Cache-Oblivious B-Trees

Michael A. Bender* Erik D. Demaine† Martin Farach-Colton‡

Abstract

We present dynamic search-tree data structures that perform well in the setting of a hierarchical memory (including various levels of cache, disk, etc.), but do not depend on the number of memory levels, the block sizes and number of blocks at each level, or the relative speeds of memory access. In particular, between any pair of levels in the memory hierarchy, where transfers between the levels are done in blocks of size B, our data structures match the optimal search bound of $\Theta(\log_B N)$ memory transfers. This bound is also achieved by the classic B-tree data structure, but only when the block size B is known, which in practice requires careful tuning on each machine platform. One of our data structures supports insertions and deletions in $\Theta(\log_B N)$ amortized memory transfers, which matches the B-tree's worst-case bounds. We augment this structure to support scans optimally in $\Theta(N/B)$ memory transfers. In this second data structure insertions and deletions require $\Theta(\log_B N + \frac{\log^2 N}{B})$ amortized memory transfers. Thus, we match the performance of the B-tree for $B = \Omega(\log N \log\log N)$.

1. Introduction

Steep Memory Hierarchy. The memory hierarchies of modern computers are becoming increasingly "steep," ranging from fast registers and on-chip cache down to relatively slow disks and networks. Because the speed of processors is increasing more quickly than the speed of memory and disk, the disparity between the various levels of the memory hierarchy is growing. For example, the Alpha 21264 chip can deliver 2 words from L1 cache in one cycle, but it requires approximately 100 cycles to bring data from main memory [12]. For any computer the access times of L1 cache and disk differ by approximately 6 orders of magnitude. Because of these large variations in magnitude, it is becoming increasingly dangerous to design algorithms assuming a "flat" memory with uniform access times.

*Department of Computer Science, State University of New York, Stony Brook, NY 11794-4400, USA. (*bender@cs.sunysb.edu*). Supported in part by HRL Laboratories and ISX Corporation.

†Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada. (*eddemaine@uwaterloo.ca*).

‡Google Inc. Work done while at Department of Computer Science, Rutgers University, Piscataway, NJ 08855, USA. (*martin@google.com*, *http://www.cs.rutgers.edu/~farach*). Supported by NSF grant CCR-9820879.

The distinguishing feature of multilevel memory hierarchies is that memory transfers are done in *blocks* in order to amortize the cost of a transfer [4]. This amortization only works when each transfer contains many pieces of data to be used by the CPU. Thus, our objective is to maintain *locality of reference*, meaning that memory accesses are clustered in time and space.

Maintaining Locality in Irregular and Dynamic Data. Data locality is easier to maintain in algorithmic problems having *regular* and/or *static* data, because the flow of data is predictable. Examples of such problems include matrix multiplication, fast Fourier transform, and LU decomposition. Even in sorting, there is a fixed and predetermined order in which the data must end up. In contrast, it is more challenging to maintain data locality in *irregular* and *dynamic* problems because by definition the data flow is continually changing and unpredictable, making it difficult to organize data locality a priori.

Irregular and dynamic problems often require *data-structure solutions* because data structures specialize in moving data efficiently throughout memory. It has long been recognized that the level of the memory hierarchy affects the choice of data structure. For example, one of the most fundamental data structures for manipulating arbitrary data is a *balanced search tree*. The basic functionality that we ask of a balanced search tree is that it maintain an ordered collection of elements subject to insertions, deletions, and searches. Thus, balanced search trees such as AVL trees [1], BB[α] trees [22], red-black trees [17], randomized search trees [29], skip lists [26], and splay trees [33] are appropriate for main memory, whereas B-trees [10] are more appropriate for external memory.

In this paper we develop search trees that are memory-efficient at all levels of the memory hierarchy. A central innovation of our data structures is that they *avoid any memory-specific parameterization*. That is, our data structures do not use any information about memory access times, or cache-line or disk-block sizes. It may be surprising that data structures can be made cache-efficient without using the parameters that describe the structure of the memory hierarchy, but we demonstrate that this is possible. Such data structures and algorithms are called *cache-oblivious* [16, 25]. The cache-oblivious model has been successfully explored in the context of regular and static problems [16, 25]. This paper initiates the problem of manipu-

lating irregular and dynamic data cache-obliviously.

Performance Models for Memory Hierarchy. The classic way to measure the running time of an algorithm (e.g., in the RAM model) is to count the number of machine instructions. In a machine with a memory hierarchy, an important additional factor to measure is the number of memory block transfers at each level, scaled according to the relative speed of accesses. The idea of counting memory transfers was introduced in the cell-probe model [43]. We stress that these memory transfer times should not be treated as constants because they can vary by many orders of magnitude and are often the dominant feature of running time.

There is a tradeoff between the accuracy of a model and its ease of use. One body of work explores multilevel hierarchies [2, 3, 5, 28, 37, 39] and more complicated models of memory [6, 27]. A problem with many of these models is that algorithms must take into account many parameters, e.g., the relative speeds and block sizes at each memory level. While this leads to accurate time predictions, it makes it difficult to design and analyze optimal algorithms in these models.

A second body of work concentrates on two-level memory hierarchies, either in the context of memory and disk [4, 9, 18, 37, 38], or cache and memory [30, 20]. In such a model there are only a few parameters, making it relatively easy to design efficient algorithms. The motivation is that it is common for one level of the memory hierarchy to dominate the running time. The difficulty with this approach is that the programmer must focus efforts on a particular level of the hierarchy, resulting in a program that is less flexible to different-scale problems, and does not adapt to when the dominating level changes, e.g., as a program starts to page in virtual memory.

B-Trees. Before developing our cache-oblivious search trees, we review the standard solutions for two-level and multilevel hierarchies. The classic two-level solution, both in theory and practice, is a *B-tree* [10]. The basic idea is to maintain a balanced tree having a fanout proportional to the memory block size B, rather than constant size. This means that one block read determines the next node out of B nodes, so a search completes in $\Theta(\log_B N)$ memory transfers. A simple information-theoretic argument shows that this bound is optimal.

The situation becomes more complex with more than two levels of memory hierarchy. We need a multilevel structure, one level per transfer block size. Suppose $B_1 > B_2 > \cdots > B_k$ are the block sizes between the $k+1$ levels of memory. At the top level we have a B_1-tree; each node of this B_1-tree is a B_2-tree, etc.

At this point such a data structure becomes very difficult to manage. Code that is generic to the number of memory levels is complicated and almost certainly inefficient. Hierarchy-specific code needs to be rewritten for each machine architecture. Even with generic code, the parameters in the code needed to be changed or tuned to the specific memory hierarchy each time the program is ported to another machine. This parameter tuning can be difficult and time-consuming, and if it is done improperly it can have worse effects than much simpler algorithms. Finally, such tuning is impossible in a heterogeneous computing environment where processors with different local memory characteristics access the same data, for example, across a network, or stored on some distributable permanent medium such as CD-ROM.

Cache-Oblivious Algorithms. The cache-oblivious model enables us to reason about a simple two-level memory model, but prove results about an unknown multilevel memory model. This model was introduced by Frigo, Leiserson, Prokop, and Ramachandran [16, 25] introduced the cache-oblivious model as a clean way to They show how several basic problems—namely matrix multiplication, matrix transpose, Fast Fourier Transform, and sorting—have optimal algorithms that are cache-oblivious. Optimal cache-oblivious algorithms have also been found for LU decomposition [11, 34] and a static, complete binary tree [25]. All these algorithms perform an *asymptotically optimal* number of memory transfers for *any* memory hierarchy and at *all levels* of the hierarchy. More precisely, the number of memory transfers between any two levels is within a constant factor of optimal. In particular, any linear combination of the transfer counts is optimized.

The theory of cache-oblivious algorithms is based on the *ideal-cache model* of Frigo, Leiserson, Prokop, and Ramachandran [16, 25]. In this model there are two levels in the memory hierarchy, which we call *cache* and *disk*, although they could represent any pair of levels. The disk is partitioned into *memory blocks* each consisting of a fixed number B of consecutive cells. The cache has room for C memory blocks, although the exact value of this parameter will not be important in our applications. The cache is *fully associative*, that is, it can contain an arbitrary set of C memory blocks at any time. Recall that parameters B and C are unknown to the cache-oblivious algorithm or data structure, and they should not be treated as constants.

When the algorithm accesses a location in memory that is not stored in cache, the relevant memory block is automatically fetched from disk. in what we call a *memory transfer*. If the cache is full, the ideal memory block in the cache is elected for replacement, based on the future characteristics of the algorithm.

While this model may superficially seem unrealistic, Frigo et al. have shown that it can be simulated by essentially any memory system with only a small constant-factor overhead. For example, if we run a cache-oblivious algorithm on a several-level memory hierarchy, we can use the

ideal-cache model to analyze the number of memory transfers between each pair of adjacent levels. Because the algorithm is cache-oblivious, it performs well on all levels. See [16] for details.

The algorithms in our paper only need a constant number of memory blocks in the cache at once, so any semi-intelligent block-replacement strategy will suffice for our purposes. In general, however, the least-recently-used (LRU) block-replacement strategy can be used to approximate the omniscient strategy within a constant factor [16, 32]. Finally, the assumptions of full associativity and automatic block replacement are also reasonable, because they can be implemented in normal memory with only an expected constant factor overhead [16].

The concept of algorithms that are uniformly optimal across multiple memory models was considered previously by Aggarwal, Alpern, Chandra, and Snir [2]. These authors introduce the HMM model, in which the cost to access memory location x is $\lceil f(x) \rceil$, where $f(x)$ is monotone nondecreasing and polynomially bounded. They show matrix multiplication and FFT algorithms that are optimal for any cost function $f(x)$. One distinction between the HMM model and the cache-oblivious model is that, in the HMM model, memory is managed by the algorithm designer, whereas in the cache-oblivious model, memory is managed by the existing caching and paging mechanisms. Also, the HMM model does not model block transfers. The HMM model was extended by Aggarwal, Chandra, and Snir to the BT model to take into account block transfers [3]. In the BT model the algorithm can choose and vary the block size, whereas in the cache-oblivious model the block size is fixed and unknown.

Results. We develop three cache-oblivious search-tree data structures. These results demonstrate that even irregular dynamic problems can be solved efficiently in the cache-oblivious model. Let B be the (unknown) memory block size and let N be the number of elements in the tree.

Our first cache-oblivious search tree is implemented in a *single* array of size $O(N)$. This data structure has the same optimal query bound as B-trees, $O(\log_B N)$ memory transfers.[1] Furthermore, searching from an arbitrary node at height h has an optimal cost of $O(1 + h/B)$ memory transfers. Scans of k elements also run optimally in $O(1 + k/B)$ memory transfers. Insertions and deletions require $O(\log_B N + \frac{\log B}{\sqrt{B}} \log^2 N)$ amortized memory transfers. This bound matches the B-tree search cost of $O(\log_B N)$ provided $B = \Omega((\log N)^2 (\log \log N)^4)$.

The memory block size B being at least polylogarithmic in the problem size N is reasonable, just as the transdichotomous model [14] specifies that the number of bits in a machine word is logarithmic in the problem size. Furthermore, we have the most to gain from locality of reference when the block size is large, so our data structures are optimized for these cases. As memory hierarchies get deeper and the lowest level becomes farther from the CPU, our data structures become more important.

We improve the insertion and deletion costs by using one level of indirection. Our second cache-oblivious search tree again performs searches in $O(\log_B N)$ memory transfers and scans in $O(1 + \frac{N}{B})$ memory transfers. However, insertions and deletions are even more efficient, requiring $O(\log_B N + \frac{\log^2 N}{B})$ amortized memory transfers. This last bound matches the B-tree search cost of $O(\log_B N)$ provided $B = \Omega(\log N \log \log N)$.

In our third cache-oblivious search tree, we remove the requirement that the data structure perform scans. This data structure has the same optimal query bound as B-trees, $O(\log_B N)$ memory transfers, and the same amortized insert/delete bound as B-trees, $O(\log_B N)$ amortized memory transfers.

These data structures use new tools for cache-oblivious manipulation of data. Several of these tools are of more general interest than search trees. Following the work of Itai, Konheim, and Rodeh [19] and Willard [40, 41, 42], we develop a *packed-memory structure* for maintaining an ordered collection of N items in an array of size $O(N)$ subject to insertions and deletions in $O(\frac{\log^2 N}{B})$ memory transfers; see Section 2.3. In another context, this structure can be thought of as a cache-oblivious *linked list* that supports scanning k consecutive elements in $O(1 + k/B)$ memory transfers (instead of the naïve $O(k)$), and updates in $O(1 + \frac{\log^2 N}{B})$ amortized memory transfers. We conjecture that the update bound of this cache-oblivious linked list is best possible subject to obtaining the optimal scan bound cache-obliviously.

This packed-memory structure seems fundamental to the problem of reorganizing data cache-obliviously. In the context of balanced search trees, we fold a tree layout resembling the van Emde Boas data structure onto the packed-memory structure. We use a strongly weight-balanced search tree, which has desirable properties for maintaining locality of reference; see Section 2.2. The packed-memory structure allows us to make room for inserted nodes, but has the side effect of moving nodes around, which invalidates the parent and child pointers of nodes. We develop additional techniques to maintain the integrity of the pointers, including an analysis of local and long-distance nodes (Section 3.2), and the notion and analysis of buffer nodes (Sections 3.3 and 3.4). Finally, we employ indirection to improve the update bounds (Section 4).

Notation. One notational convention that we will use throughout is of independent interest.[2] We define the *hy-*

[1] As with B-trees, our goal is to optimize query time, not update time as in buffer trees [7].

[2] All logarithms are base 2 if not otherwise specified.

perfloor of x, denoted $\lfloor\!\lfloor x \rfloor\!\rfloor$, to be $2^{\lfloor \log x \rfloor}$, i.e., the largest power of two smaller than x. Thus, $x/2 < \lfloor\!\lfloor x \rfloor\!\rfloor \leq x$. Similarly, the *hyperceiling* $\lceil\!\lceil x \rceil\!\rceil$ is defined to be $2^{\lceil \log x \rceil}$. Analogously, we define *hyperhyperfloor* and *hyperhyperceiling* by $\lfloor\!\lfloor\!\lfloor x \rfloor\!\rfloor\!\rfloor = 2^{\lfloor\!\lfloor \log x \rfloor\!\rfloor}$ and $\lceil\!\lceil\!\lceil x \rceil\!\rceil\!\rceil = 2^{\lceil\!\lceil \log x \rceil\!\rceil}$. These satisfy $\sqrt{x} < \lfloor\!\lfloor\!\lfloor x \rfloor\!\rfloor\!\rfloor \leq x$ and $x \leq \lceil\!\lceil\!\lceil x \rceil\!\rceil\!\rceil < x^2$.

2. Tools for Cache-Oblivious Data Structures

2.1. Static Layout and Searches

We first present a cache-oblivious *static* search-tree structure, which is the starting point for our dynamic structure. More precisely, given a base search tree, where each node has $O(1)$ children, we describe a mapping from the nodes of the tree to positions in memory. We call this mapping the *van Emde Boas layout*, because it resembles the recursive structure in the van Emde Boas data structure [35, 36].[3] Assuming the base tree has height $\Theta(\log N)$, our structure performs a search in $\Theta(\log_B N)$ memory transfers, which is optimal to within a constant factor. Our layout is somewhat modified from the layout for complete binary trees of Prokop [25, pp. 61–62].

The basic idea of the van Emde Boas layout is simple. Suppose the tree has height h, and suppose first that h is a power of two. Conceptually split the tree at the middle level of edges, between nodes of height $h/2$ and $h/2+1$. This breaks the tree into the *top recursive subtree* A of height $h/2$, and several *bottom recursive subtrees* $B_1, \ldots, B_k$ of height $h/2$. In particular, if all nonleaf nodes have the same number of children, then the recursive subtrees all have size roughly $\sqrt{N}$, and k is roughly $\sqrt{N}$. The layout of the tree is obtained by recursively laying out each recursive subtree, and combining these layouts in the order A, B_1, ..., B_k; see Figure 1.

If h is not a power of two, the obvious form of rounding is to assign the top $\lfloor h/2 \rfloor$ levels to the top recursive subtree, and the bottom $\lceil h/2 \rceil$ levels to the bottom recursive subtrees. This rounding is satisfactory for the static structure, but it is particularly useful for the dynamic structure to use a different rounding scheme. We assign a number of levels that is a power of two to the bottom recursive subtrees, and assign the remaining levels to the top recursive subtree. More precisely, the bottom subtrees have height $\lceil\!\lceil h/2 \rceil\!\rceil$ $(= \lfloor\!\lfloor h-1 \rfloor\!\rfloor)$ and the top subtree has height $h - \lceil\!\lceil h/2 \rceil\!\rceil$.

We now introduce the notion of *levels of detail*. This notion is useful both for understanding why searches use few memory transfers in the van Emde Boas layout and for understanding future manipulations of the layout. Any level of detail is a partition of the tree into disjoint recursive subtrees. The finest level of detail is level of detail 0, in which each node is its own recursive subtree. The coarsest level of detail, $\lceil \log_2 h \rceil$, is just the tree itself. In general, level of detail k is derived by starting with the entire tree, recursively partitioning it as described above, and exiting the recursion whenever we reach a recursive subtree of height $\leq 2^k$. Note that according to the van Emde Boas layout, each recursive subtree is stored in a contiguous block of memory.

One useful consequence of our method of rounding is the following:

Lemma 1 *At level of detail k, all recursive subtrees except the one containing the root have the same height of 2^k. The recursive subtree containing the root has height between 1 and 2^k inclusive.*

Lemma 2 *Consider an N-node search tree T that is stored in a van Emde Boas layout. Suppose that each node in T has between $\delta \geq 2$ and $\Delta = O(1)$ children. Let h be the height of T. Then a search in T uses at most $4\lceil \log_\delta \Delta \log_B N + \log_B \Delta \rceil = O(\log_B N)$ memory transfers.*

2.2. Strongly Weight-Balanced Search Trees

In order to convert the static static layout into a dynamic layout, we need a dynamic balanced search tree. Our structure will ultimately require important properties of the balanced search tree that most search trees lack. The first property is that, whenever a node is *rebalanced* (modified to keep the tree balanced), we can afford to scan all of its descendants. More formally, we require the following property:

Property 1 *Suppose that whenever we rebalance a node v we also* touch *all of v's descendents. Then the amortized number of elements that are touched per insertion is only $O(\log N)$.*

Many weight-balanced trees, such as BB[α] trees [22], have this property [21]. A tree is *weight-balanced* if the left subtree (including the root) and the right subtree (including the root) have sizes that differ by at most a constant factor. This is stronger than merely requiring that most modifications are near the leaves.

We require a stronger form of weight-balancedness that most weight-balanced trees lack. Standard weight-balancedness only guarantees a relative bound between subtrees with a common root, so the size difference between subtrees of the same height may be large. In contrast, a *strongly weight-balanced tree* satisfies the following absolute constraint, relating the sizes of all subtrees at the same level.

Property 2 *For some constant d, any node v at height h has $\Theta(d^h)$ descendents.*

One search tree that satisfies Properties 1 and 2 is the *weight-balanced B-tree* of Arge and Vitter [8], and we will

[3] We do not use a van Emde Boas tree—we use a normal tree with pointers from each node to its parent and children—but the order of the nodes in memory is reminiscent of van Emde Boas trees.

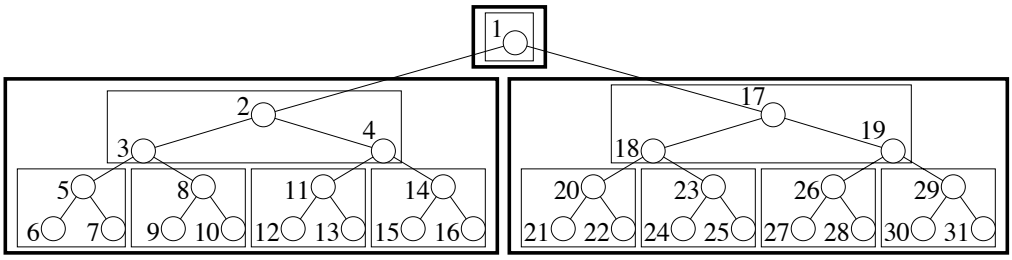

Figure 1. The van Emde Boas layout of a tree of height 5.

use this structure in our cache-oblivious B-tree.

Definition 3 (Weight-Balanced B-tree [8]) *The* weight $w(u)$ *of a node u in a tree T is the size of the subtree rooted at u. We say that T is a* weight-balanced B-tree *with branching parameter d, where $d > 4$, if the following conditions hold:*[4]

1. *All leaves of T have the same depth.*
2. *The root of T has more than one child.*
3. Balance: *Consider a nonroot node u at height h in the tree. (Leaves have height 1.) The weight of u is bounded as follows:*

$$\frac{d^{h-1}}{2} \leq w(u) \leq 2d^{h-1}.$$

4. Amortization: *If a nonroot node u at height h has just been rebalanced, then we will need $\Omega(d^h)$ insertions and/or deletions before u is rebalanced again. That is, $w(u) - d^{h-1}/2 = \Theta(d^h)$ and $2d^{h-1} - w(u) = \Theta(d^h)$.*

This definition yields the following lemma, which is proved by a simple counting argument.

Lemma 4 [8] *Consider a weight-balanced B-tree with branching parameter d. The root has between 2 and $4d$ children. All internal nodes have between $d/4$ and $4d$ children. The height of the tree is $O(\log_d N)$.*

In this paper we need an additional property about weight-balancedness of subtrees not containing leaves:

Lemma 5 *Consider the subtree A of a weight-balanced B-tree containing a node v, its children, its grand-children, etc., down some number a of levels. Then $|A| < 4d^a$.*

We now describe how to do inserts and deletes. In [8], deletes are performed using the global rebalancing technique of [24], where deleted nodes are treated as "ghost" nodes to be removed when the tree is periodically reassembled. In this paper, we will need to service deletes immediately, which is a staightforward modification of the presentation in [8].

Insertions. We search down the tree to find where to insert a new leaf w. After inserting w, some ancestors of w may become unbalanced. That is, some ancestor node u at height h may have weight higher than $2d^{h-1}$. We now bring the ancestors of w into balance starting from the ancestors closest to the leaves. (If a child of node u is split, this does not affect the weight $w(u)$ of u, and thus this is a good order in which to rebalance.)

If a node u at height h is out of balance, then we split u into two nodes u_1 and u_2, which share the node u's children, $v_1, \ldots, v_k$. In general we cannot partition $v_1, \ldots, v_k$ between u_1 and u_2 so that $w(u_1)$ and $w(u_2)$ are exactly equal (unless we rebalance u's grandchildren, great-grandchildren, etc). However, we can divide the children fairly evenly as follows. Find the longest sequence of $v_1, \ldots, v_{k'}$ such that their total weight is at most $\lceil w(u)/2 \rceil$, that is, $\sum_{i=1}^{k'} w(v_i) \leq \lceil w(u)/2 \rceil$. Thus, the smallest value $w(u_1)$ can have is $\lceil w(u)/2 \rceil - 2d^{h-2} + 1$ and the largest value that $w(u_1)$ can have is $\lceil w(u)/2 \rceil$. Therefore, $w(u_2)$ is bounded as follows: $\lfloor w(u)/2 \rfloor \leq w(v_2) \leq \lfloor w(u)/2 \rfloor + 2d^{h-2} - 1$. Because $d > 4$, we continue to satisfy the properties of Definition 3. In particular, at least $\Theta(d^h)$ insertions or deletions are needed before either u_1 or u_2 is split.

Deletions. Deletions are similar to insertions. As before, we search down the tree to find which leaf w to delete. After deleting w, some ancestors of w may become unbalanced. That is, some ancestor node u at height h may have weight lower than $\frac{1}{2}d^{h-1}$. As before, we bring the ancestors of w into balance starting from the ancestors closest to the leaves. We merge u with one of its neighbors. After merging u, however, it might now have a weight larger than its upper bound, so we immediately split it into two nodes as described in the insertion algorithm. (This may alternatively be viewed as u stealing children from its neighbor.)

These properties are also satisfied by the skip list data structure of [26] in the expected sense.

Lemma 6 *Skip lists satisfy Properties 1 and 2 in the expected sense.*

We conjecture that if our cache-oblivious B-tree is built with a skip list instead of a weight-balanced B-tree, we obtain the same (expected) bounds. Because a more delicate analysis is required to analyze this structure, we opt for a deterministic structure.

Finally, by Lemma 2, we obtain the following corollary:

Corollary 7 *Searching in a strongly weight-balanced search tree stored in the van Emde Boas layout costs at most*

[4] In [8] there is also a *leaf parameter* $k > 0$, but we simply fix $k = 1$.

$O(1 + \log_B N)$ *memory transfers*.

2.3. Packed-Memory Maintenance

Now we describe a data structure that will help us maintain a *dynamic* van Emde Boas layout of a strongly weight-balanced search tree. The primary difficulty is that we must achieve two seemingly contradictory goals. On one hand, we should pack the nodes densely into memory to achieve locality of reference. On the other hand, we should leave enough extra space between the nodes to permit future insertions. We develop a *packed-memory structure* to resolve this dilemma.

In the *packed-memory problem*, we have N elements $x_1, x_2, \ldots, x_N$ to be stored in an array A of size $O(N)$. The elements have *precedence constraints* $x_1 \prec x_2 \prec \cdots \prec x_N$ which determine the order of the elements in memory. We must support two update operations: a new element may be inserted between two existing elements, and an existing element may be deleted. We must maintain the following invariants throughout the dynamic changes of the elements:

1. x_i precedes x_j in array A precisely if $x_i \prec x_j$.
2. The elements are evenly distributed in the array A. That is, any set of k contiguous elements $x_{i_1}, \ldots, x_{i_k}$ is stored in a contiguous subarray of size $O(k)$.

In order to maintain these invariants, elements must move in the array. A naïve solution is to maintain all N elements tightly packed in exactly N memory cells. Now traversing the data structure uses at most $\lceil N/B \rceil + 1$ memory transfers, which is within 1 of optimal. Unfortunately, a single insertion requires $\Theta(N/B)$ memory transfers, because all the elements may have to be moved to make room for one more.

Our solution has the following performance guarantees:

1. Scanning any set of k contiguous elements $x_{i_1}, \ldots, x_{i_k}$ uses $O(1 + k/B)$ memory transfers.
2. Inserting or deleting a new element uses $O(1 + \frac{\log^2 N}{B})$ amortized memory transfers.

Our solution is closely related to the paper of Itai, Konheim, and Rodeh [19]. They consider the problem of storing elements in an array to maintain the first invariant. Their cost measure is the number of elements touched, and they obtain an $O(\log^2 N)$ amortized bound. Willard [40, 41, 42] presents a more complicated data structure that achieves an $O(\log^2 N)$ worst-case bound.[5]

Described roughly, our packed-memory structure is as follows. When a window of the array becomes too full or too empty, we evenly spread out the elements within a larger window. For correctness and efficiency we must set the following parameters: (1) the window size, and (2) the thresholds determining when a window is too full or too empty.

[5]This problem is closely related to, but distinct from, the problem of answering linked-list order queries. See for example [13].

Because rebalancing simply involves a scan of elements, we achieve the desired bound on memory transfers:

Theorem 8 *The packed-memory problem can be solved in $O(1 + \frac{\log^2 N}{B})$ amortized memory transfers per insert and delete.*

Now whenever we require extra space for a new node to insert into the strongly weight-balanced search tree, the packed-memory structure makes room in $O(1 + \frac{\log^2 N}{B})$ amortized memory transfers.

3. Main Structure

Our main structure is the weight-balanced B-tree (described in Section 2.2) organized according to the van Emde Boas layout (described in Section 2.1). This section demonstrates how to update the weight-balanced B-tree structure, while maintaining the van Emde Boas layout and using few memory transfers.

There are four components to the cost of an update:

1. The initial cost of *searching* for the given element, which is $O(\log_B N)$ memory transfers by Lemma 2 in Section 2.1.
2. The cost of *making room* for nodes by using the packed-memory structure from Section 2.3. By Theorem 8, this cost is $O(1 + \frac{\log^2 N}{B})$ amortized memory transfers.
3. The cost of *updating pointers* to any nodes moved as a result of the packed-memory structure. In the data structure presented in Section 3.1, we will only be able to obtain the (poor) bound of $O(\log^2 N)$ amortized memory transfers. In Sections 3.2–3.4, we will use the idea of *buffer nodes* to improve this to $O(1 + \frac{\log B}{\sqrt{B}} \log^2 N)$ amortized memory transfers.
4. The remaining cost of modifying the structure of the tree to accomodate *splits and merges* caused by an update. This is the topic of Section 3.1. We will show that this cost is $O(1 + \frac{\log N}{B})$ amortized memory transfers.

At the end of this section, we will obtain the following:

Theorem 9 *The data structure with buffer nodes maintains an ordered set, subject to searches in $O(\log_B N)$ memory transfers, scans of k elements in $O(1 + k/B)$ memory transfers, and insertions and deletions in $O(\log_B N + \frac{\log B}{\sqrt{B}} \log^2 N)$ amortized memory transfers.*

Corollary 10 *If $B = \Omega((\log N)^2 (\log \log N)^4)$, then insertions and deletions take $O(\log_B N)$ amortized memory transfers, which matches the B-tree bound.*

3.1. Splits and Merges

In this section, we outline the entire insertion and deletion algorithms for our structure, and then we analyze the cost of splits and merges (Cost 4 above). Splits and merges cause significant changes to the van Emde Boas layout. This

means that we need to move large blocks of nodes around in memory to maintain the proper layout.

An insertion or deletion into a weight-balanced B-tree consists of splits and merges along a leaf-to-root path, starting at a leaf and ending at a node at some height. We will show how to split or merge a node at height h in $O(1 + d^h/B)$ memory transfers plus the memory transfers from a single packed-memory insertion or deletion. By the last property in Definition 3, the amortized split-merge cost of rebalancing a node v is $O(1/B)$ memory transfers per insertion or deletion into the subtree rooted at v. When we insert or delete an element, this element is added or removed in $O(\log N)$ such subtrees. Hence, the split-merge cost of an update is $O(1 + \frac{\log N}{B})$ amortized memory transfers.

Next we describe the algorithm to split a node v. First, if v is the root of the tree, we insert a new root node at the beginning of the file, and add parent-child pointers. Because of the rounding scheme in the van Emde Boas layout, the rest of the layout does not change when the height of the tree changes.

Second, we insert a new node v' into the packed-memory structure, immediately after v (Cost 2 above). The packed-memory insert may cause several nodes to move in memory, in which case we also update the pointers to those nodes (Cost 3 above). Then we redistribute the pointers among v and v' according to the split algorithm of weight-balanced B-trees, using $O(1)$ memory transfers.

Third, we need to repair the van Emde Boas layout. Consider the coarsest level of detail in which v is the root of a recursive subtree S. Suppose S has height h', which can only be smaller than h, the height of node v. Let S be composed of top recursive subtree A of height $h' - \lfloor\!\lfloor h'/2 \rfloor\!\rfloor$ and bottom recursive subtrees $B_1, \ldots, B_k$ each of height $\lfloor\!\lfloor h'/2 \rfloor\!\rfloor$; refer to Figure 2. The split algorithm recursively splits A into A' and A''. (In the base case, A is the singleton tree $\{v\}$ and is already split into $\{v\}$ and $\{v'\}$.)

At this point, A' and A'' are next to each other. Now we must move them to the appropriate locations in the van Emde Boas layout. Let $B_1, \ldots, B_i$ be the children recursive subtrees of A', and let $B_{i+1}, \ldots, B_k$ be the children recursive subtrees of A''. We need to move $B_1, \ldots, B_i$ in between A' and A''. This move is accomplished by three linear scans. Specifically, we scan to copy A'' to some temporary space, then we scan to copy $B_1, \ldots, B_i$ immediately after A', overwriting A'', and then we scan to copy the temporary space containing A'' to immediately after B_i.

Now that A'' and $B_1, \ldots, B_i$ have been moved, we need to update the pointers to the nodes in these blocks. First we scan through the nodes in A', and update the child pointers of the leaves to point to the new locations of $B_1, \ldots, B_i$. That is, we increase the pointers by $||A''||$, the amount of space occupied in the packed-memory structure by A'', including unused nodes. Second we update the parent pointers of $B_{i+1}, \ldots, B_k$ to A'', decreasing them by $||B_1|| + \cdots + ||B_k||$. Finally, we scan the recursive subtrees of height h' that are children of $B_1, \ldots, B_i$, and update the parent pointers of the roots, decreasing them by $||A''||$. This can be done in a single scan because the children recursive subtrees of $B_1, \ldots, B_i$ are stored contiguously.

Finally, we analyze the number of memory transfers made by moving blocks at all levels of detail. At each level h' of the recursion, we perform a scan of all the nodes at most 6 times (3 for the move, and 3 for the pointer updates). By Property 2, these scans cost at most $O(1+d^{h'}/B)$ memory transfers. The total split-merge cost is given by the cost of recursing on the top recursive subtree of at most half the height, and by the cost of the 6 scans. This recurrence is dominated by the top level:

$$T(h') \le T(h'/2) + c\left(1 + \frac{d^{h'}}{B}\right) \le c\left(1 + \frac{d^{h'}}{B}\right) + O\left(\frac{d^{h'/2}}{B}\right).$$

Hence, the cost of a split is $O(1 + d^{h'}/B) \le O(1 + d^h/B)$ memory transfers, plus the cost of a packed-memory insertion.

Merges can be performed within the same memory-transfer bound by using the same overall algorithm. In the beginning, we merge two nodes and apply a packed-memory deletion. In each step of the recursion, we perform the above algorithm in reverse, i.e., the opposite transformation from Figure 2.

Therefore, Cost 4 is small:

Lemma 11 *The split-merge cost is $O(1 + \frac{\log N}{B})$ amortized memory transfers per insertion or deletion.*

3.2. Local Versus Long-Distance Nodes

So far we have shown that Costs 1, 2, and 4 are reasonably small, $O(\log_B N + \frac{\log^2 N}{B})$ amortized memory transfers. It remains to show how to bound Cost 3, the cost of updating pointers to nodes moved by the packed-memory structure. Superficially it appears sufficient to apply the packed-memory structure to move nodes, and we have already bounded this cost by $O(\frac{\log^2 N}{B})$. Unfortunately, updating pointers to nodes that have moved is often significantly more expensive. This is because the moved nodes are always consecutive in memory, but the nodes that point to them (parents and children) may each be in a different memory block. Thus, we may incur an additional memory transfer for each pointer update, for an (unimpressive) bound of $O(\log^2 N)$ amortized memory transfers per update in the packed-memory structure.

In order to provide a bound on the number of additional memory transfers, we define two classes of nodes, *local nodes* and *long-distance nodes*. Informally, a local node is a node whose immediate family (parent and children) are within distance B in memory, and hence in the same or abutting memory block. Otherwise, a node is a long-distance node. (This terminology is based on the telephone

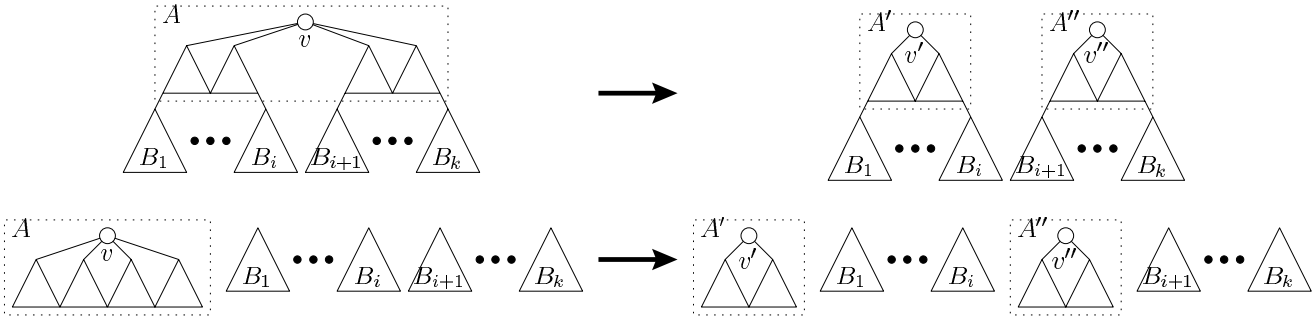

Figure 2. Splitting a node. The top shows the modification in the recursive subtree S, and the bottom shows the modification in the van Emde Boas layout.

system: it is cheaper to call your parents and children if the call is local instead of long-distance.)

In order to identify which nodes are local nodes and which nodes may be long-distance nodes, we examine the van Emde Boas layout at the appropriate level of detail. We consider the coarsest level of detail ℓ such that the *leaf* recursive subtrees (those containing leaves of the tree) have at most B nodes.

Lemma 12 *The leaf recursive subtrees at level of detail ℓ have $\Omega(\sqrt{B})$ nodes.*

We distinguish three types of nodes within each recursive subtree, the *root*, the *leaves*, and the *internal nodes*. Refer to Figure 3. All internal nodes are local nodes, because their immediate family is in the same recursive subtree. The root of each recursive subtree may be a long-distance node, because its parent is in a different recursive subtree. The leaves of the whole tree (i.e., the leaves of the leaf recursive subtrees) are all local nodes. The leaves of nonleaf recursive subtrees may be long-distance nodes, because their children are in different recursive subtrees.

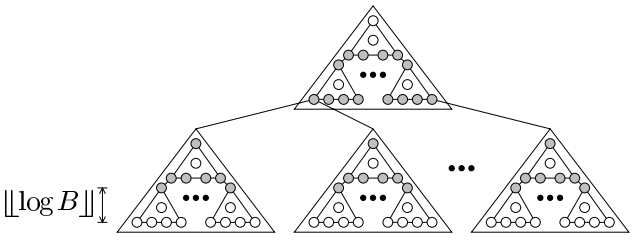

Figure 3. Local nodes (white) and long-distance nodes (gray).

In the nonleaf recursive subtrees, half of the nodes are long-distance nodes. To simplify the analysis, we treat all nodes in nonleaf recursive subtrees as long-distance nodes.

Lemma 13 *Within the subtree rooted at any node in a strongly weight-balanced search tree, $O(1/\sqrt{B})$ of the nodes are long-distance nodes.*

In particular, in the entire tree a fraction of $O(1/\sqrt{B})$ of the nodes are long-distance nodes. If the long-distance nodes were evenly distributed in the van Emde Boas layout, we could immediately obtain a bound of $O(\frac{\log^2 N}{\sqrt{B}})$ on Cost 3. However, local nodes are clustered near the leaves, and long-distance nodes are also clustered; see Figure 4. This clustering is not directly a problem because updates in the strongly weight-balanced search tree are concentrated near the leaves. Unfortunately, some leaves are located near large clusters of long-distance nodes. Therefore, when we insert a leaf into the packed-memory structure, it may move many long-distance nodes, where each move costs an additional memory transfer.

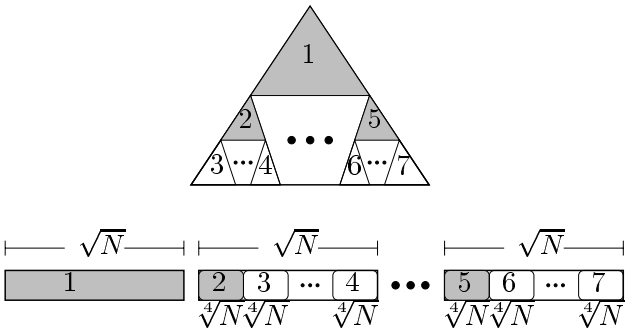

Figure 4. Distribution of local and long-distance nodes in the van Emde Boas layout. Shaded regions are mostly long-distance nodes, and the clear regions are local nodes. In this example, regions $3, 4, 6, 7$ are leaf recursive subtrees at level of detail ℓ.

3.3. Buffer Nodes

We need to prevent long-distance nodes from being moved around too much by the packed-memory structure. Because we do not know the memory block size B, we do not know which nodes are local and which are long-distance. Thus we need to protect all nodes that might be long-distance nodes. We do this by inserting *buffers* of dummy nodes around each cluster of potential long-distance nodes. These *buffer nodes* do not participate in the tree structure (i.e., they have

no parent and child pointers), so they are automatically local nodes and can be moved cheaply by the packed-memory structure.

Buffer nodes are placed as follows. Recall that the van Emde Boas layout of a subtree T is the layout of the top recursive subtree A, followed by the layouts of the bottom recursive subtrees $B_1, \ldots, B_k$. Depending on the memory block size, it may be that we are at precisely the level of detail ℓ, i.e., the B_i's are the leaf recursive subtrees. In this case, recursive subtree A has mostly long-distance nodes, so we must separate it from adjacent recursive subtrees of local nodes. For example, at level of detail 1 in Figure 1, $\{17, 18, 19\}$ is a recursive tree of mostly long-distance nodes, and it is adjacent to leaf recursive subtrees $\{14, 15, 16\}$ and $\{20, 21, 22\}$.

Our solution is to add two buffers, each with d^{h-1}/h nodes, one immediately before A and the other immediately after A. Now A has been separated from its neighbors, so we recursively lay it out with the normal van Emde Boas layout. This may have been too coarse a level of detail, so we recursively layout the bottom recursive subtrees $B_1, \ldots, B_k$ using the buffered van Emde Boas layout.

We have carefully chosen the size of the buffers to satisfy two seemingly conflicting properties. First, the buffers are much larger (nearly quadratically) than the top recursive subtree they barrier. Second, the total size of any recursive subtree is still linear in the number of contained tree nodes, so the query bound from Lemma 2 still holds.

Lemma 14 *The buffers adjacent to top recursive subtree A have size $\Omega(|A|^2/\log|A|)$.*

Lemma 15 *The total size of buffers, contained in a recursive subtree at any level of detail, is linear in the number of tree nodes in that subtree. In particular, the data structure has linear size.*

Proof: Focus on the recursive subtree S of interest, and let h be its height. Associate the buffers surrounding A with the root of A. Recall that A is layed out using the normal van Emde Boas layout, without recursive buffers. Thus, each node in S is associated with at most two buffers. Furthermore, only nodes at height h, $\lceil h/2 \rceil$, $\lceil h/2 \rceil/2$, $\lceil h/2 \rceil/4$, ... are associated with buffers. Thus, it suffices to count nodes at those heights and multiply them by the size of the corresponding buffers. By strong weight-balancedness, the number of descendants of a node at height h is at least $\frac{1}{2}d^{h-1}$, so the number of nodes at height h is at most $2|S|/d^{h-1}$. Hence the total size of buffers at height 2^i within S is at most $(2|S|/d^{2^i-1})(2d^{2^i-1}/2^i) = 4|S|/2^i$. Summing over all i, the total size of all buffers in S is at most $4|S| + 4|S|/2 + 4|S|/4 + 4|S|/8 + \cdots \leq 8|S|$. □

Next we describe how to maintain buffers during splits and merges. Before a split, there is a buffer before A and after A. We leave the first buffer before A, i.e., A', and move the second buffer to after A''. We move half of the buffer before A' to after A' (by performing a block swap in three linear scans), and move half of the buffer after A'' to before A''. Then we double the sizes of the buffers by inserting buffer nodes into their *middles* using the packed-memory structure. Similarly, a merge is done as follows. First we move the buffer after A' to before A', and move the buffer before A'' to after A'', using linear scans. Then we halve the sizes of the buffers by deleting buffer nodes from their middles, using the packed-memory structure.

3.4. Cost of Updating Pointers

In this section, we show that buffers reduce the cost of updating pointers, which is the culmination of our analysis.

Lemma 16 *Consider a node v that is either in a leaf recursive subtree in level of detail ℓ or a buffer node in the middle of a buffer. Consider any interval I of packed memory containing v. Then the number of long-distance nodes in I is at most $O(1 + \frac{\log B}{\sqrt{B}}|I|)$.*

Proof: Consider building the interval by walking from v. By construction, before we encounter a cluster of x long-distance nodes, we will first hit a cluster of $\Omega(x^2/\log x)$ buffer nodes, which are local nodes. Hence, the ratio of long-distance to local nodes is $O(\frac{\log x}{x})$, which is largest when x is small. The smallest cluster of long-distance nodes has size $x = \Omega(\sqrt{B})$, so the worst possible ratio is $O(\frac{\log B}{\sqrt{B}})$. □

Lemma 17 *The pointer-update cost is $O(1 + \frac{\log B}{\sqrt{B}} \log^2 N)$ amortized memory transfers per insertion or deletion.*

Proof: An insertion or deletion in the tree consists of splits and/or merges in the tree, which cause insertions and/or deletions in the packed-memory structure. We analyze the cost of these packed-memory updates separately in three cases, and the total cost is their sum.

1. A node v is inserted or deleted in the packed-memory structure within a leaf recursive subtree. The packed-memory structure moves some interval I of nodes containing v. By Lemma 16, there are $O(1 + \frac{\log B}{\sqrt{B}}|I|)$ long-distance nodes in I. Thus, the amortized cost of a packed-memory move is $O(\frac{\log B}{\sqrt{B}})$ memory transfers. By the last property in Definition 3, the amortized number of packed-memory updates in this region is $O(1)$ per tree update. The product of these two values is $O(\frac{\log B}{\sqrt{B}})$.
2. A node v is inserted or deleted in the packed-memory structure in a nonleaf recursive subtree. In this case, the cost of a packed-memory move is $O(1)$ memory transfers. By the last property in Definition 3 and by Lemma 12, the amortized number of packed-memory updates in this region is $O(1/\sqrt{B})$ per tree update. The product is $O(1/\sqrt{B})$.

3. A node v is inserted or deleted in the packed-memory structure in the middle of a buffer. Similar to Case 1, by Lemma 16, the amortized cost of a packed-memory move is $O(\frac{\log B}{\sqrt{B}})$ memory transfers. By the last property in Definition 3 and by Lemma 15, the amortized number of packed-memory updates in this region is $O(1)$ per tree update. The product is $O(\frac{\log B}{\sqrt{B}})$.

By Theorem 8, the amortized number of packed-memory moves per packed-memory update is $O(\log^2 N)$. Combining with the above products, the total amortized cost per tree update is $O(1 + \frac{\log B}{\sqrt{B}} \log^2 N)$ memory transfers. □

This lemma concludes the proof of the update bounds in Theorem 9.

3.5. Queries

Scans can be supported optimally by introducing cousin pointers between adjacent leaves. This modification does not increase the pointer-update cost by more than $O(1)$ memory transfers. For if an interval of leaves is moved by the packed-memory structure, only the pointers in those leaves and their two neighbors are affected.

Fingers can be implemented in a similar way, without changing the update bounds, by adding cousin pointers to all nodes. The analysis of the additional pointer-update cost is left to the full paper.

4. Using Indirection

First we show how to use one level of indirection to reduce the bounds in Theorem 9 to the bounds we claimed in the introduction. We store the data structure in two arrays. The *leaf array* stores all N of the elements, logically grouped into blocks of size $\Theta(\log N)$. The *tree array* stores the cache-oblivious search-tree structure from Section 3 containing the first element in each block of the leaf array, for a total of $\Theta(N/\log N)$ elements. Each leaf of the tree array points to the corresponding block in the leaf array (child pointers), but there are no pointers from the leaf array to the tree array (parent pointers).

We use one of two structures to store the leaf array, depending on the data structure that we ultimately want to build. Either we maintain the leaf-array blocks ordered into one file using the packed-memory structure, or we store them unordered in arbitrary locations in the array. In either case, insertions and deletions to the structure cause local modifications to blocks. Whenever a block becomes full or a constant-fraction empty, we split or merge it, respectively, causing an insertion or deletion into the tree array. Thus, only a fraction of $O(1/\log N)$ of the insertions and deletions into the data structure cause modifications to the tree array. Consequently the amortized cost of updating the tree array is $O(\log_B N) + O(\frac{\log B}{B} \log N) = O(\log_B N)$. A query takes $O(\log_B N)$ memory transfers to search through the tree array, plus $O(\frac{\log N}{B})$ memory transfers to scan through a block in the leaf array.

If we wish to support *scanning* k consecutive elements in $O(1 + k/B)$ time, then the blocks in the leaf array must be maintained consecutively using the packed-memory structure. As a result, inserting or deleting an element in a leaf-array block costs $O(\frac{\log^2 N}{B})$ amortized memory transfers to move nodes. Because moving k elements only affects $O(1 + k/\log N)$ pointers to the leaf array, we can update these pointers in $O(1+k/(B \log N))$ time by scanning leaves of the tree array.

Theorem 18 *The indirect data structure with ordered leaf blocks maintains an ordered set, subject to searches in $O(\log_B N)$ memory transfers, insertions and deletions in $O(\log_B N + \frac{\log^2 N}{B})$ amortized memory transfers, and scanning k consecutive elements in $O(1 + k/B)$ memory transfers.*

If scanning is not required, we can instead store the blocks in the leaf array in an arbitrary order. This change means that we no longer need the packed-memory structure, so the update cost matches that of B-trees:

Theorem 19 *The indirect data structure with unordered leaf blocks maintains an ordered set, subject to searches in $O(\log_B N)$ memory transfers, and insertions and deletions in $O(\log_B N)$ amortized memory transfers.*

In fact, we can use two levels of indirection to avoid the use of buffer nodes. The top level stores $O(N/\log^2 N)$ nodes in the search-tree structure from Section 3.1. The middle and bottom levels are composed of blocks of $\Theta(\log N)$ nodes. Depending on how blocks are maintained, we obtain the bounds in Theorem 18 or Theorem 19.

5. Conclusion

We have presented cache-oblivious search-tree data structures that perform searches optimally. One data structure is stored in a single file, permitting optimal scans and searches from any node in the tree. The insertion and deletion costs for this data structure match the B-tree bound for $B = \Omega((\log N)^2(\log\log N)^4)$. A second data structure uses indirection and performs scans optimally. For this structure, insertions and deletions match the B-tree bound for $B = \Omega(\log N \log\log N)$. A third data structure also uses indirection, and insertions and deletions always match the B-tree bound. We believe this work to be an important step in the emerging area of dynamic irregular data structures in the cache-oblivious model.

Acknowledgments. We gratefully acknowledge Charles Leiserson for suggesting this problem to us.

References

[1] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for organization of information (in Russian). *Doklady Akademii Nauk SSSR*, 146:263–266, 1962.

[2] A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir. A model for hierarchical memory. In *Proc. 19th ACM Sympos. Theory of Computing*, pp. 305–314, New York, May 1987.

[3] A. Aggarwal, A. K. Chandra, and M. Snir. Hierarchical memory with block transfer. In *Proc. 28th IEEE Sympos. Found. Comp. Sci.*, pp. 204–216, Los Angeles, Oct. 1987.

[4] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Comm. ACM*, 31(9):1116–1127, Sept. 1988.

[5] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2–3):72–109, 1994.

[6] M. Andrews, M. A. Bender, and L. Zhang. New algorithms for the disk scheduling problem. In *Proc. 37th IEEE Sympos. Found. Comp. Sci.*, pp. 580–589, Oct. 1996.

[7] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proc. 4th Internat. Workshop on Algorithms and Data Structures*, LNCS 955, pp. 334–345, Kingston, Canada, Aug. 1995.

[8] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *Proc. 37th IEEE Sympos. Found. Comp. Sci.*, pp. 560–569, Burlington, VT, Oct. 1996.

[9] R. D. Barve and J. S. Vitter. A theoretical framework for memory-adaptive algorithms. In *Proc. 40th IEEE Sympos. Found. Comp. Sci.*, pp. 273–284, New York, Oct. 1999.

[10] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, Feb. 1972.

[11] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proc. 8th ACM Sympos. Parallel Algorithms and Architectures*, pages 297–308, June 1996.

[12] COMPAQ. `http://ftp.digital.com/pub/Digital/info/semiconductor/literature/dsc-li%brary.html`. 1999.

[13] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proc. 19th ACM Sympos. Theory of Computing*, pp. 365–372, New York, May 1987.

[14] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. System Sci.*, 47(3):424–436, 1993.

[15] M. Frigo. A fast Fourier transform compiler. *ACM SIGPLAN Notices*, 34(5):169–180, May 1999.

[16] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th IEEE Sympos. Found. Comp. Sci.*, pp. 285–297, New York, Oct. 1999.

[17] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th IEEE Sympos. Found. Comp. Sci.*, pp. 8–21, Ann Arbor, Michigan, 1978.

[18] J.-W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proc. 13th ACM Sympos. Theory of Computation*, pp. 326–333, Milwaukee, Wisconsin, May 1981.

[19] A. Itai, A. G. Konheim, and M. Rodeh. A sparse table implementation of priority queues. In *Proc. 8th Colloquium on Automata, Languages, and Programming*, LNCS 115, pp. 417–431, July 1981.

[20] A. LaMarca and R. Ladner. The influence of caches on the performance of sorting. In *Proc. 8th ACM-SIAM Symposium on Discrete Algorithms*, pp. 370–379, Jan. 1997.

[21] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*, theorem 5, pages 198–199. Springer-Verlag, 1984.

[22] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM J. Comput.*, 2:33–43, 1973.

[23] M. Nodine and J. Vitter. Greed sort: Optimal deterministic sorting on parallel disks. *Journal of the ACM*, 42(4):919–933, July 1995.

[24] M. H. Overmars. *The Design of Dynamic Data Structures*, LNCS 156. Springer-Verlag, 1983.

[25] H. Prokop. Cache-oblivious algorithms. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, June 1999.

[26] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. In *Proc. Workshop on Algorithms and Data Structures*, LNCS 382, pp. 437–449, Ottawa, Canada, Aug. 1989.

[27] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–29, 1994.

[28] J. E. Savage. Extending the Hong-Kung model to memory hierachies. In *Proc. 1st Internat. Conf. Computing and Combinatorics*, LNCS 959, pp. 270–281, Aug. 1995.

[29] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16(4–5):464–497, 1996.

[30] S. Sen and S. Chatterjee. Towards a theory of cache-efficient algorithms. In *Proc. 11th ACM-SIAM Sympos. Discrete Algorithms*, pp. 829–838, San Francisco, Jan. 2000.

[31] R. C. Singleton. Mixed radix fast fourier tranforms. In *Programs for Digital Signal Processing*. IEEE Press, 1979.

[32] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Comm. ACM*, 28(2):202–208, 1985.

[33] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985.

[34] S. Toledo. Locality of reference in LU decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4):1065–1081, Oct. 1997.

[35] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proc. 16th IEEE Sympos. Found. Comp. Sci.*, pp. 75–84, Berkeley, California, 1975.

[36] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Systems Thoery*, 10(2):99–127, 1977.

[37] J. S. Vitter. External memory algorithms. LNCS 1461, 1998.

[38] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.

[39] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica*, 12(2–3):148–169, 1994.

[40] D. E. Willard. Maintaining dense sequential files in a dynamic environment. In *Proc. 14th ACM Sympos. Theory of Computing*, pp. 114–121, San Francisco, May 1982.

[41] D. E. Willard. Good worst-case algorithms for inserting and deleting records in dense sequential files. In *Proc. 1986 ACM SIGMOD Internat. Conf. Management of Data*, pp. 251–260, Washington, D.C., May 1986.

[42] 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.

[43] A. C. C. Yao. Should tables be sorted? *Journal of the Association for Computing Machinery*, 28(3):615–628, 1981.