A compact data structure for representing a dynamic multiset

Jyrki Katajainen [a,1], S. Srinivasa Rao [b,*]

[a] *Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen East, Denmark*

[b] *School of Computer Science and Engineering, Seoul National University, 599 Gwanakro, Gwanak-Gu, Seoul 151-744, S. Korea*

Abstract

We develop a data structure for maintaining a dynamic multiset that uses $O(n \lg\lg n / \lg n)$ bits and $O(1)$ words, in addition to the space required by the n elements stored, supports searches in $O(\lg n)$ worst-case time and updates in $O(\lg n)$ amortized time. Compared to earlier data structures, we improve the space requirements from $O(n)$ bits to $O(n \lg\lg n / \lg n)$ bits, but the running time of updates is amortized, not worst-case.

Key words: Data structures, space efficiency, memory management, dictionaries, balanced search trees

1 Introduction

A *dictionary* is a classical data structure that stores a (multi)set of elements, and supports searches and updates. To define the dictionary operations rigorously, we use the locator abstraction discussed in (5). A *locator* is a mechanism for maintaining the association between an element and its current position in a data structure. A locator follows its element even if the element changes its position inside the data structure.

* Corresponding author.
Email addresses: jyrki@diku.dk (Jyrki Katajainen), ssrao@cse.snu.ac.kr (S. Srinivasa Rao).

[1] Partially supported by the Danish Natural Science Research Council under contract 272-05-0272 (project "Generic programming—algorithms and tools").

For two elements x and y in multiset S, we define that $x \prec y$ if either x is smaller than y, or if x is equal to y and x was inserted into S before y. This defines a total order on S. For a multiset S, we define the dictionary operations as follows. The first two operations are collectively called *element-based searches*, the next two *locator-based searches*, and the last two *updates*.

lower-bound(S, e)**:** Return a locator to the first element of S that is not less than element e. If no such element exists, return a *null* locator.
upper-bound(S, e)**:** Return a locator to the first element of S that is greater than element e. If no such element exists, return a *null* locator.
successor(S, x)**:** Return a locator to the next element in S (with respect to $\prec$), or a *null* locator if locator x refers to the last element of S.
predecessor(S, x)**:** Return a locator to the previous element in S (with respect to $\prec$), or a *null* locator if locator x refers to the first element of S.
insert(S, e)**:** Add a new element e to S.
delete(S, x)**:** Remove the element with locator x from S.

One side-effect of defining the operations with respect to the insertion order (among elements with the same key) is that the *stability property* can be maintained with the dictionary. Using this dictionary one can also support searching and deleting an element with a given insertion rank.

We separate the maintenance of the elements and their locators by storing the elements in a dictionary and locators in a *quasi-dictionary* (9). Each time an element is moved within the dictionary, the corresponding pointer to that element is updated in the quasi-dictionary. We assume that the number of locators simultaneously in use is a constant and we implement the quasi-dictionary as a list. Since the time and space overhead caused by the quasi-dictionary is negligible, we disregard it hereinafter.

As the model of computation, we use the standard *word RAM* (7) with words of w bits each. We number the words from 0 to $2^w - 1$, and count the space usage as one plus the highest-numbered word currently in use. We assume that $w = \Omega(\lg n)$, and hence a pointer to a data structure of size n can be stored using $O(1)$ words [2]. We also assume that the elements stored in the dictionary are of equal size, $O(1)$ words each, and that element comparisons take $O(1)$ time. Under these assumptions, the running time of an algorithm is simply the number of word operations to be executed.

An *implicit dictionary* is one that requires only $O(1)$ words of extra storage in addition to the elements stored. Recently, an optimal implicit dictionary was developed (3) that supports searches and updates in $O(\lg n)$ worst-case time. But the method does not work for multisets in general. More specifically, if we are not interested in supporting deletion of elements by insertion rank, then

[2] We use $\lg n$ to denote $\max\{1, \log_2 n\}$.

one can easily modify the implicit dictionary structure of Franceschini and Grossi (3) to maintain a multi set with $O(1)$ additional words and support searches in $O(\lg n)$ worst-case time and updates in $O(\lg n)$ amortized time. But there is no easy way to extend their implicit dictionary to support both searches and deletions of elements by insertion rank. In fact, in the special case where all the elements in the dictionary have the same key, this problem reduces to the *memory management* problem which we address in the paper. It is unlikely that even this special case of the problem can be solved in optimal time with $O(1)$ additional space (i.e., in the implicit model).

Brönnimann et al. (1) designed a structure that maintains a multiset using $O(n/\lg n)$ words (or $O(n)$ bits if $w = \Theta(\lg n)$) in addition to the n elements stored, and supports element-based searches and updates in $O(\lg n)$ worse-case time, and locator-based searches in $O(1)$ worst-case time. Munro and Suwanda (10) called this type of data structures *semi-implicit*, since they use $o(n)$ words of extra space.

Any balanced search tree can be used to maintain a multiset in linear space (measured in words), and to support searches and updates in worst-case logarithmic time. The performance of the best space-efficient data structures is given in the following two lemmas. Lemma 1 draws inspiration from the work of Frederickson (4) and Lemma 2 is proved by Brönnimann et al. (1).

Lemma 1 *Let n denote the number of elements stored just prior to a dictionary operation. A semi-implicit dictionary representing a dynamic multiset can be maintained using an additional $O(\sqrt{n})$ pointers of $\lg n$ bits each, apart from an array of size n storing the elements, to support element-based searches in $O(\lg n)$ worse-case time, locator-based searches in $O(1)$ worst-case time, and updates in $O(\sqrt{n})$ worst-case time.*

Proof. We use a two-level rotated array (called a two-level rotated list in (4)) to store the elements, with additional pointers to point to the beginning of each rotated array. A *rotated array* is an arbitrary cyclic shift of a sorted array. In a *two-level rotated array*, the elements are stored in an array divided into $\lceil\sqrt{2n}\rceil$ blocks, where the ith block is a rotated array of length i. All the elements in the ith block are less than or equal to any element in the $(i+1)$st block. We store the starting position of each block (rotated array) explicitly.

To search for an element e, we first perform a binary search to find two adjacent blocks where the first (last) occurrence of e can lie. We then perform a binary search in those two blocks to find the answer. Thus element-based searches can be supported in $O(\lg n)$ worst-case time. The locator of an element can be realized by recording the block in which the element is stored together with the offset from the beginning of the block. Given a locator, the locator to its predecessor or successor can be easily obtained in $O(1)$ worst-case time.

To insert an element e, we first find the location where e should be inserted (using *upper-bound*). We perform the insertion into the block by removing the last element of the rotated array, and shifting all the elements between the new element and the last element, one position forward. We then have to update every block following the block in which the insertion was made as follows: remove the last element of the current rotated array and insert the last element from the previous rotated array which now becomes the new first element of the rotated array. Also, we update the starting position of each block in $O(1)$ time per block. Thus, insertions can be supported in $O(\sqrt{n})$ worst-case time. Deletions can be performed analogously. □

Lemma 2 (1, Theorem 4) *Let n denote the number of elements stored just prior to a dictionary operation. A semi-implicit dictionary representing a dynamic multiset can be maintained using an additional $O(n/\lg n)$ words, apart from the space required to store the elements, to support searches and updates in $O(\lg n)$ worst-case time. The whole data structure can be maintained in a contiguous segment of memory.*

In this article, we develop a more compact semi-implicit dictionary that uses $O(n \lg\lg n/\lg n)$ extra bits and $O(1)$ extra words, and supports searches in $O(\lg n)$ worst-case time and updates in $O(\lg n)$ amortized time.

2 Memory management

Assume that we manipulate equal-sized records, each consisting of $O(1)$ words. As a useful abstraction, we define an *extendible array* to be a data structure which stores n such records, each assigned a unique index between 0 and $n-1$, and supports the following operations. The last two operations are collectively called *updates*.

- *access*(A, i): access the record with index i in A (for reading or writing),
- *grow*(A): allocate space for a new record with index n in A, and increase n by one, and
- *shrink*(A): decrease n by one, and free space reserved for the record with index n in A.

We consider the problem of maintaining a collection of extendible arrays under the following operations:

- *create*(): create a new empty extendible array,
- *destroy*(A): destroy the empty array A, and
- *access*(A, i), *grow*(A), *shrink*(A), which are as above.

Hagerup et al. (8) showed how to maintain a collection of extendible arrays in $O(n)$ words of memory such that all operations are supported in $O(1)$ worst-case time, where n is a pre-specified upper bound in the total size of all the extendible arrays. This result holds under the assumption that an upper bound on the total size of all extendible arrays is known beforehand. The space bound can be made almost optimal by allowing updates to take $O(1)$ amortized time, as shown in (11).

In the special case, where all the extendible arrays are small, a simple space-efficient solution, which is fast enough for our purposes, is obtained by relying on the *compactor zone* technique of (2):

Lemma 3 *Let b be an integer parameter, and S a collection of k extendible arrays whose sizes are at most b. Let s be the total size of all the extendible arrays, measured in words. We can maintain S in a single extendible array of $s + O(b + k)$ words, while supporting access, create and destroy in $O(\lg k)$ worst-case time, and grow and shrink in $O(b + \lg k)$ worst-case time.*

Proof. We assume that the names of extendible arrays are time stamps that can be stored in a single word. We use a dictionary to maintain this dynamic set of names. Each name is associated with the length and starting position of the corresponding extendible array. We need to update the associated satellite data efficiently when an extendible array is relocated. For this, we store along with the contents of each extendible array a backpointer pointing back to the associated information on that particular array in the dictionary.

For $j \in \{1, 2, \ldots, b\}$, all the extendible arrays of size j are grouped together to form the jth *zone*. We store the zones in the increasing order of j from left to right. All the arrays that belong to the jth zone are stored consecutively in arbitrary order, except at most one *rotated array* for which the first i elements are stored at the end of the zone, and the remaining $j - i$ elements at the beginning of the zone. We store pointers to the beginning and end of each zone in a separate table. Additionally, we store the length of the suffix of the rotated array held at the beginning of the jth zone, for $j \in \{1, 2, \ldots, b\}$.

We must rely on a dictionary implementation that allows individual nodes to be relocated. For example, a red-black tree (6) would be suitable for our purposes. The main requirement relevant for us is that each node has constant in-degree and out-degree, and that each node knows which nodes are pointing to it. When a node is moved, only a constant number of pointers have to be updated, so a node movement takes $O(1)$ worst-case time.

We maintain the allocated memory in three parts. In the first part we store the fixed-sized table containing information on zone boundaries, in the second part the b rotating zones of elements, and in the third part a pool of dictionary nodes. Since in these three parts the record sizes may be different, there can

be a gap of $O(1)$ words between the parts. Space for a new node is allocated from the end of the memory pool and, when a node is freed, the last node is moved to its place. Since nodes can be relocated freely, the pool of nodes can be rotated along the zones, whenever necessary.

Both allocation and destruction of nodes take $O(1)$ time. Thus, due to the efficiency of dictionary updates, *create* and *destroy* can be supported in $O(\lg k)$ worst-case time (since the dictionary is a red-black tree on k elements). To support *access*(A), we first search the dictionary to identify the boundaries of A. Given the boundaries, *access* can be easily accomplished in constant additional time. Thus the overall access time is $O(\lg k)$ in the worst case.

To perform *grow*(A), we first find the boundaries of A using the dictionary as before. Suppose that A belongs to the jth zone. If A is the rotated array in the jth zone, we take the suffix of the last unrotated array (if any) in the same zone and swap that suffix with the suffix of A at the beginning of the zone. This makes the elements of A consecutive, and to get the elements in correct order we swap the suffix and the prefix of A. If A is unrotated, we swap it with the last unrotated array in the jth zone. Then we interchange the prefix of the rotated array (if any) in the jth zone and the whole A, and update the boundary of the jth zone. To make space for a new element, we rotate the zones from $j+1$ to b by one element to the right. We can now make A one larger. To move A into a proper position in the $(j+1)$th zone, we interchange the suffix of the rotated block in the $(j+1)$th zone (if any) and the grown A. Whenever we move or rotate an extendible array, we also update the corresponding satellite information in the dictionary using the backpointers stored in connection with the corresponding extendible array.

Shrinks are performed in a similar way by rotating the zones to the left. Both grows and shrinks touch at most $O(b)$ elements so the overall running time is $O(b + \lg k)$ for both in the worst case. □

Given an extendible array A of size n and a positive integer a, we define the bulk updates as follows:

- *bulk-grow*$_a(A)$: create space for a new records with indices from n to $n+a-1$ in A, and increase n by a, and
- *bulk-shrink*$_a(A)$: decrease n by a, and free space reserved for the a records with indices n to $n+a-1$ in A. If $a > n$, this operation is undefined.

To be able to handle larger extendible arrays, we need the following lemma that operates on bulk updates.

Lemma 4 *Let c be an integer parameter, and S a collection of k extendible arrays whose sizes are multiples of $\sqrt{c}$ and at most c. Let s be the total size of all the extendible arrays, measured in words. We can maintain S in a sin-*

gle extendible array of $s + O(\sqrt{c} + k)$ *words, while supporting access, create and destroy in* $O(\lg k)$ *worst-case time, and bulk-grow*$_{\sqrt{c}}$ *and bulk-shrink*$_{\sqrt{c}}$ *in* $O(c + \lg k)$ *worst-case time.*

Proof. The proof is analogous to the proof of Lemma 3; single elements are just replaced with chunks of $\sqrt{c}$ elements. Observe that now there are only $\sqrt{c}$ zones. □

3 Our improvement

In this section, we first give a slight modification of (1, Theorem 4) (cf. Lemma 2) and then use it to obtain our amortized solution.

Lemma 5 *Let* S *be a multiset of* n *elements, and* b *a fixed integer parameter. There is a dictionary for* S *that supports element-based searches in* $O(\lg n)$ *worst-case time, locator-based searches in* $O(1)$ *worst-case time, and updates in* $O(\lg n + b)$ *worst-case time; and uses* $O((n \lg n)/b)$ *bits and* $O(1)$ *words of additional space, apart from the space used for storing the* n *elements. The whole data structure can be maintained in a contiguous segment of memory.*

Proof. If $n \leq b$, we maintain the elements in a single extendible array of size n. Clearly, searches can be supported in $O(\lg n)$ time and updates in $O(b)$ time. Let us therefore assume that $n > b$.

We partition S into $\Theta(n/b)$ chunks, and maintain the size of each chunk to be between b and $2b - 1$. The elements within each chunk consist of consecutive elements of S and are maintained in sorted order. In other words, each chunk defines an interval in the element universe and these intervals are non-overlapping, except perhaps at their endpoints. For $1 \leq j \leq b$, a chunk of size $b + j$ is stored as a *full block* containing the first b elements and a *partial block* containing the remaining j elements. We maintain these blocks as extendible arrays using Lemma 3.

For representing S, we maintain a standard dictionary (e.g. a red-black tree) on top of the chunks, and call it the *chunk dictionary*. This dictionary stores pointers to the chunks, and chunks store backpointers to the corresponding position in the dictionary. Each chunk stores pointers to its partial (if any) and full block. Each partial and full block also stores a backpointer to its corresponding chunk. The total space used by the chunk dictionary together with the backpointers is $O((n/b) \lg n)$ bits.

Element-based searches (*lower-bound*, *upper-bound*) can be supported by first finding the chunk containing the query element using the chunk dictionary in

$O(\lg(n/b))$ time, and then searching for the element in the chunk in $O(\lg b)$ time using binary search. The locator of an element must provide a pointer to the chunk storing the element together with the offset within the chunk. Locator-based searches (*successor*, *predecessor*) can be supported in $O(1)$ worst-case time, provided that the chunk dictionary supports these operations in $O(1)$ worst-case time (as is the case for a red-black tree).

When inserting an element e with a given locator, we first find the chunk into which e should be inserted (using the chunk dictionary). If e has to be inserted into the full block (of the chunk), then we insert e into the full block retaining the sorted order. Then we remove the largest element from the full block, and insert it into the corresponding partial block (if there is no corresponding partial block, we create a new partial block of size 1). Thus, in any case we need to perform an insertion into a partial block.

To insert an element e into a partial block B, we first invoke $grow(B)$ to make space for the new element and then insert e into B retaining the sorted order of the elements in B. Finally, if after the insertion the size of the partial block is b, we split the chunk into two new chunks of size b each, and update the chunk dictionary accordingly.

Deletions are performed in a similar way by invoking *shrink* and merging neighbouring chunks if necessary. □

We can reduce the amount of space needed if the running times for updates are amortized. First, we assume that the parameter c used is a fixed value.

Lemma 6 *Let S be a multiset of n elements, and c be a fixed integer parameter. There is a dictionary for S that supports element-based searches in $O(\lg(n/c) + \lg c)$ worst-case time, locator-based searches in $O(1)$ worst-case time, and updates in $O(\lg(n/c) + \sqrt{c})$ amortized time; and uses $O((n \lg c)/\sqrt{c})$ bits and $O(1)$ words of additional space, apart from the space used for storing the n elements. The whole data structure can be maintained in a contiguous segment of memory.*

Proof. As in the proof of Lemma 5, we partition S into $\Theta(n/c)$ chunks, and maintain the size of each chunk between c and $3c - 1$. Within each chunk the elements are organized using the data structure of Lemma 1. Now each chunk stores $O(\sqrt{c})$ pointers each consisting of $\lg c$ bits. So the overall space usage is $O((n/c)\sqrt{c} \lg c)$ bits apart from the elements stored.

For $0 \leq i < 2\sqrt{c} - 1$ and $0 \leq j < 2\sqrt{c}$, a chunk of size $c + i\sqrt{c} + j$ is stored as a *first block* of size c, a *middle block* of size $i\sqrt{c}$, and a *last block* of size j. We maintain all the last blocks using the structure of Lemma 3 with $b = \sqrt{c}$, and all the middle and first blocks using the structure of Lemma 4. According to these lemmas, the extra space used for memory management in both cases is

$O((\sqrt{c}+n/c)\lg n)$ bits. As in Lemma 5, we also maintain a dictionary on top of the chunks. This chunk dictionary stores $O(n/c)$ pointers each consisting of $O(\lg n)$ bits, i.e. it uses $O(n \lg n/c)$ bits in total.

Element-based searches can be supported by first finding the chunk containing the query element using the chunk dictionary in $O(\lg(n/c))$ worst-case time, and then searching for the element within the chunk using the search procedure of Lemma 1 in $O(\lg c)$ worst-case time. Locator-based searches can easily be supported in $O(1)$ worst-case time, assuming the locator knows the chunk where the element pointed to is stored and the offset within that chunk.

An insert begins with a search for the position where the given element should be inserted. Assume that we know the chunk and the location within the chunk. Insertion into the chunk is done as described in Lemma 1 in $O(\sqrt{c})$ worst-case time. A chunk is made one longer by using the tools provided by Lemma 5. Every insertion involves an extension of the compactor zone, meaning $O(\sqrt{c}+\lg(n/c))$ worst-case time per operation. If a middle block reaches the maximum size, we execute a *bulk-grow*$_{\sqrt{c}}$ which requires $O(c+\lg(n/c))$ worst-case time, but it will require $\Omega(\sqrt{c})$ insertions or deletions before the same block is considered again so the amortized cost of this bulk operation is $O(\sqrt{c}+\lg(n/c)/\sqrt{c})$.

If the size of the chunk becomes $3c$, we split the chunk into two halves, the first half containing the first $\lceil 3c/2 \rceil$ elements in sorted order and the second half containing the remaining elements. Then we rebuild the rotated-array structure within the chunks and we also update the chunk dictionary to contain the new chunk. This requires $O(c+\lg(n/c))$ worst-case time, but a split or a join has to be done only after $\Omega(c)$ updates meaning that the amortized cost is $O(1)$. To summarize, the amortized cost of an insertion is $O(\sqrt{c}+\lg(n/c))$. Deletions are symmetric and can be performed within the same time. □

We now prove our main result. In principle, the result is obtained from the previous lemma by letting the parameter c vary as a function of n.

Theorem 7 *Assume that we are given a dictionary D that for a collection of n elements supports element-based searches in $E_D(n)$ worst-case time, locator-based searches in $L_D(n)$ worst-case time, updates in $U_D(n)$ worst-case time, and uses $O(n)$ words of extra space in addition to the elements themselves. Furthermore, assume that functions E_D, L_D, and U_D are increasing. There exists a dictionary that supports element-based searches in $O(E_D(n)+\lg\lg n)$ worst-case time, locator-based searches in $O(L_D(n))$ worst-case time, and updates in $O(E_D(n)+\sum_{i=1}^{\lfloor \lg\lg n \rfloor} U_D(2^{2^i})+U_D(n)+\lg n)$ amortized time; and uses $O(n\lg\lg n/\lg n)$ bits and $O(1)$ words of additional space, apart from the space used for storing the n elements. The whole data structure can be maintained in a contiguous segment of memory.*

Proof. We use Frederickson's partitioning (4) and maintain $O(\lg \lg n)$ portions of doubly-exponentially increasing sizes. The size of the ith portion is fixed to 2^{2^i}, except that the last portion can contain a fewer number of elements. To support locator-based searches, we modify Frederickson's scheme such that, for $i < j$, every element in the ith portion is smaller than or equal to every element in the jth portion.

We organize the elements in each portion as in Lemma 6. In the ith portion the parameter c_i is fixed to 2^{2i}. That is, in the last portion the parameter is proportional to $\lg^2 n$. We use a dictionary of the same type as D to realize the chunk dictionary. Additionally, we maintain for each portion locators to its minimum and maximum elements.

Element-based searches are carried out by locating the portion in which the searched element is stored, and then looking for the element in that portion. Because of the locators to the minimum and maximum elements, the scan over the portions only requires $O(1)$ worst-case time per portion. Due to the choice of the chunk dictionary and the fixed integer parameter, a search inside a portion takes at most $O(E_D(n) + \lg \lg n)$ worst-case time. Thus, element-based searches can be supported in the time bound claimed.

The locators are implemented by recording the portion and the chunk in which the element is stored, together with the offset inside that particular chunk. When this information is available, locator-based searches can just visit the portion and chunk in question, but it may be necessary to access the first or last element of the next or previous chunk or those of the next or previous portion. Since we store locators to the minimum and maximum elements for each portion, locator-based searches for the chunks dominate the overall costs. Thus, all predecessor and successor queries take $O(L_D(n))$ worst-case time.

Each insert is performed by locating the portion into which the new element is to be put. If that particular portion is not the last portion, or if it is the last portion and the last portion is full, the maximum element is removed from that portion to make space for the new element. The removed maximum is inserted into the next portion and the process is repeated until the last portion gets an element. A deletion is similar, but an element is taken from the last portion and inserted into the previous portion, and the process is repeated (if at all necessary) until a replacement element is got to the portion that lost an element. Thus, at most two updates are to be performed at each portion. The claimed time bound follows from this discussion and Lemma 6.

The amount of extra space used by the two largest portions is $O(n \lg \lg n / \lg n)$ bits and $O(1)$ words, and that used by the other portions $o(\sqrt{n})$ bits and $O(1)$ words, as the pointers referring to the data structure itself can be represented with $O(\lg n)$ bits each. □

Corollary 1 *Let n denote the number of elements currently stored. There exists a dictionary that supports element-based searches in $O(\lg n)$ worst-case time, locator-based searches in $O(1)$ worst-case time, and updates in $O(\lg n)$ amortized time; and uses $O(n \lg\lg n / \lg n)$ bits and $O(1)$ words of additional space, apart from the space used for storing the n elements. The whole data structure can be maintained in a contiguous segment of memory.*

Proof. We use Theorem 7 to make a red-black tree (6) compact. □

4 Open problems

We were able to improve the space bound proved in (1) by relying on amortization. However, we do not know how to achieve the same time bounds in the worst case. Also, it is open whether our space bound can be improved.

References

[1] H. Brönnimann, J. Katajainen, and P. Morin, Putting your data structure on a diet, CPH STL Report **2007-1**, Department of Computer Science, University of Copenhagen (2007).

[2] G. Franceschini and R. Grossi, Optimal cache-oblivious implicit dictionaries, *Proceedings of the 30th International Colloquium on Automata, Languages and Programming*, *Lecture Notes in Computer Science* **2719**, Springer-Verlag (2003), 316–331.

[3] G. Franceschini and R. Grossi, Optimal worst-case operations for implicit cache-oblivious search trees, *Proceedings of the 8th International Workshop on Algorithms and Data Structures*, *Lecture Notes in Computer Science* **2748**, Springer-Verlag (2003), 114–126.

[4] G. N. Frederickson, Implicit data structures for the dictionary problem, *Journal of the ACM* **30**, 1 (1983), 80–94.

[5] M. T. Goodrich and R. Tamassia, *Algorithm Design: Foundations, Analysis, and Internet Examples*, John Wiley & Sons, Inc. (2002).

[6] L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts, A new representation for linear lists, *Proceedings of the 9th Annual ACM Symposium on the Theory of Computing*, ACM (1977), 49–60.

[7] T. Hagerup, Sorting and searching on the word RAM, *Proceedinds of the 15th Annual Symposium on Theoretical Aspects of Computer Science*, *Lecture Notes in Computer Science* **1373**, Springer-Verlag (1998), 366–398.

[8] T. Hagerup, K. Mehlhorn, and J. I. Munro, Maintaining discrete probability distributions optimally, *Proceedings of the 20th International Col-*

loquium on Automata, Languages and Programming, *Lecture Notes in Computer Science* **700**, Springer-Verlag (1993), 253–264.

[9] T. Hagerup and R. Raman, An efficient quasidictionary, *Proceedings of the 8th Workshop on Algorithm Theory*, *Lecture Notes in Computer Science* **2368**, Springer-Verlag (2002), 1–18.

[10] J. I. Munro and H. Suwanda, Implicit data structures for fast search and update, *Journal of Computer and System Sciences* **21** (1980), 236–250.

[11] R. Raman and S. S. Rao, Succinct dynamic dictionaries and trees, *Proceedings of the 30th International Colloquium on Automata, Languages and Programming*, *Lecture Notes in Computer Science* **2719**, Springer-Verlag (2003), 357–368.