Cache-Oblivious B-Trees*

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

Abstract

This paper presents two dynamic search trees attaining near-optimal performance on any hierarchical memory. The data structures are independent of the parameters of the memory hierarchy, e.g., the number of memory levels, the block-transfer size at each level, and the relative speeds of memory levels. The performance is analyzed in terms of the number of memory transfers between two memory levels with an arbitrary block-transfer size of B; this analysis can then be applied to every adjacent pair of levels in a multilevel memory hierarchy. Both search trees match the optimal search bound of $\Theta(1 + \log_{B+1} N)$ memory transfers. This bound is also achieved by the classic B-tree data structure on a two-level memory hierarchy with a known block-transfer size B. The first search tree supports insertions and deletions in $\Theta(1 + \log_{B+1} N)$ amortized memory transfers, which matches the B-tree's worst-case bounds. The second search tree supports scanning S consecutive elements optimally in $\Theta(1 + S/B)$ memory transfers, and supports insertions and deletions in $\Theta(1 + \log_{B+1} N + \frac{\log^2 N}{B})$ amortized memory transfers, matching the performance of the B-tree for $B = \Omega(\log N \log \log N)$.

1 Introduction

The memory hierarchies of modern computers are becoming increasingly steep. Typically, an L1 cache access is 2 orders of magnitude faster than a main memory access and 6 orders of magnitude faster than a disk access [COM99]. Thus, it is dangerously inaccurate to design algorithms assuming a flat memory with uniform access times.

Many computational models attempt to capture the effects of the memory hierarchy on the running times of algorithms. There is a tradeoff between the accuracy of the model and its ease of use. One body of work explores multilevel memory hierarchies [AACS87, ACS87, ACFS94, ABZ96, ABZ02, RW94, Sav95, Vit01, VS94b], though the proliferation of parameters in these models makes them cumbersome for algorithm design. A second body of work concentrates on two-level memory hierarchies, either main memory and disk [AV88, BV99, HK81, Vit01, VS94a] or cache and main memory [LL97, SC00]. With these models the programmer must anticipate which level of the memory hierarchy is the bottleneck, resulting in a program that is less flexible to different-scale problems and that does not adapt when the bottleneck changes, e.g., as a program starts to page into virtual memory.

*A preliminary version of this paper appeared in FOCS 2000 [BDFC00].

†Department of Computer Science, State University of New York, Stony Brook, NY 11794-4400, USA, email: bender@cs.sunysb.edu. Supported in part by HRL Laboratories, ISX Corporation, Sandia National Laboratories, and NSF grants EIA-0112849 and CCR-0208670.

‡MIT Computer Science and Artificial Intelligence Laboratory, 200 Technology Square, Cambridge, MA 02139, email: edemaine@mit.edu. Supported in part by NSF grant EIA-0112849.

§Department of Computer Science, Rutgers University, Piscataway, NJ 08855, USA; email: farach@cs.rutgers.edu. Supported by NSF grant CCR-9820879.

1.1 Cache-Oblivious Algorithms

The cache-oblivious model enables us to reason about a simple two-level memory but prove results about an unknown multilevel memory. This model was introduced by Frigo, Leiserson, Prokop, and Ramachandran [FLPR99, Pro99]. They show that 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 [BFJ+96, Tol97] and static binary search [Pro99]. 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 et al. [FLPR99, Pro99]. In the ideal-cache model there are two levels in the memory hierarchy, called *cache* and *main memory*, although they could represent any pair of levels. Main memory is partitioned into *memory blocks*, each consisting of a fixed number B of consecutive cells.[1] The cache has size M, and consequently has capacity to store M/B memory blocks. In this paper, we require only that M/B be greater than a sufficiently large constant. The cache is *fully associative*, that is, it can contain an arbitrary set of M/B memory blocks at any time.

The parameters B and M are unknown to the cache-oblivious algorithm or data structure. As a result, the algorithm cannot explicitly manage memory, and this burden is taken on by the system. When the algorithm accesses a location in memory that is not stored in cache, the system fetches the relevant memory block from main memory, in what is called a *memory transfer*. If the cache is full, a memory block is elected for replacement, based on an optimal offline analysis of the future memory accesses of the algorithm.

While this model may superficially seem unrealistic, Frigo et al. show that it can be simulated by essentially any memory system with a small constant-factor overhead. Thus, if we run a cache-oblivious algorithm on a multilevel memory hierarchy, we can use the ideal-cache model to analyze the number of memory transfers between each pair of adjacent levels. See [FLPR99, Pro99] for details.

The concept of algorithms that are uniformly optimal across multiple memory models was considered previously by Aggarwal, Alpern, Chandra, and Snir [AACS87]. 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 give algorithms for matrix multiplication and FFT 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 include block transfers, though [ACS87] later extended the HMM to the BT model to take into account block transfers. 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.

1.2 B-Trees

In this paper, we initiate the study of cache-oblivious data structures by developing cache-oblivious search trees.

The classic I/O-efficient search tree is the *B-tree* [BM72]. The basic idea is to maintain a balanced tree of N elements with node fanout proportional to the memory block size B. Thus, one

[1] Note that B and M are parameters, not constants. Consequently, they must be preserved in asymptotic notation in order to obtain accurate running time estimates.

block read determines the next node out of $\Theta(B)$ nodes, so a search completes in $\Theta(1 + \log_{B+1} N)$ memory transfers. A simple information-theoretic argument shows that this bound is optimal.

The B-tree is designed for a two-level hierarchy, and the situation becomes more complex with more than two levels. We need a multilevel structure, one level per transfer block size. Suppose $B_1 > B_2 > \cdots > B_L$ are the block sizes between the $L+1$ 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. Even when it is possible to determine all these parameters, such a data structure is cumbersome. Also, each level of recursion incurs a constant-factor wastage in storage, leading to suboptimal memory-transfer performance for $L = \omega(1)$.

1.3 Results

We develop two cache-oblivious search trees. These results are the first demonstration that even irregular and dynamic problems, such as data structures, can be solved efficiently in the cache-oblivious model. Since the conference version of this paper appeared, many other data-structural problems have been addressed in the cache-oblivious model; see Table 1.

Our results achieve the memory-transfer bounds listed below. N denotes the number of elements stored in the tree. *Updates* refer to both key insertions and deletions.

1. The first cache-oblivious search tree attains the following memory-transfer bounds:

   Search: $O(1 + \log_{B+1} N)$, which is optimal and matches the search bound of B-trees.

   Update: $O(1 + \log_{B+1} N)$ amortized, which matches the update bound of B-trees, though the B-tree bound is worst case.

2. The second cache-oblivious search tree adds the *scan* operation (also called the *range search* operation). Given a key x and a positive integer S, the scan operation accesses S elements in key order, starting after x. The memory-transfer bounds are as follows:

   Search: $O(1 + \log_{B+1} N)$.

   Scan: $O(1 + S/B)$, which is optimal.

   Update: $O(1 + \log_{B+1} N + \frac{\log^2 N}{B})$ amortized, which matches the B-tree update bound of $O(1 + \log_{B+1} N)$ when $B = \Omega(\log N \log\log N)$.

   This last relation between B and N usually holds in external memory and often does not hold in internal memory.

In the development of these data structures, we build and identify tools for cache-oblivious manipulation of data. These tools have since been used in many of the cache-oblivious data structures listed in Table 1. In Section 2.1, we show how to linearize a tree according to what we call the *van Emde Boas layout*, along the lines of Prokop's static search tree [Pro99]. In Section 2.2, we describe a type of *strongly weight-balanced search tree* [AV96] useful for maintaining locality of reference. Following the work of Itai, Konheim, and Rodeh [IKR81] and Willard [Wil82, Wil86, Wil92], 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 $O(1 + \frac{\log^2 N}{B})$ amortized memory transfers; see Section 2.3. In another context, this structure can be thought of as a cache-oblivious *linked list* that supports scanning S consecutive elements in $O(1 + S/B)$ memory transfers (instead of the naïve $O(S)$), and updates in $O(1 + \frac{\log^2 N}{B})$ amortized memory transfers.

B-tree	• Simplification via packed-memory structure/low-height trees	[BDIW02, BFJ02]
	• Simplification and persistence via exponential structures	[RCR01, BCR02]
	• Implicit	[FG03a, FG03b]
Static search trees	• Basic layout	[Pro99]
	• Experiments	[LFN02]
	• Optimal constant factor	[BBF+03]
Linked lists supporting scans		[BCDFC02]
Priority queues		[ABD+02b, BF02b]
Trie layout		[ABD+02a, BDFC02]
Computational geometry	• Distribution sweeping	[BF02a]
	• Voronoi diagrams	[KR03]
	• Orthogonal range searching	[AADHM03]
Lower bounds		[BF03]

Table 1: Related work in cache-oblivious data structures. These results, except the static search tree of [Pro99], appeared after the conference version of this paper.

1.4 Notation

We introduce a notational convention of possible independent interest. We define the *hyperfloor* of x, denoted $\lfloor\lfloor x \rfloor\rfloor$, to be $2^{\lfloor \log x \rfloor}$, i.e., the largest power of 2 smaller than x.[2] 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 $\lceil\lceil\lceil x \rceil\rceil\rceil = 2^{\lceil\lceil \log x \rceil\rceil}$, which round to the nearest powers of powers of 2. 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 the dynamic structures. Consider a $O(\lg N)$-height search tree in which every node has at least two and at most a constant number of children and in which all leaves are on the same level. We describe a mapping from the nodes of the tree to positions in memory. The cost of any search in this layout is $\Theta(1 + \log_{B+1} N)$ memory transfers, which is optimal up to constant factors. Our layout is a modified version of Prokop's layout for a complete binary tree whose height is a power of 2 [Pro99, pp. 61–62]. We call the layout the *van Emde Boas layout* because it resembles the recursive structure in the van Emde Boas data structure [vE75, vEKZ77].[3]

The van Emde Boas layout proceeds recursively. Let h be the height of the tree, or more precisely, the number of levels of nodes in the tree. Suppose first that h is a power of 2. 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, B_2, …, B_ℓ each of height $h/2$. If all nonleaf nodes have the same number of children, then the recursive

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

[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.

subtrees all have size roughly $\sqrt{N}$, and ℓ is roughly $\sqrt{N}$. The layout of the tree is obtained by recursively laying out each subtree, and combining these layouts in the order A, B_1, B_2, ..., B_ℓ; see Figure 1.

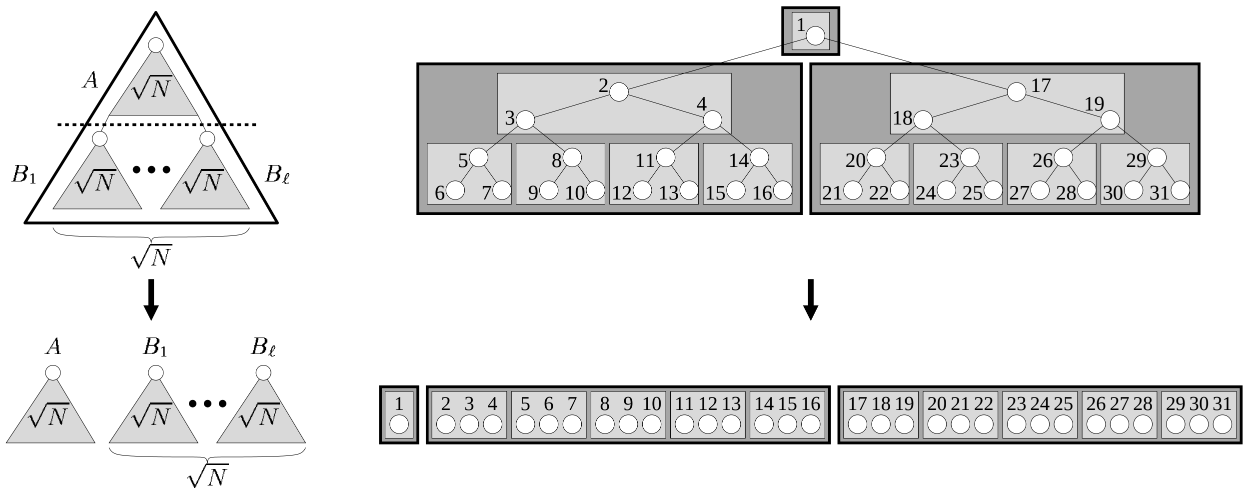

Figure 1: The van Emde Boas layout (left) in general and (right) of a tree of height 5.

If h is not a power of 2, we assign a number of levels that is a power of 2 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 $h - \lceil\!\lceil h/2 \rceil\!\rceil$. This rounding scheme is important for later dynamic structures because the heights of the cut lines in the lower trees do not vary with N. In contrast, this property is not shared by the simple rounding scheme of assigning $\lfloor h/2 \rfloor$ levels to the top recursive subtree and $\lceil h/2 \rceil$ levels to the bottom recursive subtrees.

The memory-transfer analysis views the van Emde Boas layout at a particular *level of detail.* Each level of detail is a partition of the tree into disjoint recursive subtrees. In the finest level of detail, 0, each node forms its own recursive subtree. In the coarsest level of detail, $\lceil \log_2 h \rceil$, the entire tree forms the unique recursive subtree. Level of detail k is derived by starting with the entire tree, recursively partitioning it as described above, and exiting a branch of the recursion upon reaching a recursive subtree of height $\leq 2^k$. The key property of the van Emde Boas layout is that, at any level of detail, each recursive subtree is stored in a contiguous block of memory.

One useful consequence of our rounding scheme 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.*

Proof: The proof follows from a simple induction on the level of detail. Consider a tree T of height h. At the coarsest level of detail, $\lceil \log_2 h \rceil$, there is a single recursive subtree, which includes the root. In this case the lemma is trivial. Suppose by induction that the lemma holds for level of detail k. In this level of detail the recursive subtree containing the root of T has height h', where $1 \leq h' \leq 2^k$, and all other recursive subtrees have height 2^k. To progress to the next finer level of detail, $k-1$, all recursive subtrees that do not contain the root are recursively split once more so that they have height 2^{k-1}. If the height h' of the top recursive subtree is at most 2^{k-1}, then it is not split in level of detail $k-1$. Otherwise, the root is split into bottom recursive subtrees of height 2^{k-1} and a top recursive subtree of height $h'' \leq 2^{k-1}$. The inductive step follows. □

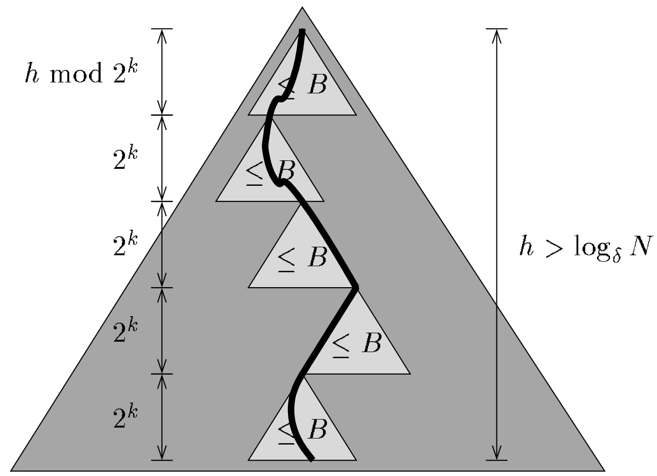

Figure 2: The recursive subtrees visited by a root-to-leaf search path in level of detail k.

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 $4\left\lceil \log_\delta \Delta \log_{B+1} N + \log_{B+1} \Delta \right\rceil = O(1 + \log_{B+1} N)$ memory transfers.*

Proof: Let k be the coarsest level of detail such that every recursive subtree contains at most B nodes; see Figure 2. Thus, every recursive subtree is stored in at most 2 memory blocks. Because tree T has height h, $\lceil h/2^k \rceil$ recursive subtrees are traversed in each search, and thus at most $2\lceil h/2^k \rceil$ memory blocks are transfered. Because the tree has height h, $\delta^{h-1} < N < \Delta^h$, that is, $\log_\Delta N < h < \log_\delta N + 1$. Because a tree of height 2^{k+1} has more than B nodes, $\Delta^{2^{k+1}} > B$ so $\Delta^{2^{k+1}} \geq B + 1$. Thus, $2^k \geq \frac{1}{2} \log_\Delta (B+1)$. Therefore, the maximum number of memory transfers is

$$2\left\lceil \frac{h}{2^k} \right\rceil \leq 4\left\lceil \frac{1 + \log_\delta N}{\log_\Delta (B+1)} \right\rceil = 4\left\lceil \left(1 + \frac{\log N}{\log \delta}\right)\left(\frac{\log \Delta}{\log (B+1)}\right) \right\rceil = 4\left\lceil \log_\delta \Delta \log_{B+1} N + \log_{B+1} \Delta \right\rceil.$$

Because δ and Δ are constants, this number of memory transfers is $O(1 + \log_{B+1} N)$. □

2.2 Strongly Weight-Balanced Search Trees

To convert the static layout into a dynamic layout, we use a dynamic balanced search tree. We require two properties of the balanced search tree:

Property 1 (Descendant Amortization) *Suppose that whenever we* rebalance *a node v (i.e., modify it to keep balance) we also* touch *all of v's descendents. Then the amortized number of elements touched per insertion is $O(\log N)$.*

Property 2 (Strong Weight Balance) *For some constant d, every node v at height h has $\Theta(d^h)$ descendants.*

Property 1 is normally implied by Property 2 as well as by a weaker property called weight balance. A tree is *weight balanced* if, for every node v, its left subtree (including v) and its right subtree (including v) have sizes that differ by at most a constant factor. Weight balancedness 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, strong weight balance requires an absolute constraint that relates the sizes of all subtrees at the same level. For example, BB[α] trees [NR73] are weight-balanced binary search trees based on rotations, but they are not strongly weight balanced.

Search trees that satisfy Properties 1 and 2 include weight-balanced B-trees [AV96], deterministic skip lists [MPS92], and skip lists [Pug89] in the expected sense. We choose to use weight-balanced B-trees:

Definition 3 (Weight-Balanced B-tree [AV96]) *A rooted tree 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 $w(u)$ *of u is the number of nodes in the subtree rooted at u. This weight is bounded as follows:*
$$\frac{d^{h-1}}{2} \le w(u) \le 2d^{h-1}.$$
4. Amortization: *If a nonroot node u at height h is rebalanced, then $\Omega(d^h)$ updates are necessary before u is rebalanced again. That is, $w(u) - d^{h-1}/2 = \Theta(d^h)$ and $2d^{h-1} - w(u) = \Theta(d^h)$.*[5]

Properties 1–4 have the following consequence:

5. *The root has between 2 and $4d$ children. All internal nodes have between $d/4$ and $4d$ children. The height of the tree is $O(1 + \log_d N)$.*

From the strong weight balancedness of the subtree rooted at a node, we can conclude strong weight balancedness of the top a levels in such a subtree:

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

Proof: Let T denote the subtree rooted at a node of height h. Consider the descendants of v down precisely a levels, and let $B_1, B_2, \ldots, B_k$ be the subtrees rooted at those nodes. In other words, the B_i's are the child subtrees of A. Let $b = h - a$ denote the height of each B_i. Because T and the B_i's are strongly weight balanced, $|T| \le 2d^{h-1}$ and $|B_i| \ge \frac{1}{2}d^{b-1}$ for all i. Now $|B_1| + \cdots + |B_k| < |T| \le 2d^{h-1}$, so $k \le 4d^{h-b} = 4d^a$. But the number of nodes in A is less than the number of children subtrees $B_1, B_2, \ldots, B_k$, so the result follows. □

Next we show how to perform updates, making a small modification to the presentation in [AV96]. Specifically, [AV96] performs deletes using the global rebalancing technique of [Ove83], where deleted nodes are treated as "ghost" nodes to be removed when the tree is periodically re-assembled. In the cache-oblivious model, we need to service deletes immediately to avoid large holes in the structure.

[4] In [AV96] there is also a *leaf parameter* $k > 0$, but we simply fix $k = 1$.

[5] This property is not included in the definition in [AV96], but it is an important invariant satisfied by the structure.

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 greater than $2d^{h-1}$. We bring the ancestors of w into balance starting from the ancestors closest to the leaves. 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, $v_1, \ldots, v_k$. 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, $\lceil w(u)/2 \rceil - 2d^{h-2} + 1 \leq w(u_1) \leq \lceil w(u)/2 \rceil$ and $\lfloor w(u)/2 \rfloor \leq w(u_2) \leq \lfloor w(u)/2 \rfloor + 2d^{h-2} - 1$. 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 $\frac{1}{2}d^{h-1}$. We merge u with one of its neighbors. After merging u, 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. As in insertions, we handle such out-of-bound nodes u in order of increasing height.

2.3 Packed-Memory Maintenance

A *packed-memory structure* maintained N elements in order in an array of size $P = cN$ subject to element insertion between two existing elements and element deletion. The remaining fraction $1 - c$ of the array is blank. The packed-memory structure must achieve two seemingly contradictory goals. On one hand, we should pack the nodes densely so that scanning is fast: S consecutive elements must occupy $O(S)$ cells in the array so that scanning those elements uses $O(1 + S/B)$ memory transfers. On the other hand, we should leave enough blank space between the nodes to permit future insertions. We achieve the following balance:

Theorem 5 *For any desired $c > 1$, the packed-memory structure maintains N elements in an array of size cN and supports insertions and deletions in $O(1 + \frac{\log^2 N}{B})$ amortized memory transfers and scanning S consecutive elements in $O(1 + S/B)$ memory transfers.*

Our data structure and analysis closely follow Itai, Konheim, and Rodeh [IKR81]. They consider the same problem of maintaining elements in order in an array of linear size, but in a different cost model and without the scanning requirement. Their structure moves $O(\log^2 N)$ amortized elements per insertion, but has no guarantee on the number of memory transfers for inserting, deleting, or scanning. This structure has been de-amortized by Willard [Wil82, Wil86, Wil92] and subsequently simplified by Bender et al. [BCD+02].[6]

At a high level, the packed-memory structure keeps every interval of size $\Omega(1)$ a constant fraction full, where the constant fraction depends on the interval size. When an interval of the array becomes too full or too empty, we evenly spread out (*rebalance*) the elements within a larger interval. It remains to specify the size of the interval to rebalance and the thresholds determining when an interval is too full or too empty.

[6] This problem is closely related to, but distinct from, the problem of answering linked-list order queries [DS87, BCD+02].

Tree Structure and Thresholds. We divide the array into segments each of size $\Theta(\lg P)$ so that the number of segments is a power of 2. We then implicitly build a perfect binary tree on top of these $\Theta(P/\lg P)$ segments, making each segment a leaf. Each node in the tree represents the subarray containing the segments in the subtree rooted at this node. In particular, the root node represents the entire array and each leaf node represents a single segment. to be a power of 2.

To define the thresholds controlling rebalance, we need additional terminology. The *capacity* of a node u in the tree, denoted capacity(u), is the size of its subarray. The *density* of a node u in the tree, denoted density(u), is the number of elements stored in u's subarray divided by capacity(u). Define the root node to have depth 0 and the leaf nodes to have depth $d = \lg \Theta(P/\lg P)$.

For each node, we define two *density thresholds* specifying the desired range on the node's density. Let $0 < \rho_d < \rho_0 < \tau_0 < \tau_d = 1$ be arbitrary constants. For a node u at depth k, the *upper-bound density threshold* τ_k is $\tau_0 + \frac{\tau_d - \tau_0}{d} k$ and the *lower-bound density threshold* ρ_k is $\rho_0 - \frac{\rho_0 - \rho_d}{d} k$. Thus,

$$0 < \rho_d < \rho_{d-1} < \cdots < \rho_0 < \tau_0 < \tau_1 < \cdots < \tau_d = 1.$$

A node u at depth k is *within threshold* if $\rho_k \le \text{density}(u) \le \tau_k$.

The main difference between our packed-memory structure and the data structure of [IKR81] is that we add lower-bound thresholds.

Insertion and Deletion. To insert an element x, we proceed as follows. First we find the leaf node w where x belongs. If leaf w has free space, then we *rebalance* w by evenly distributing x and the $\Theta(\lg P)$ elements in the leaf. Otherwise, we proceed up the tree until we find the first ancestor u of w that is within threshold. Then we *rebalance* node u by evenly distributing all elements in u's subarray throughout that subarray. Leaf w is now within threshold and we can insert x as before.

Deleting an element is similar. To delete an element x, we remove x from the leaf node w containing x. If w is still within threshold, we are done. Otherwise, we find the lowest ancestor u of w that is within threshold and we rebalance u. Leaf w is now within threshold.

Although we describe these algorithms as conceptually visiting nodes in a tree, the tree is not actually stored. The operations are implemented by two parallel scans, one moving left and one moving right, that visit the subarrays specified implicitly by the tree nodes along a leaf-to-root path. During these scans we count the number of elements found and the capacity traversed, stopping when the density of a subarray is within threshold. The memory-transfer cost of the scans is $O(1 + K/B)$ where K is the capacity of the subarray traversed. We then rebalance this subarray using $O(1)$ additional scans, so the total cost is $O(1 + K/B)$; K is also the capacity of the subarray rebalanced.

We can support the number N of elements growing and shrinking by recopying the structure whenever the size changes by a constant factor.

Analysis. Next we bound the amortized size of a rebalance during an insertion or deletion. Suppose that we rebalance a node u at depth k. The rebalance was trigged by an insertion or deletion in the subarray of some child v of u. Before rebalancing, u is within threshold, i.e., $\rho_k \le \text{density}(u) \le \tau_k$, and v is not within threshold, i.e., density$(v) > \tau_{k+1}$ or density$(v) < \rho_{k+1}$. After rebalancing, the density of v (and of v's sibling) is not only within threshold, but also within the density thresholds of its parent u, i.e., $\rho_k \le \text{density}(v) \le \tau_k$. Before the density of node v (or its sibling) next exceeds its upper-bound threshold, we must have at least $(\tau_{k+1} - \tau_k)\,\text{capacity}(v)$ additional insertions within its subarray. Similarly, before the density of node v (or its sibling) next falls below its lower-bound threshold, we must have at least $(\rho_k - \rho_{k+1})\text{capacity}(v)$ additional deletions within its subarray.

We charge the size capacity(u) of the rebalance at u to its child v. Therefore the amortized size of a rebalance per insertion into v's subarray is

$$\frac{\text{capacity}(u)}{\text{capacity}(v)\ (\tau_{k+1}-\tau_k)} = \frac{2}{\tau_{k+1}-\tau_k} = \frac{2d}{\tau_d-\tau_0} = O(\lg P),$$

and the amortized size of a rebalance per deletion in v's subarray is

$$\frac{\text{capacity}(u)}{\text{capacity}(v)\ (\rho_k-\rho_{k+1})} = \frac{2}{\rho_k-\rho_{k+1}} = \frac{2d}{\rho_0-\rho_d} = O(\lg P).$$

When we insert or delete an element x, we insert or delete within $d = O(\lg P)$ different subintervals containing x. Therefore the total amortized size of a rebalance per insertion or deletion is $O(\lg^2 P) = O(\lg^2 N)$.

Because the amortized size of a rebalance per insertion or deletion is $O(\lg^2 N)$, the amortized number of memory transfers per insertion or deletion is $O(1 + \frac{\lg^2 N}{B})$. This concludes the proof of Theorem 5.

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. **Searching** — The initial cost of searching for the given element, which is $O(1 + \log_{B+1} N)$ memory transfers by Lemma 2 in Section 2.1.

2. **Minding the Gaps** — The cost of rearranging gaps to make room or reduce excess space around the updated node in the packed-memory structure from Section 2.3. By Theorem 5, this cost is $O(1 + \frac{\log^2 N}{B})$ amortized memory transfers. In Section 3.2, we use indirection to reduce this cost when we do not need to support scans optimally.

3. **Updating Pointers** — The cost of updating pointers to any nodes moved as a result of the packed-memory structure. In Section 3.1, we obtain the poor bound of $O(\log^2 N)$ amortized memory transfers. In Section 3.2, we use two levels of indirection to reduce the cost to $O(1)$ amortized memory transfers.

4. **Splits and Merges** — The remaining cost of modifying the structure of the tree to accommodate splits and merges caused by an update. In Section 3.1 we show that this cost is $O(1 + \frac{\log N}{B})$ amortized memory transfers, which is negligible.

3.1 Splits and Merges

In this section, we outline the insertion and deletion algorithms for our structure and analyze the cost of splits and merges (Cost 4). Splits and merges significantly change the van Emde Boas layout. To compensate, we 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 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. Thus, we must pay to mind the gaps (Cost 2). The packed-memory insert may also cause several nodes to move in memory, in which case we must update the pointers to those nodes (Cost 3). 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 $B_1, B_2, \ldots, B_k$ each of height $\lfloor\lfloor h'/2 \rfloor\rfloor$; refer to Figure 3. 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 $B_1, B_2, \ldots, B_i$ be the children recursive subtrees of A', and let $B_{i+1}, B_{i+2}, \ldots, B_k$ be the children recursive subtrees of A''. We need to move $B_1, B_2, \ldots, B_i$ 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 $B_1, B_2, \ldots, B_i$ immediately after A', overwriting A'', and then we scan to copy the temporary space containing A'' to immediately after B_i.

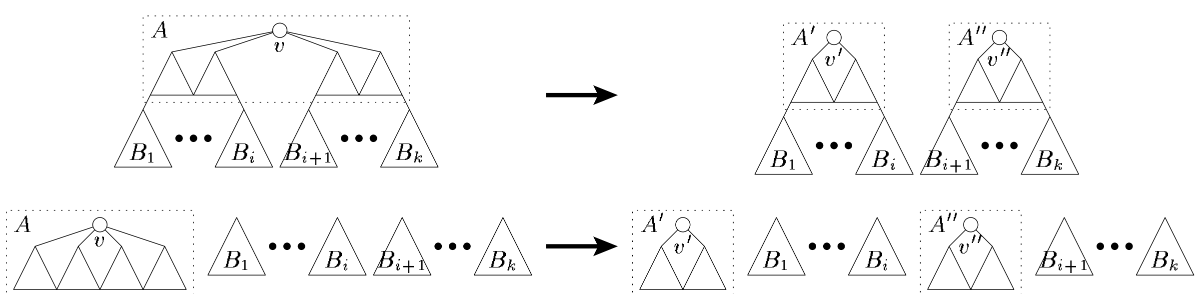

Figure 3: Splitting a node. The top shows the modification in the recursive subtree S, and the bottom shows the modification in the van Emde Boas layout.

Now that A'' and $B_1, B_2, \ldots, B_i$ 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 $B_1, B_2, \ldots, B_i$. 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 $B_1, B_2, \ldots, B_i$, 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 3.

Therefore, the split-and-merge cost (Cost 4) is small:

Lemma 6 *The split-merge cost is $O(1 + \frac{\log N}{B})$ amortized memory transfers per insertion or deletion.*

3.2 Using Indirection

We use two levels of indirection to reduce the cost of updating pointers in the search structure of the previous section, and we thereby attain the bounds claimed in the introduction. The resulting data structure has three layers.

The bottom layer stores all N elements, clustered into $\Theta(N/\log N)$ groups of $\Theta(\log N)$ consecutive elements each. Associated with each of these bottom groups is a *representative element*, the minimum element in the group. A representative element may in fact be a *ghost element* which has been deleted but cannot be removed from the structure because it is used in higher layers.

The middle layer stores the $\Theta(N/\log N)$ representative elements from the bottom layer (some of which may be ghost elements). The middle layer may also store ghost elements which have been deleted from the bottom layer but are still representatives in the middle layer. The middle elements are clustered into $\Theta(N/\log^2 N)$ groups of $\Theta(\log N)$ consecutive elements each. Again we elect the minimum element of each middle group as a representative element.

The top layer stores the $\Theta(N/\log^2 N)$ representative elements from the middle layer (some of which may be ghost elements).

The layers are stored according to the following data structures. The top layer is implemented by the search structure from the previous section. The middle layer is implemented by a single packed-memory structure, where the representative elements serve as markers between groups. The bottom layer is implemented by a packed-memory structure if we require optimal scan operations. Otherwise, the bottom layer is implemented by an unordered collection of groups, where the elements in each group are stored in an arbitrary order within a contiguous region of memory. These memory regions are all of the same size, and are allowed to be a constant fraction empty. Elements are inserted or deleted in a group simply by rewriting the entire group.

Elements appearing on multiple layers have “down” pointers from each instantiation to the instantiation at the next lower layer. Elements appearing on both the top and middle layers have "up" pointers from the middle instantiation to the top instantiation. However, elements appearing on both the middle and bottom layers do not have up pointers from the bottom layer to the middle layer.

To search for an element in this data structure, we search in the top layer for the query element or, failing that, its predecessor. Then we follow the pointer to the corresponding group in the middle layer, and scan through this middle group to find the query element or its predecessor. If the found element is a ghost element of the middle layer that has been deleted from the bottom layer, we move to the previous element in the middle layer. Finally, we follow the pointer to the corresponding group in the bottom layer, and scan through this bottom group, searching for the query element.

When an element is inserted, it is added to a bottom group according to order; if the inserted element could fit in two bottom groups, we favor the smaller of the two groups. Thus, a freshly inserted element never becomes the new representative element of its group. A group in the bottom or middle layer may become too full, in which case we split the group evenly into two groups, create a new representative element for the second group, and insert this representative element into the next level up. Similarly, a group in the bottom or middle layer may become too empty (by a fixed constant fraction less than $1/2$), in which case we merge the group with an adjacent group and delete the larger representative element from the next level up. A merge may cause an immediate split.

Next we analyze the performance of searches and updates in two versions of the data structure. The *ordered B-tree* stores the bottom layer as a packed-memory structure, and therefore supports scans optimally. The *unordered B-tree* stores the bottom layer as an unordered collection of groups.

Lemma 7 *In both B-tree structures, a search uses $O(1 + \log_{B+1} N)$ memory transfers.*

Proof: A search examines each layer once from the top down. Searching through the tree in the top layer costs $O(1 + \log_{B+1} N)$ memory transfers by Lemma 2. Scanning through a group in the middle or bottom layer costs $O(1 + \frac{\log N}{B})$ memory transfers because each group contains $O(\log N)$ elements and is stored contiguously. The cost at the top layer dominates. □

Lemma 8 *In the ordered B-tree, an update uses $O(1 + \log_{B+1} N + \frac{\log^2 N}{B})$ memory transfers.*

Proof: An update (insertion or deletion) examines each layer once from the bottom up. At the bottom layer, we pay $O(1 + \frac{\log^2 N}{B})$ amortized memory transfers to rebalance the packed-memory structure to preserve constant-size gaps (Cost 2, minding the gaps). Of the $O(\log^2 N)$ amortized nodes that move during this rebalance on the bottom layer, $O(\log N)$ amortized nodes are also present on the middle layer, and we must update the down pointers from these nodes on the middle layer to the moved nodes on the bottom layer. To update these pointers, we scan the bottom and middle layers in parallel, with the middle-layer scan advancing at a relative speed of $\Theta(1/\log N)$. These pointer updates cost $O(1 + \frac{\log^2 N}{B})$ amortized memory transfers.

A bottom group causes a split or merge after $\Omega(\log N)$ updates to that group since the last split or merge. When such a split or merge occurs, we pay $O(1 + \frac{\log^2 N}{B})$ amortized memory transfers to rebalance the packed-memory structure on the middle layer. This split-or-merge cost can be charged to the $\Omega(\log N)$ updates that caused it, reducing the amortized cost by a factor of $\Omega(\log N)$. Thus, we pay only $O(1 + \frac{\log N}{B})$ amortized memory transfers per update. Of the $O(\log^2 N)$ amortized nodes that move during this rebalance on the middle layer, $O(\log N)$ amortized nodes are also present on the top layer, and we must update the down pointers from these nodes on the top layer to the moved nodes on the middle layer. To update these pointers, we scan the moved nodes on the middle layer, follow each node's up pointer if it is present, and update the down pointer of each node reached. These pointer updates cost at most $O(\log N + \frac{\log^2 N}{B})$ amortized memory transfers per split or merge, or $O(1 + \frac{\log N}{B})$ per update. (At this point we could also afford to update up pointers from the bottom layer to the middle layer at an amortized cost of $\Theta(1 + \frac{\log^3 N}{B})$ per split or merge, or $\Theta(1 + \frac{\log^2 N}{B})$ per update, but we do not need these up pointers, and furthermore the unordered B-tree cannot afford to maintain them.)

A middle group causes a split or merge after $\Omega(\log N)$ updates to that group, which correspond to $\Omega(\log^2 N)$ updates to the data structure. When such a split or merge occurs, we pay $O(\log^2 N)$ amortized memory transfers to update the top layer, which is only $O(1)$ amortized memory transfers per update to the data structure. Furthermore, the number of nodes moved in top layer is also $O(\log^2 N)$ amortized, so $O(\log^2 N)$ amortized up pointers from the middle layer to the top layer need to be updated. Thus the pointer-update cost is $O(\log^2 N)$ amortized memory transfers per split or merge, or $O(1)$ per update to the data structure.

Summing all costs, an update costs $O(1 + \frac{\log^2 N}{B})$ amortized memory transfers plus the cost of searching. □

Lemma 9 *In the unordered B-tree, an update uses $O(1 + \log_{B+1} N)$ amortized memory transfers.*

Proof: The analysis is nearly identical. The only difference is that we no longer rebalance a packed-memory structure on the bottom layer, and instead maintain an unordered collection of contiguous groups. As a result, we do not pay $O(1 + \frac{\log^2}{B})$ amortized memory transfers per update, neither for the packed-memory update nor for updating down pointers from the middle layer to the bottom layer. The additional cost of rewriting a bottom group during each update is $O(1 + \frac{\log N}{B})$ memory transfers. The cost of splitting and/or merging a bottom group is the same. Therefore, the total cost of an update is $O(1 + \frac{\log N}{B})$ amortized memory transfers plus the cost of searching. □

Combining Lemmas 7–9, we obtain the following main results:

Theorem 10 *The ordered B-tree maintains an ordered set subject to searches in $O(1 + \log_{B+1} N)$ memory transfers, insertions and deletions in $O(1 + \log_{B+1} N + \frac{\log^2 N}{B})$ amortized memory transfers, and scanning S consecutive elements in $O(1 + S/B)$ memory transfers.*

Theorem 11 *The unordered B-tree maintains an ordered set subject to searches in $O(1 + \log_{B+1} N)$ memory transfers, and insertions and deletions in $O(1 + \log_{B+1} N)$ amortized memory transfers.*

4 Extensions and Alternative Approaches

A previous version of this paper [BDFC00] presents a different cache-oblivious B-tree based on *buffer nodes*. One distinguishing aspect of this approach is that it uses no indirection, storing the tree in one packed-memory structure. This B-tree achieves an update bound of $O(1 + \log_{B+1} N + \frac{\log B}{\sqrt{B}} \log^2 N)$ amortized memory transfers. With one level of indirection, the B-tree's update bound reduces to $O(1 + \log_{B+1} N + \frac{\log^2 N}{B})$ amortized memory transfers while supporting optimal scans, or $O(1 + \log_{B+1} N)$ amortized memory transfers without optimal scans. Thus, we ultimately obtain the same bounds as the simpler B-trees presented above, though with one fewer level of indirection.

The basic idea of buffer nodes is to enable storing data of different "fluidity" in a single packed array, ranging from rapidly changing data that is cheap to update to slowly changing data that is expensive to update. In the context of trees, leaves are frequently updated but their pointers are to relatively nearby nodes, so updating these pointers is usually cheap, while nodes in e.g. the middle level are updated infrequently but their pointers are to disparate regions of memory. The buffer-node solution is to add a large number of extra (dataless) nodes in between data of different fluidity to "protect" the low-fluidity data from the frequent updates of high-fluidity data. In the context of trees, we add $O(N/\log N)$ buffer nodes in between the top recursive subtree A and bottom recursive subtrees $B_1, B_2, \ldots, B_\ell$. These buffers prevent the rebalance intervals in the packed-memory structure from touching the nodes in A for a long time, leading to an $O(\frac{\log B}{\sqrt{B}} \log^2 N)$ term in the update bound.

For any of the cache-oblivious B-tree structures, a natural question is whether the $O(\frac{\log^2 N}{B})$ term in the update bound can be removed while still supporting scans optimally. Effectively, in addition to a search-tree structure, we need a linked-list data structure for supporting fast scans. Recently, Bender et al. [BCDFC02] developed a cache-oblivious linked list that supports insertions and deletions in $O(1)$ memory transfers and scans of S consecutive elements in $O(1+S/B)$ amortized memory transfers. Using this linked list we obtain the following optimal bounds:

Corollary 12 [BCDFC02, Corollary 1] *There is a cache-oblivious data structure that maintains an ordered set subject to searches in $O(1 + \log_{B+1} N)$ memory transfers, insertions and deletions in $O(\log_B N)$ amortized memory transfers, and scanning S consecutive elements in $O(1 + S/B)$ amortized memory transfers.*

It remains open whether these bounds can be achieved in the worst case. See [BCDFC02, BCR02] for cache-oblivious B-trees achieving some worst-case bounds.

5 Conclusion

We have presented cache-oblivious B-tree data structures that perform searches optimally. The first data structure maintains the data in sorted order in an array with gaps, and it stores an auxiliary structure for searching within this array. The second data structure attains the $O(\log_{B+1} N)$ update bounds of B-trees but breaks the sorted order of elements, thus slowing sequential scans of consecutive elements. These cache-oblivious B-trees represent the first dynamic cache-oblivious data structures. The cache-oblivious tools presented in this paper play an important role in the dynamic cache-oblivious data structures that have followed.

Acknowledgments

We gratefully acknowledge Charles Leiserson for suggesting this problem to us. We thank the anonymous referees for their many helpful comments.

References

[AACS87] Alok Aggarwal, Bowen Alpern, Ashok K. Chandra, and Marc Snir. A model for hierarchical memory. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 305–314, New York, May 1987.

[AADHM03] Pankaj K. Agarwal, Lars Arge, Andy Danner, and Bryan Holland-Minkley. Cache-oblivious data structures for orthogonal range searching. In *Proceedings of 19th Annual ACM Symposium on Computational Geometry*, pages 237–245, San Diego, California, June 2003.

[ABD+02a] Stephen Alstrup, Michael A. Bender, Erik D. Demaine, Martin Farach-Colton, J. Ian Munro, Theis Rauhe, and Mikkel Thorup. Efficient tree layout in a multilevel memory hierarchy. arXiv:cs.DS/0211010, November 2002. `http://www.arXiv.org/abs/cs.DS/0211010`.

[ABD+02b] Lars Arge, Michael A. Bender, Erik D. Demaine, Bryan Holland-Minkley, and J. Ian Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, pages 268–276, Montréal, Canada, May 2002.

[ABZ96] Matthew Andrews, Michael A. Bender, and Lisa Zhang. New algorithms for the disk scheduling problem. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, pages 580–589, Burlington, Vermont, October 1996.

[ABZ02] Matthew Andrews, Michael A. Bender, and Lisa Zhang. New algorithms for the disk scheduling problem. *Algorithmica*, 32(2):277–301, February 2002.

[ACFS94] Bowen Alpern, Larry Carter, Ephraim Feig, and Ted Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2–3):72–109, August–September 1994.

[ACS87] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. Hierarchical memory with block transfer. In *Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science*, pages 204–216, Los Angeles, CA, October 1987.

[AV88] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.

[AV96] Lars Arge and Jeffrey Scott Vitter. Optimal dynamic interval management in external memory. In *Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Science*, pages 560–569, Burlington, VT, October 1996.

[BBF+03] Michael A. Bender, Gerth Stølting Brodal, Rolf Fagerberg, Dongdong Ge, Simai He, Haodong Hu, John Iacono, and Alejandro López-Ortiz. The cost of cache-oblivious searching. In *Proceedings of the 44st Annual Symposium on Foundations of Computer Science*, pages 271–282, Cambridge, Massachusetts, October 2003.

[BCD+02] Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *Proceedings of the 10th Annual European Symposium on Algorithms*, volume 2461 of *Lecture Notes in Computer Science*, pages 152–164, Rome, Italy, September 2002.

[BCDFC02] Michael A. Bender, Richard Cole, Erik D. Demaine, and Martin Farach-Colton. Scanning and traversing: Maintaining data for traversals in a memory hierarchy. In *Proceedings of the 10th Annual European Symposium on Algorithms*, volume 2461 of *Lecture Notes in Computer Science*, pages 139–151, Rome, Italy, September 2002.

[BCR02] Michael A. Bender, Richard Cole, and Rajeev Raman. Exponential structures for efficient cache-oblivious algorithms. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 195–207, Málaga, Spain, July 2002.

[BDFC00] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious B-trees. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 399–409, Redondo Beach, California, November 2000.

[BDFC02] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Efficient tree layout in a multilevel memory hierarchy. In *Proceedings of the 10th Annual European Symposium on Algorithms*, volume 2461 of *Lecture Notes in Computer Science*, pages 165–173, Rome, Italy, September 2002.

[BDIW02] Michael A. Bender, Ziyang Duan, John Iacono, and Jing Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 29–38, San Francisco, California, January 2002.

[BF02a] Gerth Stølting Brodal and Rolf Fagerberg. Cache oblivious distribution sweeping. In *Proceedings of the 29th International Colloquium on Automata, Languages, and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 426–438, Malaga, Spain, July 2002.

[BF02b] Gerth Stølting Brodal and Rolf Fagerberg. Funnel heap – a cache oblivious priority queue. In *Proceedings of the 13th Annual International Symposium on Algorithms and Computation*, volume 2518 of *Lecture Notes in Computer Science*, pages 219–228, Vancouver, Canada, November 2002.

[BF03] Gerth Stølting Brodal and Rolf Fagerberg. On the limits of cache-obliviousness. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, pages 307–315, San Diego, California, June 2003.

[BFJ+96] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 297–308, Padua, Italy, June 1996.

[BFJ02] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. Cache oblivious search trees via binary trees of small height. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 39–48, San Francisco, California, January 2002.

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, February 1972.

[BV99] Rakesh D. Barve and Jeffrey Scott Vitter. A theoretical framework for memory-adaptive algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 273–284, New York, October 1999.

[COM99] COMPAQ. `http://ftp.digital.com/pub/Digital/info/semiconductor/literature/dsc-library.html`. 1999.

[DS87] Paul F. Dietz and Daniel D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 365–372, New York City, May 1987.

[FG03a] Gianni Franceschini and Roberto Grossi. Optimal cache-oblivious implicit dictionaries. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming*, volume 2719 of *Lecture Notes in Computer Science*, pages 316–331, 2003.

[FG03b] Gianni Franceschini and Roberto Grossi. Optimal worst-case operations for implicit cache-oblivious search trees. In *Proceedings of the 8th Workshop on Algorithms and Data Structures*, 2003. To appear.

[FLPR99] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 285–297, New York, October 1999.

[HK81] Jia-Wei Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computation*, pages 326–333, Milwaukee, Wisconsin, May 1981.

[IKR81] Alon Itai, Alan G. Konheim, and Michael Rodeh. A sparse table implementation of priority queues. In S. Even and O. Kariv, editors, *Proceedings of the 8th Colloquium on Automata, Languages, and Programming*, volume 115 of *Lecture Notes in Computer Science*, pages 417–431, Acre (Akko), Israel, July 1981.

[KR03] Piyush Kumar and Edgar Ramos. I/O-efficient construction of Voronoi diagrams. Manuscript, 2003.

[LFN02] Richard E. Ladner, Ray Fortna, and Bao-Hoang Nguyen. A comparison of cache aware and cache oblivious static search trees using program instrumentation. In *Experimental Algorithmics: From Algorithm Design to Robust and Efficient Software*, volume 2547 of *Lecture Notes in Computer Science*, pages 78–92, 2002.

[LL97] Anthony LaMarca and Richard E. Ladner. The influence of caches on the performance of sorting. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 370–379, New Orleans, Louisiana, January 1997.

[MPS92] J. Ian Munro, Thomas Papadakis, and Robert Sedgewick. Deterministic skip lists. In *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 367–375, Orlando, Florida, January 1992.

[NR73] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM Journal on Computing*, 2:33–43, 1973.

[Ove83] Mark H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*. Springer-Verlag, 1983.

[Pro99] Harald Prokop. Cache-oblivious algorithms. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, June 1999.

[Pug89] William Pugh. Skip lists: a probabilistic alternative to balanced trees. In F. Dehne, J.-R. Sack, and N. Santoro, editors, *Proceedings of the Workshop on Algorithms and Data Structures*, volume 382 of *Lecture Notes in Computer Science*, pages 437–449, Ottawa, Ontario, Canada, August 1989.

[RCR01] Naila Rahman, Richard Cole, and Rajeev Raman. Optimised predecessor data structures for internal memory. In *Proceedings of the 5th International Workshop on Algorithm Engineering*, volume 2141 of *Lecture Notes in Computer Science*, pages 67–78, Aarhus, Denmark, August 2001.

[RW94] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–29, 1994.

[Sav95] John E. Savage. Extending the Hong-Kung model to memory hierachies. In *Proceedings of the 1st Annual International Conference on Computing and Combinatorics*, volume 959 of *Lecture Notes in Computer Science*, pages 270–281, August 1995.

[SC00] Sandeep Sen and Siddhartha Chatterjee. Towards a theory of cache-efficient algorithms. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 829–838, San Francisco, California, January 2000.

[Tol97] Sivan Toledo. Locality of reference in LU decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4):1065–1081, October 1997.

[vE75] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, pages 75–84, Berkeley, California, 1975.

[vEKZ77] Peter van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10(2):99–127, 1977.

[Vit01] Jeffrey Scott Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, June 2001.

[VS94a] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.

[VS94b] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica*, 12(2–3):148–169, 1994.

[Wil82] Dan E. Willard. Maintaining dense sequential files in a dynamic environment. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, pages 114–121, San Francisco, California, May 1982.

[Wil86] Dan E. Willard. Good worst-case algorithms for inserting and deleting records in dense sequential files. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pages 251–260, Washington, D.C., May 1986.

[Wil92] Dan 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, April 1992.