA 1996), Lecture Notes in Computer Science, Springer-Verlag, Berlin, Vol. 1136, pp. 121–137.
21. J. M. Robson, An estimate of the store size necessary for dynamic storage allocation, *J. ACM* **18** (1971), pp. 416–423.
22. J. M. Robson, Bounds for some functions concerning dynamic storage allocation, *J. ACM* **21** (1974), pp. 491–499.
23. M. Thorup, On RAM priority queues, *SIAM J. Comput.* **30** (2000), pp. 86–109.
24. P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, Dynamic storage allocation: A survey and critical review, Proc., International Workshop on Memory Management (IWMM 1995), Lecture Notes in Computer Science, Springer-Verlag, Berlin, Vol. 986, pp. 1–116.