Scanning and Traversing: Maintaining Data for Traversals in a Memory Hierarchy

Michael A. Bender[1⋆], Richard Cole[2⋆⋆], Erik D. Demaine[3⋆⋆⋆], and Martin Farach-Colton[4†]

[1] Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY 11794-4400, USA. bender@cs.sunysb.edu.
[2] Courant Institute, New York University, 251 Mercer Street, New York, NY 10012, USA. cole@cs.nyu.edu.
[3] MIT Laboratory for Computer Science, 200 Technology Square, Cambridge, MA 02139, USA. edemaine@mit.edu.
[4] Google Inc., 2400 Bayshore Parkway, Mountain View, CA 94043, USA, and Department of Computer Science, Rutgers University, Piscataway, NJ 08855, USA. farach@cs.rutgers.edu.

Abstract. We study the problem of maintaining a dynamic ordered set subject to insertions, deletions, and traversals of k consecutive elements. This problem is trivially solved on a RAM and on a simple two-level memory hierarchy. We explore this traversal problem on more realistic memory models: the cache-oblivious model, which applies to unknown and multi-level memory hierarchies, and sequential-access models, where sequential block transfers are less expensive than random block transfers.

1 Introduction

A basic computational task is to maintain a dynamic ordered set of elements subject to insertions, deletions, and *logical traversals*. By a logical traversal we mean an in-order access of the k elements following an element x, for a given k and x. These three operations are performed by nearly any computer program that uses even the most common data structures, such as linked lists or search trees.

At first glance it does not seem that there is much to study. On a RAM, the problem is trivially solved by the lowly linked-list. Even on a two-level memory hierarchy, as described by the Disk Access Model (DAM model) [1], we can guarantee optimal results: For blocks of size B, we obtain $O(\lceil k/B \rceil)$ memory accesses per traversal and $O(1)$ accesses per insertion and deletion by maintaining $\Theta(B)$ contiguous elements in each block.

⋆ Supported in part by HRL Laboratories, NSF Grant EIA-0112849, and Sandia National Laboratories.
⋆⋆ Supported in part by NSF Grants CCR-9800085 and CCR-0105678.
⋆⋆⋆ Supported in part by NSF Grant EIA-0112849.
† Supported in part by NSF Grant CCR-9820879.

The meat of the problem lies in exploring more realistic memory models. For example, the DAM model makes no distinction between sequential and random-access block transfers. Furthermore, the solution proposed above is sensitive to the parameter B. Can we design a data structure that works well on more than two levels or for all values of B? The main contribution of this paper is a systematic study of the memory-traversal problem under a variety of tradeoffs and assumptions about memory, e.g.: cache aware versus cache oblivious, minimizing accesses versus minimizing seeks, and amortized versus worst case.

Some special cases of the traversal problem have been explored previously. By taking a systematic approach to memory issues—e.g., when does it pay to keep elements in sorted order—we improve bounds for almost all versions of the problem. By proving separations between the versions, we demonstrate the impact of making particular assumptions on the memory hierarchy.

1.1 Hierarchical Memories

The *Disk Access Model* (DAM model), introduced by Aggarwal and Vitter [1], is the standard two-level memory model. In this model, the memory hierarchy consists of an internal memory of size M and an arbitrarily large external memory partitioned into blocks of size B. Memory transfers between the two levels are performed in blocks.

The *cache-oblivious model* enables us to reason about a simple two-level memory model, but prove results about an unknown, multilevel memory model. The idea is to avoid any memory-specific parameterization, that is, to design algorithms that avoid using information about memory-access times or about cache-line and disk-block sizes. Such algorithms are optimal at all levels of the memory hierarchy, because the analysis is not tailored to any particular level.

The cache-oblivious model was introduced by Frigo, Leiserson, Prokop, and Ramachandran [14, 22]. They show how several basic problems—namely matrix multiplication, matrix transpose, Fast Fourier Transform, and sorting—have optimal algorithms that are cache-oblivious. More recently, cache-oblivious data structures matching the bounds of B-trees, priority queues, and tries have been developed [2, 4–7].

Most memory models do not distinguish between random block transfers and sequential block transfers. The difference in access times is caused by the seek times and latencies on disk and by prefetching in disk and main memory. The current difference in speed between random access and sequential access on disk is around a factor of 10 [15], and this factor appears to increase by 5 every two years [23, 21]. We could incorporate these issues into the models by specifying the balance between physical scans and random block access, but this is another unwieldy parameter at each level of the memory hierarchy. Instead, we make the restriction that all operations in the traversal problem involve only a constant number of scans. This restriction can lead to better performance in both the DAM model and the cache-oblivious model. The idea of minimizing scans was also considered by Farach-Colton, Ferragina, and Muthukrishnan [12].

	DAM	Cache-Oblivious
Unsorted, unrestricted block transfers	$O\left(\lceil k/B \rceil\right)$ [well-known]	$O\left(\left\lceil k\frac{(\log\log N)^{2+\varepsilon}}{B}\right\rceil\right)$
Sorted, unrestricted block transfers	$O\left(\left\lceil k\frac{\log^2(N/(k+B))}{B}\right\rceil\right)$ amortized [4] $O\left(\left\lceil k\frac{\log^3(N/(k+B))}{B}\right\rceil\right)$ worst-case	$O\left(\left\lceil k\frac{\log^2(N/k)}{B}\right\rceil\right)$ amortized [4] $O\left(\left\lceil k\frac{\log^3(N/k)}{B}\right\rceil\right)$ worst-case
Sorted, $O(1)$ physical scans	$O\left(\left\lceil \frac{\log^2(N/B)}{B}\right\rceil\right)$	$O\left(\left\lceil \frac{\log^2 N}{B}\right\rceil\right)$

Table 1. Number of memory transfers for a cluster of k updates in a structure supporting fast traversals, amortized except where noted as worst-case. All results are new, except those marked with a reference. Uncaptured by this table are the precise traversal bounds: they are all $O(\lceil k/B \rceil)$ except for the polyloglog cache-oblivious result; in this case, the bound is off by a small factor in the worst case, but holds in the amortized sense.

1.2 Results

We present the first positive progress on the traversal problem in almost two decades (cf. Section 1.3). Results are summarized in Table 1. We consider a variety of settings, depending on whether the memory hierarchy has two levels with a known block size (DAM) or multiple levels with unknown parameters (cache oblivious); whether the block accesses must be in sequential order or may be in arbitrary order; and whether the elements must be kept sorted[5]. One encouraging aspect of the traversal problem is that the update cost is surprisingly low (polylogarithmic or better) in all contexts, especially compared to the naïve update cost of $\Theta(N)$ when the elements are stored in sorted order without gaps.

When the elements must be sorted, the best general lower bound is $\Omega((\log N)/B)$ memory transfers, which follows directly from [9], and the best upper bound was $O((\log^2 N)/B)$ amortized memory transfers for unrestricted block transfers [4, 17, 31]. We make two main contributions in this setting. In Section 4 we extend the upper-bound result to hold even when updates can make only a constant number of physical scans. This restriction is important because it causes block accesses to be mainly sequential. Finally, we strengthen the result in a different direction, achieving even worst-case bounds. This algorithm follows a different, de-amortized strategy, and is more complex. We defer the description of this algorithm to the full paper.

One of our main contributions is to remove the implicit assumption in the literature that elements have to be sorted. Allowing elements to be stored

[5] By keeping elements in sorted order, we mean that their logical order is monotone with respect to their memory addresses. This notion is valuable for algorithms that use memory addresses as proxies for order. See, for example, [6].

out-of-order leads to update bounds exponentially below the lower bound of $\Omega((\log N)/B)$ for the sorted case. In particular, in Section 2 we develop a cache-oblivious structure using $O(\lceil(\log\log N)^{2+\varepsilon}/B^1\rceil)$ unrestricted block transfers, for any $\varepsilon > 0$. This is our only result for which the traversal bound is (slightly) worse than the optimal $\Theta(\lceil k/B\rceil)$; if k is at least $B^{1-\varepsilon}$, then there is an additive B^ε term. However, in Section 3, we show how to modify scans to be self-adjusting (following splay trees [26]) and achieve the optimal $O(\lceil k/B\rceil)$ bound in an amortized sense, while maintaining the near-optimal worst case.

This polyloglog structure uses a weak version of the tall-cache assumption: $M = \Omega(B^\tau)$ for some $\tau > 1$. This assumption is commonly made in other external-memory and cache-efficient algorithms and usually holds in practice.

Our result is an exponential improvement in update cost over the best known solutions for sequential block accesses and for when elements must be in sorted order. An intriguing open problem is to prove a nonconstant lower bound.

A natural question is whether the per-operation performance of a traversal structure can be improved when the updates are batched and possibly clustered around a common element. Although some improvement is possible for large clusters, as shown in Table 1, it seems that this improvement is marginal.

Our results form a body of tools for manipulating dynamic data in unknown and multilevel memory hierarchies. In particular, they can be used to improve cache-oblivious B-trees. The only cache-oblivious B-tree structures that support traversals optimally [4, 6, 7] require $O(\log_B N + \frac{\log^2 N}{B})$ amortized memory transfers per update. By applying the structure in Section 4, we obtain the following improvement:

Corollary 1. *There is a cache-oblivious data structure that maintains an ordered set subject to searches in $O(\log_B N)$ memory transfers, insertions and deletions in $O(\log_B N)$ amortized memory transfers, and traversals of k consecutive elements in $O(\lceil k/B\rceil + [B^\varepsilon$ if k is at least $B^{1-\varepsilon}])$ memory transfers in the worst-case and $O(\lceil k/B\rceil)$ memory transfers in the amortized sense. Furthermore, scanning all N element requires $O(\lceil N/B\rceil)$ worst case memory transfers.*

1.3 Related Work

The main body of related work maintains a set of N elements ordered in $O(N)$ locations of memory, subject to insertions and deletions at specific locations in the order. This *ordered-file maintenance* problem is one version of the traversal problem, in which the elements must be sorted. Most of this work analyzes running time, but these results often adapt to multilevel memory hierarchies.

Itai, Konheim, and Rodeh [17] give a structure using $O(\log^2 N)$ amortized time per update. Similar results were also obtained by Melville and Gries [20] and Willard [29]. Willard [29–31] gives a complicated structure using $O(\log^2 N)$ worst-case time per update. A modification to the structure of Itai et al. results in a cache-oblivious traversal algorithm running in $O((\log^2 N)/B)$ amortized memory transfers per update.

The best lower bounds for this problem are $\Omega(\log N)$ time in general [9] and $\Omega(\log^2 N)$ time for "smooth relabeling strategies" [9, 10, 32]. The problem has also been studied in the context of average-case analysis [13, 16, 17].

Raman [23] gives a scheme for maintaining N elements in order using polynomial space ($\Theta(N^{1+\varepsilon})$) and $O(\log^2 N)$ time per update. The update bound can be improved to $O(\log N)$ by tuning existing algorithms [3, 8, 11, 27], and this bound is optimal [9]. However, such a blowup in space is disastrous for data locality, so this work does not apply to the traversal problem.

2 Cache-Oblivious Traversal in $O((\log\log N)^{2+\varepsilon})$

We first consider the cache-oblivious traversal problem without restriction on the number of scans. In this context, we can store the data out-of-order, but because the cache-oblivious algorithm does not know the block size, we do not know how out-of-order the data can be. To resolve this conflict, our data layout keeps the data "locally ordered" to obtain the following result:

Theorem 1. *There is a cache-oblivious data structure supporting traversals of k elements in $O(\lceil k/B\rceil + [B^\varepsilon$ if $B^{1-\varepsilon} \le k \le B])$ memory transfers, and insertions and deletions in $O(\lceil(\log\log N)^{2+\varepsilon}/B\rceil)$ amortized memory transfers, for any $\varepsilon > 0$. Each update moves $O((\log\log N)^{2+\varepsilon})$ amortized elements. More generally, a block of k consecutive elements can be inserted or deleted in $O(\lceil k(\log\log N)^{2+\varepsilon}/B\rceil)$ memory transfers, for any $\varepsilon > 0$. If $k = N$, then the number of memory transfers is $O(\lceil N/B\rceil)$. These bounds assume that $M = \Omega(B^\tau)$ for some $\tau > 1$.*

For intuition, we first describe the structure top-down, but for formality, we define the structure bottom-up in the next paragraph. Consider the entire memory region storing the N elements as a *widget* W_N. This widget is recursively laid out into *subwidgets* as follows. Each widget W_ℓ consists of $(1+1/(\log\log\ell)^{1+\varepsilon})\ell^{1-\alpha}$ subwidgets of type W_{ℓ^α} for some $\varepsilon > 0$ and $0 < \alpha < 1$. The constant α is chosen near 1, depending both on ε (see Lemma 2) and on the weak tall-cache assumption. Specifically, if $M = \Omega(B^\tau)$, then $\alpha \ge 1/\tau$. For example, under the standard tall-cache assumption in cache-oblivious and external-memory algorithms, $\tau = 2$ and $\alpha \ge 1/2$. The closer τ and α are to 1, the weaker the assumption on the memory hierarchy.

This top-down definition of widgets suffers from rounding problems. Instead, we define the finest widget, $W_{(\log\log N)^{2+\varepsilon}}$, to consist of $\Theta((\log\log N)^{2+\varepsilon})$ elements. From finer widgets of type W_ℓ we build widgets of type $W_{\ell^{1/\alpha}}$, from which we build widgets of type $W_{\ell^{(1/\alpha)^2}}$, etc. The coarsest widget type is between W_N and $W_{N^{1/\alpha}}$, specifically, $W_{\tilde{N}}$ where $\tilde{N} = (1/\alpha)^{\lceil\log_{1/\alpha} N\rceil}$. Thus, conceptually, we keep the top widget underfull to avoid rounding at every level. In fact, this approach wastes too much space, and we partition the structure into widgets of various sizes, as described at the end of this section.

At each recursive level of detail, each widget W_ℓ is either *empty*, which means it stores no elements, or *active*, which means it stores at least ℓ elements. We

recursively impose the restriction that if a widget W_ℓ is active, at least $\ell^{1-\alpha}$ of its subwidgets are active.

Lemma 1. *Widget W_ℓ occupies $\Theta(\ell)$ space.*

Proof. Let $S(\ell)$ denote the space occupied by widget W_ℓ. Then $S(\ell) = (1 + 1/(\log\log\ell)^{1+\varepsilon})\,\ell^{1-\alpha}\,S(\ell^\alpha)$, which has the solution $S(\ell) = \Theta(\ell)$ for any constant $\varepsilon > 0$ and $0 < \alpha < 1$. It is for this space bound that we need $\varepsilon > 0$. □

Within a widget W_ℓ, the subwidgets of type W_{ℓ^α} are stored in a consecutive segment of memory (for fast traversals), but out-of-order (for fast updates). This "unordered divide-and-conquer" recursive layout is powerful, allowing us to break through the polylogarithmic barrier.

Lemma 2. *Traversing k elements in a widget W_ℓ requires $O(\lceil k/B\rceil + [B^\varepsilon$ if $B^\varepsilon \le k \le B])$ memory transfers.*

Proof. Let j be the smallest integer such that widget W_j has size greater than B. Thus subwidget W_{j^α} fits into a block. There are three cases.

First, if $k = O(j^\alpha) \le O(B)$, then by Lemma 1 the elements to be traversed fit in a constant number of subwidgets of type W_{j^α}. Each subwidget W_{j^α} occupies a contiguous region of memory, and hence occupies at most two memory blocks. Thus, the traversal takes $O(1)$ memory transfers, which is $O(\lceil k/B\rceil)$ as desired.

Second, if $k = \Omega(j) > \Omega(B)$, then the number of elements to be traversed is $\Omega(1)$ times the size of a widget W_j. Because $j^\alpha \le O(B)$, the weak tall-cache assumption implies that $j = (j^\alpha)^{1/\alpha} \le O(B^{1/\alpha}) \le O(B^\tau) \le O(M)$. Hence, each widget W_j fits in memory. Thus we spend at most $O(\lceil j/B\rceil)$ memory transfers to read an entire widget or part of a widget. Hence, the total number of memory transfers is $O(\lceil k/j\rceil\lceil j/B\rceil)$, which is $O((k/j)(j/B)) = O(k/B)$ because $k = \Omega(j)$ and $j = \Omega(B)$.

Third, if k is $\Omega(j^\alpha)$ and $O(j)$, then k is also $\Omega(B^\alpha)$ and $O(B)$. Accessing each subwidget of type W_{j^α} only costs $O(1)$ block transfers because $j^\alpha = O(B)$, but the number of subwidgets of type W_{j^α} may be as much as $j^{1-\alpha} = O(B^{1/\alpha})^{1-\alpha} = O(B^{1/\alpha-1})$. Setting $\varepsilon = 1/\alpha - 1$, ε approaches 0 as α approaches 1, and the desired bound holds. □

We now outline the behavior of this layout under insertions. At the top level, when $W_{\tilde{N}}$ is active, we have at least $\tilde{N}^{1-\alpha}$ active subwidgets, so at most $\tilde{N}^{1-\alpha}/(\log\log\tilde{N})^{1+\varepsilon}$ empty subwidgets. When all of a widget's subwidgets are active, we say that the widget is *hyperactive*. Insertions are handled locally if possible. Insertion into a hyperactive widget may find room in a subwidget, or it may "overflow." When a widget overflows, it passes its logically last subwidget to its parent widget for handing. At this point, the widget is no longer hyperactive, because it has a (single) empty subwidget. The remaining issues are how the parent widget "handles" a given subwidget, and when that causes an overflow.

More precisely, the following steps define the possible outcomes of an insertion into a W_ℓ widget. We will not move individual elements (except at the finest level of detail); rather, we will manipulate entire subsubwidgets of type $W_{\ell^{\alpha^2}}$, inserting them into subwidgets of type W_{ℓ^α}.

1. Conceptually, we recursively insert the element into the appropriate subwidget S of type W_{ℓ^α}, and if that widget does not overflow, we are finished. In fact, the insertion algorithm is not recursive; it begins at the finest level and works its way up to coarser levels.
2. If the subwidget S overflows (in particular, it was hyperactive), then S gave us one of its subsubwidgets T of type $W_{\ell^{\alpha^2}}$ to handle. We insert this subsubwidget T into the logical successor S' of S. If S' was not hyperactive, it has room for T and we are finished. If S' was hyperactive, we push T into S' anyway, and out pops the last subsubwidget of S', T'. We pass T' onto the successor of S', and *cascade* in this fashion for $(\log\log\ell)^{1+\varepsilon}$ steps, or until we find a nonhyperactive subwidget. (In fact, if a cascade causes the last subsubwidget of S to pop out of subwidget S after fewer than $(\log\log\ell)^{1+\varepsilon}$ steps, then we reverse-cascade; below for simplicity we only consider cascading forward.)
3. If we finish the cascade without finding a nonhyperactive subwidget, we enter the *activation stage*. Consider the last subwidget L of the cascade. Take its logically last $\ell^{(1-\alpha)^2}$ subsubwidgets of type $W_{\ell^{\alpha^2}}$, and form a new subwidget of type W_{ℓ^α} from these. This creates a new active subwidget (to be placed later), but leaves L nearly empty with exactly $\ell^{(1-\alpha)^2}/(\log\log\tilde{N})^{1+\varepsilon}$ subsubwidgets (because L was hyperactive), which is not enough for L to be active. Next take enough subsubwidgets from L's logical predecessor to make L active. Repeat this process back through the cascade chain until the beginning. Because all cascade subwidgets are hyperactive, we have enough subsubwidgets to activate a new subwidget, while leaving every cascade subwidget active.
4. If the W_ℓ widget was not hyperactive, we store the new active subwidget from the previous step into one of the empty subwidgets. Otherwise, the W_ℓ widget *overflows* to its parent, passing up the logically last subwidget.

Theorem 2. *The amortized cost of an insertion into this layout is* $O((\log\log N)^{2+\varepsilon})$ *time and* $O(\lceil(\log\log N)^{2+\varepsilon}/B\rceil + [B^\varepsilon \text{ if } B^\varepsilon \le k \le B])$ *memory transfers.*

Proof. Within a widget of type W_ℓ, activation of a subwidget of type W_{ℓ^α} requires a logically contiguous segment of $(\log\log\ell)^{1+\varepsilon}$ subwidgets of type W_{ℓ^α} all to be hyperactive. After such an activation, these subwidgets are all nonhyperactive, each having an empty fraction of $1/(\log\log\ell^\alpha)^{1+\varepsilon}$. Thus, each of these subwidgets will not participate in another activation until these $\ell^{\alpha(1-\alpha)}/(\log\log\ell^\alpha)^{1+\varepsilon}$ subsubwidgets of type $W_{\ell^{\alpha^2}}$ are refilled. These refills are caused precisely by activations of subsubwidgets of type $W_{\ell^{\alpha^2}}$ within a subwidget of type W_{ℓ^α}. Thus, the activation of a subwidget of type W_{ℓ^α} requires the activation of $(\log\log\ell)^{1+\varepsilon}\ell^{\alpha(1-\alpha)}/(\log\log\ell^\alpha)^{1+\varepsilon} = \Theta(\ell^{\alpha(1-\alpha)})$ subsubwidgets of type $W_{\ell^{\alpha^2}}$. Working our way down to the bottom level, where widgets of type W_1 are created by insertions, the activation of a subwidget of type W_{ℓ^α} requires $\Theta(\ell^{\alpha(1-\alpha)}\ell^{\alpha^2(1-\alpha)}\cdots 1)$ insertions. The exponents form a geometric series, and we obtain that $\Theta(\ell^{(\alpha+\alpha^2+\cdots)(1-\alpha)}) \le \Theta(\ell^{(\alpha/(1-\alpha))(1-\alpha)}) = \Theta(\ell^\alpha)$ insertions are needed for an activation of a subwidget of type W_{ℓ^α}.

An activation of a subwidget of type W_{ℓ^α} involves the movement of $O((\log\log\ell)^{1+\varepsilon})$ subwidgets, which costs $O((\log\log\ell)^{1+\varepsilon}\ell^\alpha)$ time and $O((\log\log\ell)^{1+\varepsilon}\lceil\ell^\alpha/B\rceil)$ memory transfers. We charge this cost to the $\Theta(\ell^\alpha)$ insertions that caused the activation. As a result, each insertion is charged at most $\log\log N$ times, once for each level. The amortized time for the activation is therefore $O((\log\log\ell)^{2+\varepsilon})$.

To compute the amortized number of memory transfers, there are three cases. If $\ell^\alpha \geq B$ so $\ell \geq B$, we have $O(\lceil(\log\log N)^{2+\varepsilon}/B\rceil)$ amortized memory transfers. If $\ell \leq B$ so $\ell^\alpha \leq B$, then the number of memory transfers is $O(1)$. If $\ell^\alpha < B$ and $\ell > B$, then at this particular level, there is an additional amortized cost of $(\log\log\ell)^{1+\varepsilon}\lceil\ell^\alpha/B\rceil/\ell^\alpha = (\log\log\ell)^{1+\varepsilon}/\ell^\alpha = o(1)$.

Cascades are more frequent than activations, because several cascades occur at a particular level between two activations, but a cascade can only be triggered by an activation at the next lower level. Also, cascades are cheaper than activations, because they only touch one subsubwidget per subwidget involved. Therefore, the cost of a cascade within widget W_ℓ is $O((\log\log\ell)^{1+\varepsilon}\ell^{\alpha^2})$, which can be amortized over the $\Omega(\ell^{\alpha^2})$ insertions that must take place to cause a subsubwidget to activate. The amortized number of memory transfers is bounded as above. □

The overall structure consists of widgets of several types, because one widget is insufficient by itself to support a wide range of values of N. In the ideal situation, the structure consists of some number of active widgets of type $W_{(1/\alpha)^{\lceil\log_{1/\alpha} N\rceil}}$, followed by an equal number of empty widgets of that type, followed by some number of active widgets of the next smaller type, followed by an equal number of empty widgets of that type, etc. The number of widgets of each type can be viewed roughly as a base-$(1/\alpha)$ representation of N. Whenever all widgets of a particular type become active and another is needed, the entire structure to the right of those widgets is moved to make room for an equal number of empty widgets of that type.

A slight modification of our data structure supports deletions, by allowing each widget W_ℓ to consist of between $(1 - 1/(\log\log\ell)^{1+\varepsilon})\ell^{1-\alpha}$ and $(1 + 1/(\log\log\ell)^{1+\varepsilon})\ell^{1-\alpha}$ subwidgets of type W_{ℓ^α}. We cannot use the global rebuilding technique, in which we mark deleted nodes as ghost nodes and rebuild whenever the structure doubles in size, because many ghost nodes in a region of elements decreases the density, causing small traversals within that region to become expensive.

3 Cache-Oblivious Traversal with Self-Adjusting Scans

In this section we modify the data structure in the previous section so that, in addition to achieving near-optimal worst-case traversal bounds, the structure achieves optimal traversal bounds in an amortized setting. The idea is to allow expensive traversals to adjust the data structure in order to improve the cost of future traversals. Specifically, out-of-order subwidgets make traversal expensive,

and this jumbling is caused by insertions. Thus, we augment traversals to sort the subwidgets traversed. The main difficulty is that there is little extra space in any widget, and thus little room to re-order subwidgets.

We partition each widget W_ℓ into two parts: the "main" left part, which is the initial set of subwidgets, and the "extra" right part, which is a $\Theta(1/\log\log\ell)$ fraction of the size. We never move the widgets in the main part, so the subwidgets within the main part remain sorted with respect to each other at all times. The subwidgets in the extra part serve as little additions in between adjacent subwidgets in the main part, but they are stored off to the right side.

We enforce the constraint that only a third (or any constant fraction $\leq 1/3$) of the extra part is actually occupied by subwidgets; the remaining two thirds is (a small amount of) extra empty space that we use for memory management.

Consider a traversal in widget W_ℓ. The critical case is when ℓ is asymptotically larger than B, but the size of a subwidget W_{ℓ^α} is asymptotically smaller than B, so each random access to a subwidget causes an additional not-totally-used memory transfer.

We consider the traversal as having two working sets. On the one hand, it may visit subwidgets in the main part of the widget. In this context, no additional memory transfers are incurred, because the subwidgets in the main part are in sorted order and thus are visited consecutively.

On the other hand, the traversal may visit subwidgets in the extra part of the widget. Sometimes these subwidgets are stored in sorted order, and sometimes they are out-of-order. We count the number r of consecutive *runs* of subwidgets stored in order that the traversal visits. Each distinct run incurs $O(1)$ extra memory accesses. We ignore the cost of the first run, but charge the remaining $O(r-1)$ memory accesses to the previous insertions that caused those runs to split in the first place.

Now the traversal concatenates these $r-1$ runs, so that future traversals do not also charge to the same insertions. We proceed by sorting the r runs, i.e., extracting the runs and writing them in the correct order at the end of the extra part of the widget, and we erase the original copies of the runs.

Ideally, there is enough unused space at the end of the extra part of the widget to fit the new concatenated run. If there is not enough space, at least the first two thirds of the extra part must be occupied (with holes), which means that we have already concatenated several runs, an amount equal to a third of the entire size of the extra part. We charge to this prior concatenation the cost of *recompactification*: shifting items in the extra part to the left so as to fill all holes and maintain the relative order of the items. Now only the first third of the extra part is occupied, so there is room for the new run at the right end.

The total cost, therefore, is $O(\lceil k/B\rceil)$ amortized memory transfers for traversing k elements. Each insertion is only charged at most $O(1)$ extra memory transfers, so the cost of insertions does not change.

Theorem 3. *There is a data structure achieving all bounds claimed in Theorem 1, with the additional property that traversing k elements uses only $O(\lceil k/B\rceil)$ amortized memory transfers.*

4 Constant Number of Physical Scans

We now impose the restriction that every operation uses only $O(1)$ physical scans of memory. In the cache-oblivious model, this restriction requires us to keep the elements close to their sorted order. Thus, we base our data structure on the ordered-file structures mentioned in Section 1.3. Specifically, our structure is motivated by the "packed-memory structure" of [4]. Unfortunately, the packed-memory structure performs $\Theta(\log N)$ physical scans per update in the worst case.

In this section, we show how to reduce the number of physical scans to $O(1)$ per operation in the worst case. Our structure does not use an implicit binary-tree structure on top of the array as in [4, 17]. Instead, we always rebalance in intervals to the right of the updated element. This algorithm requires a different analysis because we can no longer charge costs to internal nodes in the binary tree. Nonetheless, we show that the same performance bounds are achieved, thereby proving the following theorem:

Theorem 4. *There is a cache-oblivious data structure supporting traversals of k elements in $O(\lceil k/B \rceil)$ memory transfers and $O(1)$ physical scans, and insertions and deletions in $O(\lceil (\log^2 N)/B \rceil)$ amortized memory transfers and $O(1)$ physical scans. Each update moves $O(\log^2 N)$ amortized elements. More generally, a block of k consecutive elements can be inserted or deleted in $O(\lceil (\log^2(N/k))/B \rceil)$.*

The algorithm works as follows. We consider rebalancing intervals in the circular array, where the smallest interval has size $\log N$ and the intervals increase as powers of two up to $\Theta(N)$. The *density* of an interval is the fraction of space occupied by elements. Let $h = \log N - \log\log N$ be the number of different interval sizes. Associated with each interval are two *density thresholds* which are guidelines for the acceptable densities of the interval. (The density of an interval may deviate beyond the thresholds, but as soon as the deviation is "discovered," the densities of the intervals are adjusted to be within the thresholds.)

The density thresholds are determined by the size of the interval. We denote the upper and lower density thresholds of an interval of size $(\log N)2^j$ by τ_j and ρ_j, respectively. These thresholds satisfy $\rho_h < \rho_{h-1} < \cdots < \rho_1 < \rho_0 < \tau_0 < \tau_1 < \ldots < \tau_h$. The values of the densities are determined according to an arithmetic progression. Specifically, let $\tau_0 = \alpha < 1$ and let $\tau_h = 1$. Let $\delta = (\tau_h - \tau_0)/h$. Then define density threshold τ_k to be $\tau_k = \tau_0 + k \cdot \delta$. Similarly, let $\rho_0 = \beta < \alpha$ and $\rho_h = \gamma < \beta$. Let $\delta' = (\rho_0 - \rho_h)/h$. Then define density threshold ρ to be $\rho = \rho - k \cdot \delta'$.

The insertion and deletion algorithms work as follows. We begin with the interval of size $\log N$ starting at the updated element. If the density of the interval is outside its density thresholds, we grow the interval rightward to double its size. Then we evenly space all of the elements within the interval.

5 Traversals in the DAM Model

The standard traversal structure on a DAM model can be applied recursively to support a multilevel memory hierarchy:

Theorem 5. *For any constant $c > 1$, there is a data structure on an ℓ-level memory hierarchy supporting traversals of k elements in $O(\lceil c^\ell k/B_i \rceil)$ memory transfers at level i, and insertions and deletions in $O(c^\ell + 1/(c-1)^\ell)$ memory transfers.*

An important consequence of this theorem is that this approach does not apply when the memory hierarchy has $\omega(1)$ levels, because of the exponential blowup in space and consequently the exponential decrease in density.

To support operations in $O(1)$ physical scans, we modify the approach from Section 4. We divide the data into blocks of size $\Theta(B)$, and store them in a circular array of blocks. Within each block, the elements are unsorted, but there is a total order among the blocks. Thus, the rank of each element in the circular array is within $O(B)$ of its actual rank. When we traverse k elements, the block access pattern is sequential, even though the element access pattern is not. Thus traversals use $O(1)$ physical scans.

An insertion or deletion may cause the block to become too full or too empty, that is, cause a split and/or merge. Splitting or merging blocks means that we must insert or delete a block into the circular array. We maintain the order among the $\Theta(N/B)$ blocks by employing the algorithm in Section 4. The same bound as before applies, except that we now manipulate $\Theta(N/B)$ blocks of size $\Theta(B)$ instead of N individual elements. Each insertion or deletion of a block moves $O(B \log^2(N/B))$ elements for a cost of $O(\log^2(N/B))$ memory transfers; but such a block operation only happens every $\Omega(B)$ updates. Thus, we obtain the following theorem:

Theorem 6. *There is a data structure supporting traversals of k elements in $O(\lceil k/B \rceil)$ memory transfers and $O(1)$ physical scans, and insertions and deletions in $O(\lceil (\log^2(N/B))/B \rceil)$ amortized memory transfers and $O(1)$ physical scans. Each update moves $O(\log^2(N/B))$ amortized elements. More generally, a block of k consecutive elements can be inserted or deleted in $O(\lceil (k \log^2(N/\max\{B,k\}))/B \rceil)$.*

References

1. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *CACM*, 31(9):1116–1127, Sept. 1988.
2. L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *STOC*, 2002.
3. M. A. Bender, R. Cole, E. D. Demaine, and M. Farach-Colton. Two simplified algorithms for maintaining order in a list. In *ESA*, 2002.
4. M. A. Bender, E. Demaine, and M. Farach-Colton. Cache-oblivious search trees. In *FOCS*, 2000.

5. M. A. Bender, E. D. Demaine, and M. Farach-Colton. Efficient tree layout in a multilevel memory hierarchy. In *ESA*, 2002.
6. M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. In *SODA*, 2002.
7. G. S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height (extended abstract). In *SODA*, 2002.
8. P. Dietz. Maintaining order in a linked list. In *STOC*, 1982.
9. P. Dietz, J. I. Seiferas, and J. Zhang. A tight lower bound for on-line monotonic list labeling. In *SWAT*, 1994.
10. P. Dietz and J. Zhang. Lower bounds for monotonic list labeling. In *SWAT*, 1990.
11. P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *STOC*, 1987.
12. M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. Overcoming the memory bottleneck in suffix tree construction. In *FOCS*, 1998.
13. W. R. Franklin. Padded lists: Set operations in expected $O(\log \log N)$ time. *IPL*, 9(4):161–166, 1979.
14. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS*, 1999.
15. J. Gray and G. Graefe. The five minute rule ten years later. *SIGMOD Record*, 26(4), 1997.
16. M. Hofri and A. G. Konheim. Padded lists revisited. *SICOMP*, 16:1073, 1987.
17. A. Itai, A. G. Konheim, and M. Rodeh. A sparse table implementation of priority queues. In S. Even and O. Kariv, editors, *ICALP*, 1981.
18. R. Ladner, J. Fix, and A. LaMarca. Cache performance analysis of algorithms. In *SODA*, 1999.
19. A. LaMarca and R. E. Ladner. The influence of caches on the performance of sorting. *Journal of Algorithms*, 31:66–104, 1999.
20. R. Melville and D. Gries. Controlled density sorting. *IPL*, 10:169–172, 1980.
21. D. Patterson and K. Keeton. Hardware technology trends and database opportunities. In *SIGMOD*, 1998. Keynote address.
22. H. Prokop. Cache-oblivious algorithms. Master's thesis, MIT, 1999.
23. V. Raman. Locality preserving dictionaries: theory and application to clustering in databases. In *PODS*, 1999.
24. S. Sen and S. Chatterjee. Towards a theory of cache-efficient algorithms. In *SODA*, 2000.
25. D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *CACM*, 28(2):202–208, 1985.
26. D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985.
27. A. Tsakalidis. Maintaining order in a generalized linked list. *Acta Informatica*, 21(1):101–112, May 1984.
28. J. S. Vitter. External memory algorithms. *LNCS*, 1461, 1998.
29. D. E. Willard. Maintaining dense sequential files in a dynamic environment. In *STOC*, 1982.
30. D. E. Willard. Good worst-case algorithms for inserting and deleting records in dense sequential files. In *SIGMOD*, 1986.
31. D. E. Willard. A density control algorithm for doing insertions and deletions in a sequentially ordered file in good worst-case time. *Information and Computation*, 97(2):150–204, Apr. 1992.
32. J. Zhang. Density control and on-line labeling problems. Technical Report TR481, University of Rochester, Computer Science Department, Dec. 1993.