Available online at www.sciencedirect.com
SCIENCE DIRECT®
Journal of Algorithms 53 (2004) 115–136
Journal of Algorithms
www.elsevier.com/locate/jalgor

A locality-preserving cache-oblivious dynamic dictionary ☆

Michael A. Bender [a,1], Ziyang Duan [a,*], John Iacono [b], Jing Wu [a]

[a] *Department of Computer Science, State University of New York, Stony Brook, NY 11794-4400, USA*
[b] *Department of Computer and Information Science, Polytechnic University, 5 Metrotech Center, Brooklyn, NY 11201, USA*

Received 22 February 2002

Available online 8 July 2004

Abstract

This paper presents a simple dictionary structure designed for a hierarchical memory. The proposed data structure is *cache-oblivious* and *locality-preserving*. A cache-oblivious data structure has memory performance optimized for all levels of the memory hierarchy even though it has no memory-hierarchy-specific parameterization. A locality-preserving dictionary maintains elements of similar key values stored close together for fast access to ranges of data with consecutive keys.

The data structure presented here is a simplification of the cache-oblivious B-tree of Bender, Demaine, and Farach-Colton. The structure supports search operations on N data items using $O(\log_B N + 1)$ block transfers at a level of the memory hierarchy with block size B. Insertion and deletion operations use $O(\log_B N + \log^2 N/B + 1)$ amortized block transfers. Finally, the data structure returns all k data items in a given search range using $O(\log_B N + k/B + 1)$ block transfers.

This data structure was implemented and its performance was evaluated on a simulated memory hierarchy. This paper presents the results of this simulation for various combinations of block and memory sizes.

☆ This work appeared in preliminary form in the Proceedings of the 13th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA), pp. 29–38, January 2002.

* Corresponding author.
E-mail addresses: bender@cs.sunysb.edu (M.A. Bender), dziyang@cs.sunysb.edu (Z. Duan), jiacono@poly.edu (J. Iacono), jingwu@cs.sunysb.edu (J. Wu).

[1] Supported in part by NSF Grants ACI-032497, CCR-0208670, and EIA-0112849, HRL Laboratories, and Sandia National Laboratories.

0196-6774/$ – see front matter
doi:10.1016/j.jalgor.2004.04.014

1. Introduction

The B-tree [12,25,32,37] is the classic external-memory search tree, and it is widely used in both theory and practice. The B-tree is designed to support insert, delete, search, and scan on a two-level memory hierarchy consisting of main memory and disk. The basic structure is a balanced tree having a fan-out proportional to the disk-block size B. The B-tree uses linear space and its query and update performance are $O(\log_B N + 1)$ memory transfers. This is a $\Theta(\log B)$-factor improvement over the $O(\lg N)$ bound obtained by the RAM-model structures (e.g., [1,31,43,46,50,51]). This improvement translates to approximately an order of magnitude speedup, depending on the application.

Although B-trees are in widespread use, they have several limitations. They depend critically on the block size B and therefore are only optimized for two levels of the memory hierarchy. On the other hand, modern memory hierarchies often have many levels including registers, several levels of cache, main memory, and disk. Furthermore, the disparity in the access times of the levels is growing, and future memory hierarchies may have even more levels.

Theoretically, it is possible to create a multilevel B-tree, but the resulting structure is significantly more complex than the standard B-tree. The data structure must be carefully tuned for each memory level of interest. Furthermore, the amount of wasted space in such an implementation appears exponential in the number of levels.

In many applications, such as database management systems, it is recognized that the classic implementation of a B-tree can be optimized for modern memory hierarchies by improving the data layout. For example, many systems heuristically attempt to group logically close pages physically near each other in memory in order to improve the performance of the scan operation [26,35,36,41,48]. These efforts suggest that data locality is required at other levels of granularity besides single disk blocks.

1.1. The cache-oblivious model

Traditionally most algorithmic work assumes the *Random Access Model* (*RAM*), which consists of a "flat" memory with uniform access times. Recently, however, research has been performed on developing theoretical models for modern complicated hierarchical memory systems; see, e.g., [3–5,7,49,55,56].

In order to avoid the complications of multilevel memory models, a body of work has focused on two-level memory hierarchies. Arguably the most successful two-level model is the *Disk Access Model* (*DAM*) of Aggarwal and Vitter [6]. In the DAM, the memory hierarchy consists of an internal memory of size M and an external memory partitioned into B-sized blocks. The performance metric in this model is the number of block transfers.

Recently, a new model was proposed that combines the simplicity of the two-level models with the realism of more complicated hierarchical models. The *cache-oblivious model*, introduced by Frigo, Leiserson, Prokop, and Ramachandran [29,45], enables us to reason about a simple two-level memory model, but prove results about an unknown, multilevel memory model. The idea is to avoid any memory-specific parameterization, that is, to design algorithms that do not use any information about memory-access times or block sizes.

The theory of cache-oblivious algorithms is based on the *ideal-cache model* [29,45]. As in the DAM model, there are two levels in the memory hierarchy, which we call *cache* and *memory*, although they could represent any pair of levels. The main difference between the cache-oblivious and the DAM model is that parameters B and M are unknown to a cache-oblivious algorithm. This crucial difference forces cache-oblivious algorithms to be optimized for all values of B and M and for all levels of the memory hierarchy.

1.2. Our results

We propose a cache-oblivious and locality-preserving search tree, which is a simplification of the cache-oblivious B-tree of Bender, Demaine, and Farach-Colton [16]. At a level of the memory hierarchy with block size B, the number of block transfers during a search operation is $O(\log_B N + 1)$, which is asymptotically optimal. Insertions and deletions take $O(\log_B N + \log^2 N/B + 1)$ amortized memory transfers, while scans of k data items are performed asymptotically optimally with $O(k/B + 1)$ memory transfers. If the scan operation is not supported, our structure can be modified using indirection, as in [16], so that all operations use $O(\log_B N)$ amortized block transfers.

Like the cache-oblivious B-tree of [16], our data structure is locality preserving. Any range of k consecutive keys is stored in a contiguous region of memory of size $O(k)$. This layout facilitates scans and range queries on most architectures, where accessing sequential blocks is an order of magnitude faster than accessing random blocks [30].

Our structure can be easily modified using the method of Brown and Tarjan [23] to achieve $O(\log_B k)$ query times, where k is the difference in rank between the current and previous queries. This property of our structure, known as the *dynamic-finger property*, implies other finger-type results [33]. For example, given a constant-size subset F of the keys in the structure, let $d(x, y)$ be the difference in rank between x and y. The number of page faults to access x from F is then $O(\log_B \min_{f \in F} d(f, x))$.

Our data structure consists of two arrays. One of the arrays contains the data and a linear number of blank entries, and the other array contains an encoding of a tree that indexes the data. The search and update operations involve basic manipulations of these arrays.

We evaluated the algorithm on a simulated memory hierarchy. This paper presents the results of this simulation for various combinations of block and memory sizes.

1.3. Related work

The first cache-oblivious search tree was proposed by Bender, Demaine, and Farach-Colton [16]. Our data structures and the cache-oblivious B-tree of [16] have the same capabilities with exactly the same asymptotic performance. Specifically, both data structures support inserts, deletes, searches, and scans. Both data structures can be augmented using indirection to support slightly faster inserts and deletes, but at the cost of no longer supporting efficient scans. Both data structures can be augmented to support finger search, although the augmentation required in [16] is more complicated. The main advantage of the current data structure over [16] is that the our structure is easily implementable.

Table 1
Related work in cache-oblivious data structures

B-tree	• Simplification via packed-memory structure/low-height trees	[18,22]
	• Simplification and persistence via exponential structures	[15,47]
	• Implicit	[27,28]
Static search trees	• Basic layout	[45]
	• Experiments	[40]
	• Optimal constant factor	[13]
Linked lists supporting scans		[14]
Priority queues		[11,20]
Trie layout		[8,17]
Computational geometry	• Distribution sweeping	[19]
	• Voronoi diagrams	[39]
	• Orthogonal range searching	[2]
Lower bounds		[21]

Brodal, Fagerberg, and Jacob [22] independently developed a remarkably similar cache-oblivious search tree, whose bounds match those presented here. Their data structure maintains a balanced tree of height $\log N + O(1)$, which they lay out cache-obliviously in an array of size $\Theta(N)$. The two papers present complementary experimental results. Publication [22] gives timing results for searches and random inserts, but not worst-case updates. We evaluate the cost of updates for a range of insertion patterns and a range of memory hierarchies.

Rahman, Cole, and Raman [47] and Bender, Cole, and Raman [15] consider a different approach to building cache-oblivious search structures based on exponential search trees [9,10,52]. The paper [47] gives the first implementation of a cache-oblivious search tree. These data structures permit fast insertions but lack the ability to scan efficiently.

See Table 1 for more related work on cache-oblivious data structures.

A body of related work shows how to keep N elements ordered in $O(N)$ locations of memory, subject to insertions and deletions. Itai, Konheim, and Rodeh [34] examine the problem in the context of priority queues and propose a simple structure using $O(\log^2 N)$ amortized time per update. Similar results were obtained by Melville and Gries [42] and by Willard [57]. Willard [58–60] examines the problem in the context of dense file maintenance and develops a more complicated structure using $O(\log^2 N)$ worst-case time per update. Bender, Demaine, and Farach-Colton [16] show that a modification to the structure of Itai, Konheim, and Rodeh results in a packed-memory structure running in $O((\log^2 N)/B + 1)$ amortized memory transfers per update and $O(k/B + 1)$ memory transfers per traversal of k elements.

2. Description of the structure

Our data structure maintains a dynamic set S storing items with key values from a totally ordered universe. It supports the following operations:

1. INSERT(x): Adds x to S, i.e., $S = S \cup \{x\}$.
2. DELETE(x): Removes x from S, i.e., $S = S - \{x\}$.
3. PREDECESSOR(x): Returns the item from S that has the largest key value in S that is at most x, i.e., return $\max_{y \in S}$ such that $y \leqslant x$.
4. SCANFORWARD(): Returns the successor of the most recently accessed item in S.
5. SCANBACKWARD(): Returns the predecessor of the most recently accessed item in S.

We use two separate cache-oblivious structures, the packed-memory structure of Bender, Demaine, and Farach-Colton [16] (which is closely based upon previous structures of Itai, Kronheim, and Rodeh [34] and Willard [57–60]) and the static B-tree of Prokop [45].

The packed-memory structure maintains N elements in sorted order in an array of size $O(N)$ subject to insertions, deletions and scans. Insertions and deletions require $O(\log^2 N/B + 1)$ block transfers and a scan of k elements requires $O(k/B + 1)$ block transfers.

The packed-memory structure is used to store the items. However it does not support efficient searches. (A naïve binary search requires $O(\log(N/B) + 1)$ memory transfers, which is prohibitively large.) We thus use the static cache-oblivious tree structure as an index into the packed-memory structure, where each leaf in the static cache-oblivious tree corresponds to one item in the array of the packed-memory structure.

The difficulty with this fusion of structures is that when we insert or delete, the positions of the elements in the packed-memory structure may be adjusted, invalidating the keys in the static B-tree. Thus, the static B-tree must be updated to reflect the changes in the packed-memory structure. We show that the cost of updating the static B-tree does not dominate the insertion cost. Whenever the array becomes too full or too empty we recopy the elements into a larger or smaller array.

Before describing our main structure, we present the packed-memory structure and the static cache-oblivious layout.

2.1. Packed-memory maintenance

In a *packed-memory structure* [16], we have N totally ordered elements $x_1, x_2, \ldots, x_N$ to be stored in an array A of size $O(N)$. Two update operations are supported: a new element may be inserted between two existing elements, and an existing element may be deleted. This structure maintains the following invariants:

1. *Order constraint*: Element x_i precedes x_j in array A iff $x_i \prec x_j$.
2. *Density constraint*: The elements are evenly distributed in the array A. That is, any set of k contiguous elements $x_j, \ldots, x_{j+k-1}$ is stored in a contiguous subarray of size $\Theta(k)$.

The packed-memory structure of [16] has the following performance guarantees: Scanning any set of k contiguous elements $x_j, \ldots, x_{j+k-1}$ uses $O(k/B + 1)$ memory transfers. Inserting or deleting a new element uses $O(\log^2 N/B + 1)$ amortized memory transfers.

Described roughly, the packed-memory structure works as follows: When a window of the array becomes too unbalanced, with most of the elements in one half of the window, then we spread out the elements, evenly distributing the gaps. The window sizes range from $O(\log N)$ to $O(N)$ and are powers of 2. A window of size 2^i is a contiguous memory block of size 2^i whose left boundary has an array position that is a multiple of 2^i.

Associated with window sizes are *density thresholds*, which are guidelines to determine the acceptable densities of windows. The *upper-bound density threshold* of a window of size 2^k is denoted τ_k, where

$$\tau_{\log\log N} > \tau_{\log\log N+1} > \cdots > \tau_{\log N},$$

and the *lower-bound density threshold* of a window of size 2^k is ρ_k, where

$$\rho_{\log\log N} < \rho_{\log\log N+1} < \cdots < \rho_{\log N},$$

and $\tau_{\log N} > \rho_{\log N}$. The density of any particular window of size 2^k may deviate beyond its threshold, but as soon as the deviation is "discovered," the densities of the windows are adjusted to be within the threshold.

The values of the densities are determined according to an arithmetic progression. Specifically, let $\tau_{\log N}$ be any positive constant less than 1, and let $\tau_{\log\log N} = 1$. Let

$$\delta = (\tau_{\log\log N} - \tau_{\log N})/(\log N - \log\log N).$$

Then, define density threshold τ_k to be

$$\tau_k = \tau_{\log\log N} - (k - \log\log N)\delta.$$

Similarly, let $\rho_{\log N}$ be any constant less than $\tau_{\log N}$, and let $\rho_{\log\log N}$ be any constant less than $\rho_{\log N}$. Let

$$\delta' = (\rho_{\log N} - \rho_{\log\log N})/(\log N - \log\log N).$$

Then, define density threshold ρ_k to be

$$\rho_k = \rho_{\log\log N} + (k - \log\log N)\delta'.$$

We say that a window of the array of size 2^k is *overflowing* if the number of data elements in the region exceeds $\tau_k 2^k$. We say that a window of the array of size 2^k is *underflowing* if the number of data elements in the region is less than $\rho_k 2^k$.

To *insert* (*delete*) an element x at location $A[j]$, we examine the windows containing $A[j]$ of size 2^k, for $k = \log\log N, \ldots, \log N$ until the smallest window is found that is not overflowing or underflowing. We then insert (delete) the element x and *rebalance* this window. To rebalance the window, we rearrange the elements and the window so that the gaps are evenly spaced. The simplest way to implement a rebalance is first to compress the elements on one side of the window, and then to redistribute the elements throughout the window. Thus, each element is moved at most twice.

2.2. Static structure

We review a cache-oblivious *static* tree structure of Prokop [45], which is used in most cache-oblivious search structures. Given a complete binary tree, we describe a mapping

from the nodes of the tree to positions of an array. This mapping, called *van Emde Boas layout*, resembles the recursive structure in the van Emde Boas data structure [53,54]. The cache oblivious structure can perform any traversal from the root to a leaf in an N node tree in $O(\log_B N + 1)$ memory transfers, which is asymptotically optimal.

We now describe the van Emde Boas layout. Suppose the tree contains N items and has height $h = \lg(N + 1)$. Conceptually split the tree at the middle level of edges, below the nodes of height $h/2$. This split breaks the tree into the *top recursive subtree* A of height $\lfloor h/2 \rfloor$, and several *bottom recursive subtrees* $B_1, \ldots, B_\ell$ of height $\lceil h/2 \rceil$. Thus there are $\ell = \Theta(\sqrt{N})$ bottom recursive subtrees and each subtree contains $\Theta(\sqrt{N})$ nodes. The mapping of the nodes in each subtree to positions in memory is obtained by recursively laying out each subtree and combining these layouts in the order $A, B_1, \ldots, B_\ell$ in an array. The base case is reached when the trees have one node.

We now introduce the notion of *levels of detail* to partition the tree into disjoint recursive subtrees. In the finest level of detail, 0, each node is its own recursive subtree. In the coarsest level of detail, $\lceil \lg h \rceil$, the entire tree forms a unique recursive subtree. In general, level of detail k is derived by starting with the entire tree, recursively partitioning it, and exiting the recursion whenever a recursive subtree has height $\leqslant 2^k$. Note that according to the van Emde Boas layout, each recursive subtree is stored in a contiguous block of memory. At level of detail k, all recursive subtrees have heights between 2^{k-1} and 2^k.

Thus, the following lemma describes the performance of the van Emde Boas layout.

Lemma 1 [45]. *Consider an N-node complete binary search tree T that is stored in a van Emde Boas layout. Then a traversal in T from the root to a leaf uses $O(\log_B N + 1)$ memory transfers.*

Proof. If $N < B$, there are at most 2 memory transfers because the tree T can cross only one block boundary. If $N \geqslant B$, let k be the coarsest level of detail such that every recursive subtree contains at most B nodes. Thus, every recursive subtree is stored in at most 2 memory blocks. Since tree T has height $\lg(N + 1)$, and the height of the subtrees ranges from $(\lg B)/2$ to $\lg B$, the number of subtrees traversed from the root to a leaf is at most $2 \log N / \log B = 2 \log_B N$. Since each subtree can be in at most 2 memory blocks, traversing a path from the root to a leaf uses at most $4 \log_B N$ memory transfers. □

2.3. Dynamic cache-oblivious structure

Our dynamic cache-oblivious locality-preserving dictionary uses the packed-memory structure to store its data, and it uses the static structure as an index into the packed-memory structure. Henceforth, we use the term "array" to refer to the packed memory structure storing the data, and "tree" to refer to the static cache-oblivious structure that serves as an index. We use this terminology even though the "tree" is actually stored as an array.

The N data items are stored in a packed-memory structure, which is an array A of size $\Theta(N)$. Recall that the items appear in the array in sorted order but some of the array positions are kept blank.

Recall that the static cache-oblivious structure consists of a complete tree. Let T_i denote the ith leftmost leaf in the tree. In our structure there are pointers between array position $A[i]$ and leaf T_i, for all values of i. We maintain the invariant that $A[i]$ and T_i store the same key value. All internal nodes of T store the maximum of the nonblank key value(s) of its children. If a node has two children with blank key values, it also has a blank key value.

We now describe the supported operations.

PREDECESSOR(x): Predecessor is carried out by traversing the tree from the root to a leaf. Since each internal node stores the maximum key value of the leaves in its induced subtree, this search is similar to the standard predecessor search on a binary search tree. When the search has reached a node u, it decides whether to branch left or right by comparing the search key to the key of u's left child.

Theorem 2. *The operation* PREDECESSOR(x) *uses* $O(\log_B N + 1)$ *block transfers.*

Proof. Search in our structure is similar to the $O(\log_B N + 1)$ root-to-leaf traversal described in Lemma 1, except that the process of examining the key value of the current node's left child at every step may cause an additional $O(\log_B N + 1)$ block transfers. □

SCANFORWARD(), SCANBACKWARD(): These operations are implemented by scanning forward or backwards to the next non-blank item in the array from the last item accessed. Because of the density constraint on the array, we are guaranteed that we only scan $O(1)$ elements to find a non-blank element.

Theorem 3. *A sequence of* k SCANFORWARD *or* SCANBACKWARD *operations uses* $O(k/B + 1)$ *block transfers.*

Proof. A sequence of k SCANFORWARD or SCANBACKWARD operations only accesses $O(k)$ consecutive elements in an array in order. Thus, scan takes $O(k/B + 1)$ block transfers. □

INSERT(x), DELETE(x): We describe INSERT(x); DELETE(x) proceeds in the same manner. We insert in three stages: First, we perform a predecessor query to find the location in the array to insert x. Then, we insert x into the array using the insertion algorithm of the packed-memory structure. Finally, we update the key values in the tree to reflect the changes in the array.

The first two steps are straightforward; we now describe the third step in more detail. First, we copy the updated keys from the array into the corresponding locations in the tree. We then update all of the ancestors of the updated leaves. We proceed through this subtree according to the *postorder* traversal, that is, the tree traversal where both children are visited before their parent. The updating process changes the key value of a node to reflect the maximum of the key values of its children. By updating using the postorder traversal, we can guarantee that when we reach a given node the values of its children have been updated already.

Lemma 4. *To perform a postorder traversal on k leaves in the tree and their ancestors requires $O(\log_B N + k/B + 1)$ block transfers.*

Proof. We consider the largest level of detail in the tree where recursive subtrees are smaller than B. Consider the horizontal stripes formed by the subtrees at this level of detail. On any root-to-leaf path we pass through $\Theta(\log_B N + 1)$ stripes. We number the stripes from the bottom of the tree starting at 1. Each stripe consists of a forest of subtrees of the original tree. If the root of a tree T_a in stripe i is a child (in the full tree) of a leaf of a tree T_b in stripe $i+1$, we say that T_a is a *tree-child* of T_b.

Accessing all of the items in one tree in any stripe uses at most two memory transfers, since the subtree is stored in a consecutive region of memory of size at most B. We now analyze the cost of accessing one of the stripe-2 trees T and all of its stripe-1 tree-children. The size of all of these trees is in the range $\sqrt{B}$ to B. In the postorder traversal, all of the stripe-1 trees are accessed in the order that they are stored in the array. Since all of the stripe-1 tree-children of T are stored consecutively in memory, the number of page faults caused by accessing ℓ consecutive items in the stripe-1 trees of T in postorder is at most $1 + 2\ell/B$, provided memory can hold 2 blocks. Accessing all of the items in T takes 2 memory transfers provided that memory can hold 2 blocks. Accessing any k consecutive items in T and all the descendant tree-children involves interleaving accesses to items in T and accesses to T's tree-children. Interleaving these operations takes no more block transfers than doing operations separately, provided sufficient cache is available. Thus, at most $2 + 2k/B$ block transfers are performed if memory can hold 4 blocks.

By the above argument, an $\Omega(\log_B N)$-sized cache is enough to support the claimed bounds. However, only a constant-sized cache is actually necessary because most of the nodes belong to subtrees in stripes 1 and 2. At most a $1/B$ fraction of nodes are in all other stripes, so we can afford to pay one memory transfer for each of these accesses and then one additional transfer to bring back into memory a block that may have been prematurely evicted. More specifically, to access k consecutive items in the tree in postorder primarily consists of accessing level-1 and level-2 subtrees. In addition, there are at most $O(\log_B N + k/B)$ items accessed at stripes 3 and higher. Each of these accesses causes $O(1)$ memory transfers. Thus, $O(\log_B N + k/B + 1)$ block transfers are performed to access k consecutive items in the tree in postorder, given a cache of 5 blocks. □

Theorem 5. *The number of block transfers caused by the* INSERT(x) *and* DELETE(x) *operations is $O(\log_B N + \log^2 N/B + 1)$.*

Proof. We describe the proof for INSERT(x); DELETE(x) proceeds in the same manner. The predecessor query costs $O(\log_B N + 1)$. The cost of inserting into the packed memory structure is $O(\log^2 N/B + 1)$ amortized memory transfers. Let w be the actual number of items changed in the array by the packed-memory structure. By Lemma 4, updating the internal nodes uses $O(\log_B N + w/B + 1)$ memory transfers. Since $O(w/B + 1)$ is asymptotically the same as the actual number of block transfers performed by the packed-memory structure's insertion into the array, it is the $O(\log^2 N/B + 1)$ amortized cost of insertion into the packed-memory structure. Therefore the entire INSERT operation uses $O(\log^2 N/B + \log_B N + 1)$ amortized memory transfers. □

2.4. Finger searches

We now show how to perform finger searches on our structure. Brown and Tarjan [23] explain how to use level-linking in order to execute finger searches efficiently. The level-linking method involves using left and right level pointers on every node. We can then execute finger searches by following a combination of the regular tree pointers and the level-linking pointers.

Here for clarity of presentation we present a different approach and avoid the use of these pointers altogether.[2]

Lemma 6. *Let x and y be leaves in the static index structure. Let k be the leaf-distance between x and y, where k is* known *to the algorithm. Let p a pointer to leaf x. The pointer p can be moved to leaf y in $O(\log_B k + 1)$ block transfers.*

Proof. We assume without loss of generality that $y \geqslant x$. The following algorithm is used to search for y:

1. Move p up $\lceil \lg k \rceil$ nodes. Call this node a.
2. Move p down $\lceil \lg k \rceil$ nodes searching for y using the normal tree searching rules.
3. Pointer p now points to a leaf node z. If z is y, stop. Otherwise, move p to z's successor.
4. Move p up $\lceil \lg k \rceil$ nodes. Call this node b.
5. Move p down $\lceil \lg k \rceil$ nodes searching for y using the normal tree searching rules.

Correctness

Since the nodes a and b are at height $\lceil \log k \rceil$, the number of leaves in each of their induced subtrees is at least k. Note that the induced subtree of a contains x, and the induced subtree of b contains at least k leaves that are adjacent to the largest leaf in the induced subtree of a. Thus, y must be in the induced subtree of either a or b. The algorithm must find y because it searches the nodes in the induced subtrees of both a and b, and thus it is correct.

Runtime

In the static structure, any traversal from an internal node a of height h to a leaf in its induced subtree takes time $O(\log_B h + 1)$ memory transfers. This follows from applying the logic of Lemma 7 to the induced subtree of a. The algorithm does at most four such searches. □

Lemma 7. *Let x and y be leaves in the static index structure. Let the leaf-distance between x and y be k, where k is* unknown *to the algorithm. Let p a pointer to leaf x. The pointer p can be moved to leaf y in $O(\log_B k + 1)$ block transfers.*

[2] Note that avoiding additional pointers is not a worthwhile goal in itself because Morin's dictionary diet can be employed to reduce the number of pointers per node on any dictionary [44].

Proof. We guess the value of k using the doubly exponential sequence $2^{2^1}, 2^{2^2}, 2^{2^3}, \ldots$, and we repeat the method in Lemma 6 until y is found. This guessing ends at or before $2^{2^{\lceil \lg \lg k \rceil}}$ since $2^{2^{\lceil \lg \lg k \rceil}} \geqslant k$. Thus, the runtime is $\sum_{i=0}^{\lceil \lg \lg k \rceil} \log_B 2^{2^i} \leqslant 2 \log_B 2^{2^{\lceil \lg \lg k \rceil}} = O(\log_B k)$ block transfers. □

From Lemmas 6 and 7 we obtain the following theorem:

Theorem 8. *Let $d(x, y)$ be the rank distance between x and y. Given a pointer to an item x, the operation* PREDECESSOR(y) *can be executed in $O(\log_B(d(x, y) + 1))$ block transfers.*

3. Simulation results

We now explore how the block size B and the cache size M affect the performance of our cache-oblivious data structure and how the cache-oblivious data structure compares to a standard B-tree. In our simulations we began with an empty structure and inserted many elements. Each data entry is an unsigned 32-bit integer, so the domain space of the data elements is $[0, 2^{32} - 1]$. Whenever the array becomes too full we recopy the elements into a larger array.

We tested our structure using the following input patterns:

1. *Insert at Head*—The elements are inserted at the beginning of the array. This insertion pattern models close to worst-case behavior of the packed-memory array, where the inserts "hammer" on one region of the array.
2. *Random Inserts*—An element is chosen randomly from its domain space for each insertion. We implement the random insert by assigning each new element a random 32-bit key.
3. *Bulk Inserts*—This insertion pattern is a middle ground between random inserts and insert at head. In this strategy we pick a random element to insert and insert a sequence of elements just before it. (We perform the packed-memory-structure modification after each element is inserted.) We run the simulations with bulk sizes of 1, 10, 100, 1000, 10,000, 100,000, and 1,000,000. Observe that random inserts and insert at head are special cases of bulk insert with bulk size 1 and 1,000,000, respectively.

Our experiments have three parts. First, we test the packed-memory structure to measure the amortized number of moved elements or array positions scanned per insertion. We consider different density thresholds as well as different density patterns. We next built a memory simulator, which models which blocks are in memory and which blocks are on disk. We adopt the standard Least Recently Used (LRU) replacement strategy and assume full associativity. Thus, whenever we touch a block that is not in memory, we increment the page-fault count, bring the block into memory, and eject the least-recently-used block. We separately measure the number of page faults caused by packed-memory-structure manipulations and index-structure manipulations. Finally, we compare our structure with a standard B-tree. In the simulations, memory and block sizes are chosen from a range to represent many possible system configurations.

3.1. Scans and moves

We measure both the number of elements moved and the number of array positions scanned, that is, touched during a rebalance. Note that each moved element is moved approximately twice. This is because when we rebalance an interval, first we compress the elements at one end of the interval, and then we space the elements evenly in the interval, moving each element twice. Figure 1 shows our results of moves for the insert-at-head insertion strategy. We consider density parameters of 50%, 60%, 70%, 80%, and 90%. With a 60% density threshold the average number of moves is only 320 when inserting 1,000,000 elements, and only 350 when inserting 2,000,000 elements. Even with a 90% density threshold the number of moves is only 1100 when inserting 2,000,000 elements. Figure 2 shows in the worst case, the number of moves is $\Theta(\log^2(N))$.

Figure 3 shows the number of moves for random inserts. There are only a small constant number of element moves or scanned elements per random insertion. Figure 4 depicts bulk inserts ranging from best case (random) to worst case (insert-at-head); the number of moves increases with the bulk size.

3.2. Page faults

We first focus on the page faults caused by the packed-memory structure using the insert-at-head insertion strategy. As expected, the number of page faults is more sensitive to B than to N. The number of page faults behaves roughly linear in B, as depicted

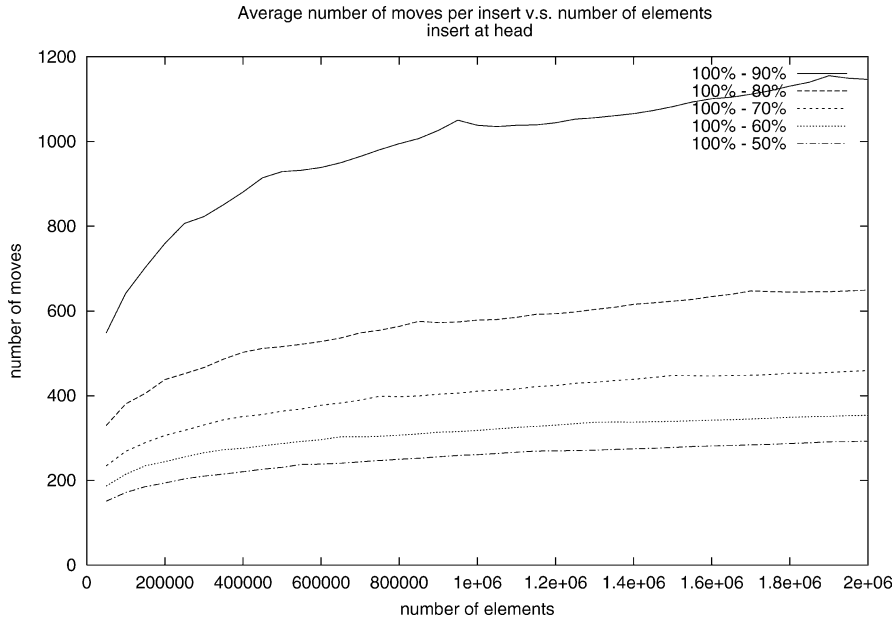

Fig. 1. Average number of moves per insert versus number of elements using different density thresholds with the insert-at-head insertion pattern.

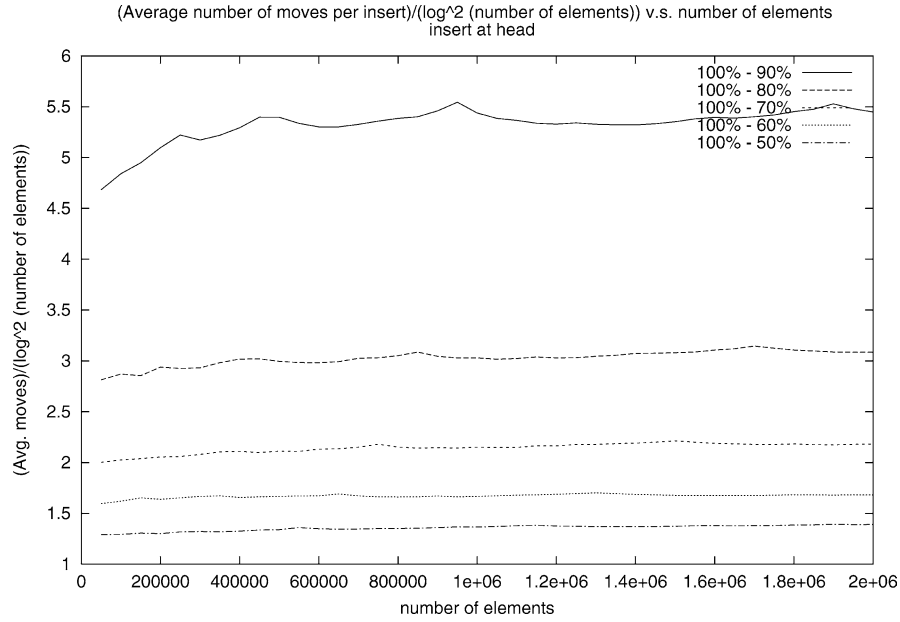

Fig. 2. (Average number of moves per insert)/($\log^2$(number of elements)) versus number of elements using different density thresholds with the insert-at-head insertion pattern. This graph demonstrates that the worst case truly is $\Theta(\log^2 N)$ amortized moves per insertion.

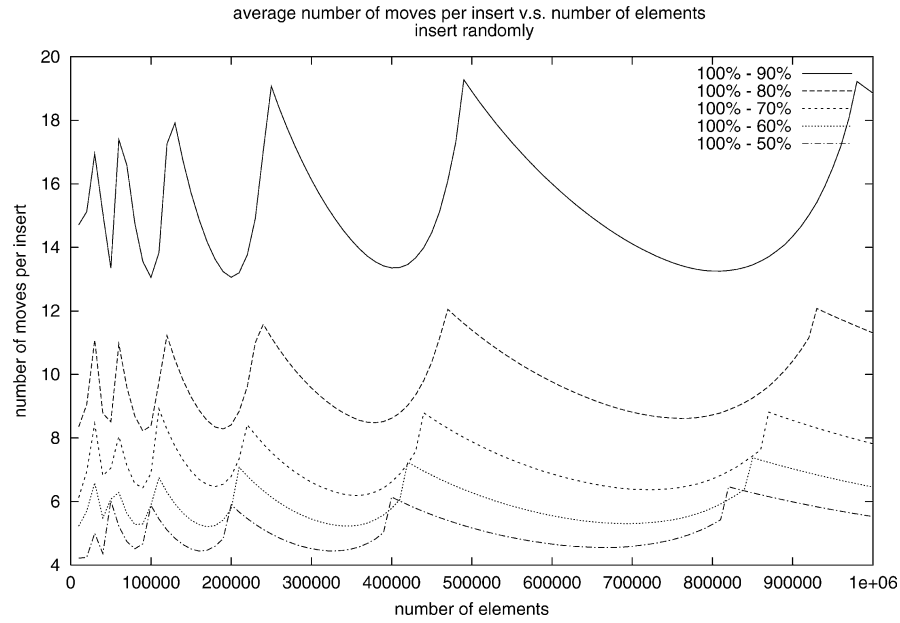

Fig. 3. Average number of moves per insert versus number of elements using different density thresholds with random insertion pattern. The dips occurs when the elements are recopied to a larger array.

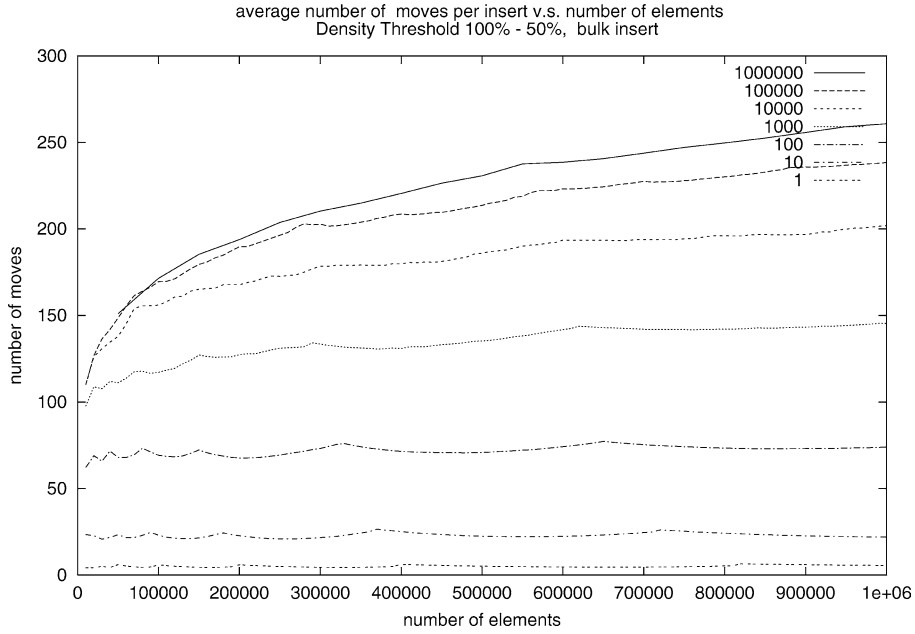

Fig. 4. Average moves per insert versus number of elements using density threshold 100%–50%, bulk insert with bulk sizes $1, 10, 100, \ldots, 1000000$.

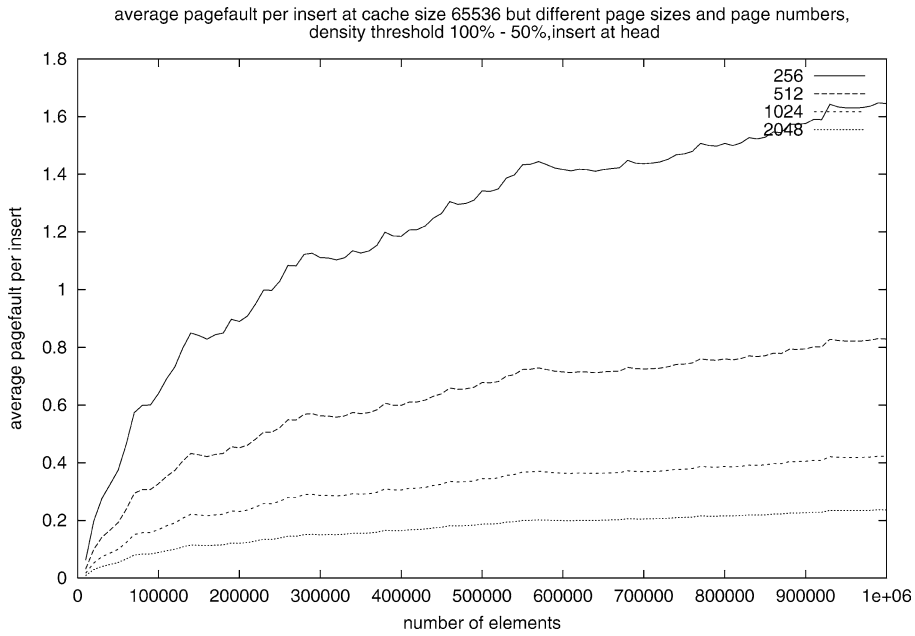

Fig. 5. Average page faults per insert, at total memory size 65536 bytes, but with different page sizes and page numbers.

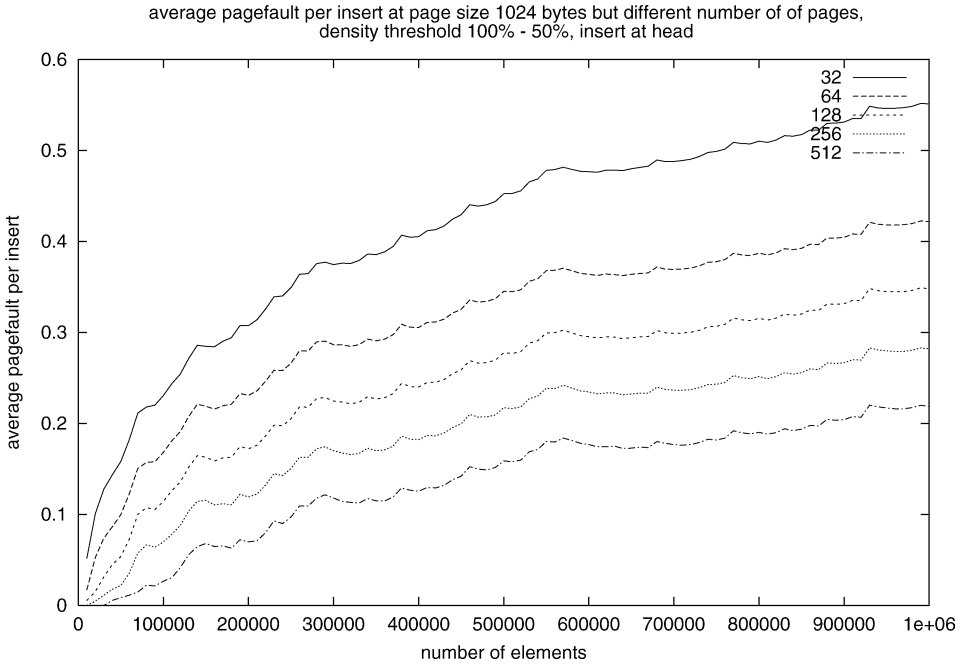

Fig. 6. Average page faults per insert, at page size 1024 but with different numbers of pages, insert at head.

in Figs. 5 and 6. Specifically, Fig. 5 suggests that the number of page faults decreases linearly when B increases linearly and M remains fixed. Figure 6 indicates that the number of page faults decreases roughly linearly when B is fixed and M increases exponentially.

Overall, these results are promising. Even for insertions at the head, the number of page faults per insertion is less than 1 for most reasonable values of B and M. There are few page faults because we are attempting to insert into one portion of the array which can be kept in cache unless a rebalance window is larger then the cache size, a relatively infrequent event.

Although random insertions are close to the best case in terms of number of elements moved, they are close to the worst case in terms of number of page faults. This is because since the location of the insert in the array is chosen randomly, The relevant page is unlikely to be in the cache already. Thus, we expect a page fault per insertion, which is supported by Fig. 7. The situation for bulk insertions as illustrated in Fig. 8 is dramatically better than for random insertions because the cost of the page fault is amortized over the number of elements inserted in a region.

We next measure the number of page faults caused by the entire structure, separately considering the contribution from the searching the index and the scanning the array. We consider bulk sizes of 1, 10, 100, 1000, 10,000, 100,000, and 1,000,000; see Figs. 9 and 10 and Tables 2 and 3. Interestingly, the worst case is for random inserts where there are typically two page faults caused by the index and one caused by the scanning structure. As the bulk size increases to 10 and then 100 we obtain almost order-of-magnitude

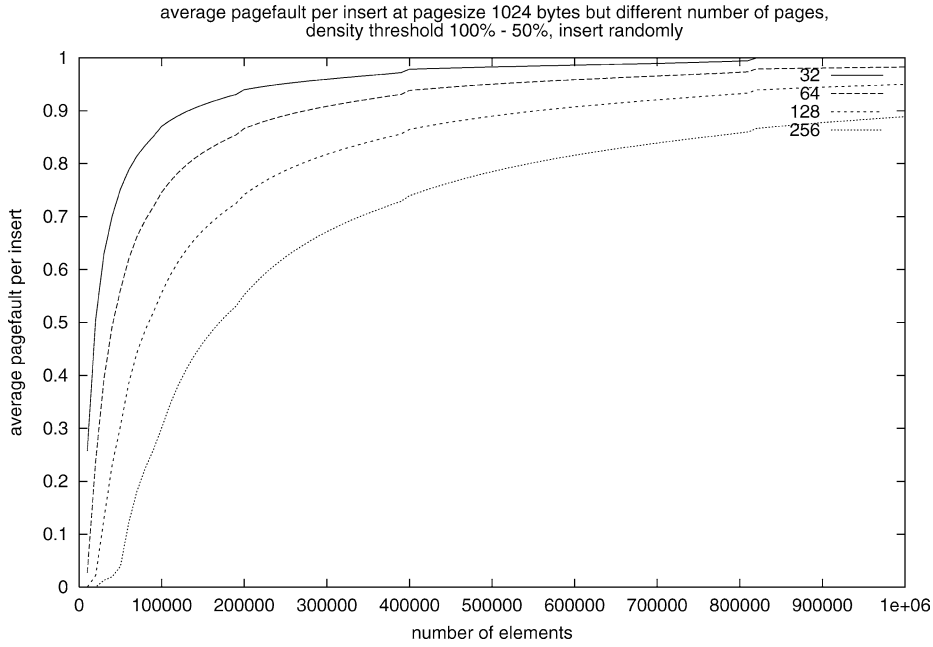

Fig. 7. Average page faults per insert, at page size 1024 but with different numbers of pages, insert randomly.

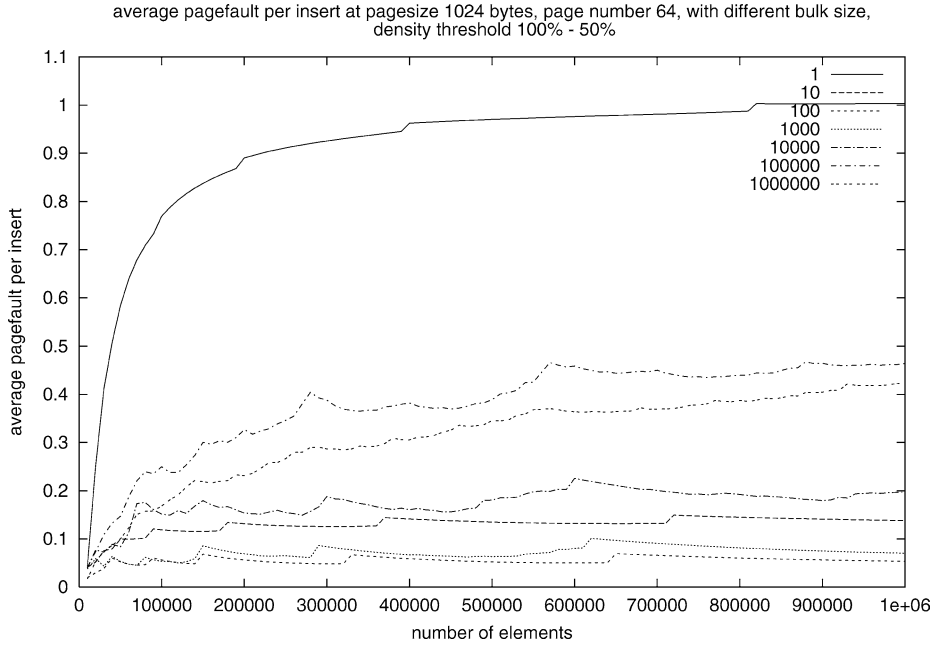

Fig. 8. Average page faults per insert, at page size 1024 and page number 64, but with different bulk sizes $1, 10, 100, \ldots, 1000000$.

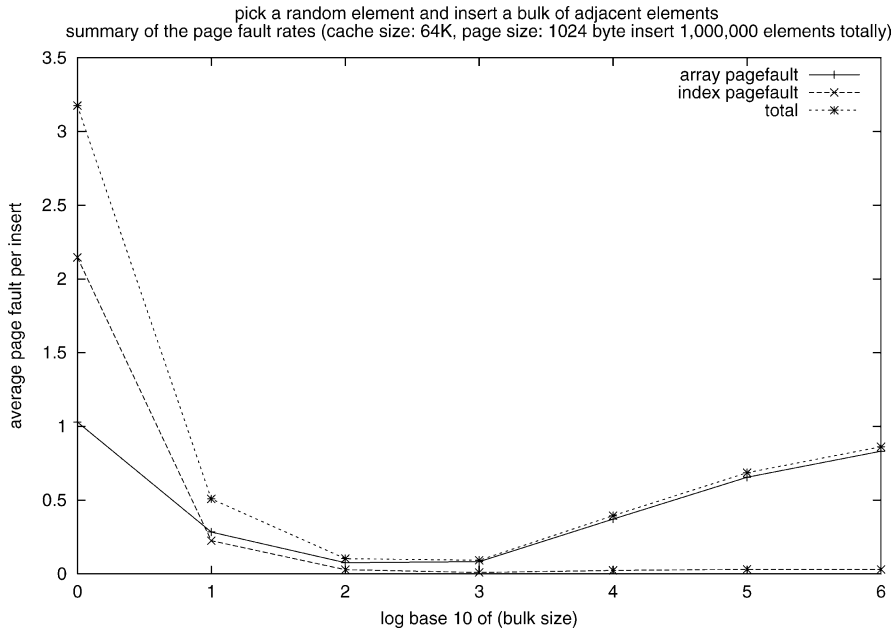

Fig. 9. Page fault rate versus $\log_{10}$(bulk size) (insert 1,000,000 elements, page size 1024 bytes, page number 64), our data structure.

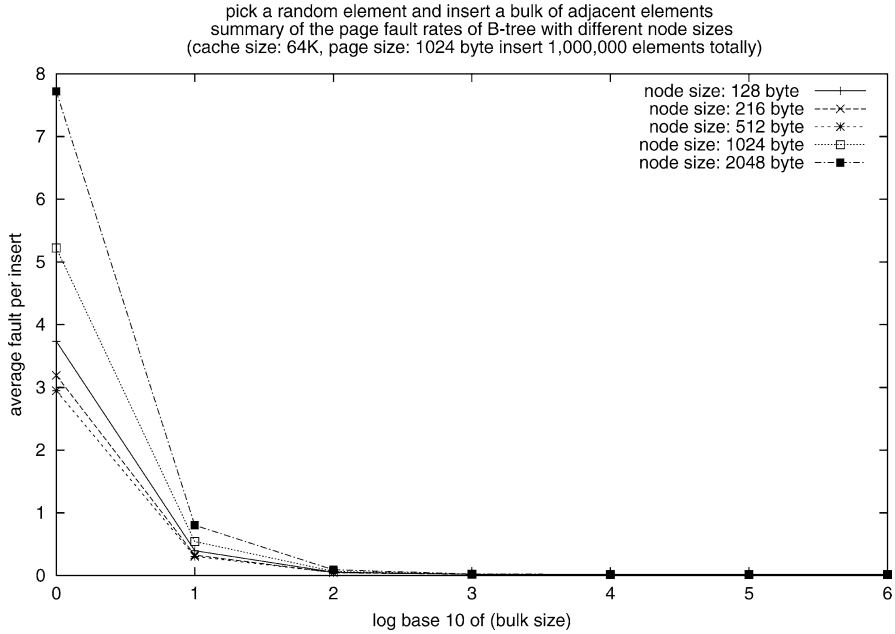

Fig. 10. Page fault rate versus $\log_{10}$(bulk size) (insert 1,000,000 elements, page size 1024 bytes, page number 64), B-tree.

Table 2
Page fault rate of our data structure. We inserted 1,000,000 elements; the page size was 1024 bytes and there were 64 pages

Bulk size	Array pagefault	Index pagefault	Total
10^0	1.0	2.2	3.2
10^1	0.28	0.23	0.51
10^2	0.076	0.028	0.10
10^3	0.083	0.0095	0.093
10^4	0.37	0.024	0.39
10^5	0.65	0.031	0.69
10^6	0.83	0.030	0.86

Table 3
Page fault rate of the B-tree. We inserted 1,000,000 elements; the page size was 1024 bytes and there were 64 pages

Bulk size	Node size				
	128	256	512	1024	2048
10^0	3.7	3.2	2.9	5.2	7.7
10^1	0.40	0.33	0.31	0.54	0.80
10^2	0.053	0.048	0.047	0.067	0.091
10^3	0.020	0.019	0.022	0.020	0.024
10^4	0.016	0.016	0.016	0.016	0.016
10^5	0.016	0.016	0.016	0.016	0.016
10^6	0.016	0.016	0.016	0.016	0.016

improvements in efficiency. For larger bulk sizes the rebalancing in the packed-memory structure hurts the cache performance, increasing the number of page faults. However, the performance is never worse than for random insertions.

We also simulated a B-tree. The data structure is a modification of David Carlson's implementation [24] based on the one found in [38]. The data entries in the B-tree are also 32 bit integers. Each node of the B-tree contains at most B data entries and $B+1$ branches. The page fault rate of insertion into the B-tree is tested with different B and different bulk sizes.

Insert at head is by far the best case for B-trees. Interestingly, this is not even the case B-trees are optimized for because there is little advantage to the B-sized fan-out since an entire root-to-leaf path fits in cache. When we increase B but keep the node size within the size of a page, the performance improves. However, if the node size gets larger than the page size, the performance gets much worse, especially when inserting randomly. We measured the search efficiency when searching for a random key from 1,000,000 indexed elements. When the page size is 1024 bytes, the average page fault per search of the B-tree with node size 816 is 3.77, whereas the average page fault per search for our structure is 3.69.

4. Conclusion

We have developed and simulated a new cache-oblivious locality-preserving dictionary, which supports INSERT, DELETE, and SCAN operations. Our structure has two advantages not held by the standard B-tree. First, it is cache oblivious, that is, it is not parameterized by the block size or any other characteristics of the memory hierarchy. Second, it is locality preserving. That is, unlike any other dynamic dictionary except for [16,22], the structure keeps data stored compactly in memory in order.

Interestingly, although our structure is algorithmically more sophisticated than the B-tree, it may be of comparable difficulty to implement. Unlike the B-tree structure, which requires dynamic memory allocation and pointer manipulation, our structure is just two static arrays.

Different insertion patterns have different costs in our structure and in the standard B-tree. Our simulations indicate that our worst-case performance is at least as good as the worst-case performance of the B-tree for typical block and memory sizes. Indeed, when the B-tree is not optimized for the block size then our structure outperforms the B-tree. This worst-case performance is exhibited during random insertions. On the other hand, because we must keep data in order, we cannot match the B-tree performance when all insertions are to the same location. However, even in the adversarial case, we still perform better than when data is evenly distributed. More research needs to be done to test our data structure on actual input distributions.

For the special case where we know the block size and where the two-level DAM model is an accurate cost model of the system, the B-tree is of course the best option since it is optimized for the DAM model. However, it is becoming increasingly important to optimize for multilevel memories. Moreover, the research effort in clustering B-tree blocks and keeping data in order suggests that even two-level memory hierarchies (i.e., disk and main memory) do not obey the DAM model. More work should be performed on developing more realistic cost models and testing our structure on these models.

If we do not need scans then we can use one level of indirection to perform searches and updates in amortized $O(\log_B N + 1)$ memory transfers (see [16] for details). We can also use our data structure to keep data ordered within superblocks that are arbitrarily placed in memory. Thus, practitioners can benefit from the cache-oblivious index structure and modify the superblocks according to need.

Acknowledgments

The authors gratefully acknowledge Jon Bentley, Erik Demaine, Martín Farach-Colton, Petr Konečný, and Torsten Suel for useful discussions.

References

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

[2] P.K. Agarwal, L. Arge, A. Danner, B. Holland-Minkley, Cache-oblivious data structures for orthogonal range searching, in: Proceedings of 19th Annual ACM Symposium on Computational Geometry, San Diego, CA, 2003, pp. 237–245.
[3] A. Aggarwal, B. Alpern, A.K. Chandra, M. Snir, A model for hierarchical memory, in: Proceedings of the 19th Annual ACM Symposium on Theory of Computing, New York, 1987, pp. 305–314.
[4] A. Aggarwal, A.K. Chandra, Virtual memory algorithms, in: Proceedings of the ACM Symposium on Theory of Computation, 1988, pp. 173–185.
[5] A. Aggarwal, A.K. Chandra, M. Snir, Hierarchical memory with block transfer, in: Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science, Los Angeles, CA, 1987, pp. 204–216.
[6] A. Aggarwal, J.S. Vitter, The input/output complexity of sorting and related problems, Comm. ACM 31 (9) (1988) 1116–1127.
[7] B. Alpern, L. Carter, E. Feig, T. Selker, The uniform memory hierarchy model of computation, Algorithmica 12 (2–3) (1994) 72–109.
[8] S. Alstrup, M.A. Bender, E.D. Demaine, M. Farach-Colton, J.I. Munro, T. Rauhe, M. Thorup, Efficient tree layout in a multilevel memory hierarchy, http://www.arXiv.org/abs/cs.DS/0211010, 2002.
[9] A. Andersson, Faster deterministic sorting and searching in linear space, in: Proceedings of the 37th Annual Symposium on Foundations of Computer Science, 1996, pp. 135–141.
[10] A. Andersson, M. Thorup, Tight(er) worst-case bounds on dynamic searching and priority queues, in: Proceedings of the 31st Annual ACM Symposium on Theory of Computing, 2000, pp. 335–342.
[11] L. Arge, M.A. Bender, E.D. Demaine, B. Holland-Minkley, J.I. Munro, Cache-oblivious priority queue and graph algorithm applications, in: Proceedings of the 34th Annual ACM Symposium on Theory of Computing, Montréal, Canada, 2002, pp. 268–276.
[12] R. Bayer, E.M. McCreight, Organization and maintenance of large ordered indexes, Acta Informatica 1 (3) (1972) 173–189.
[13] M.A. Bender, G.S. Brodal, R. Fagerberg, D. Ge, S. He, H. Hu, J. Iacono, A. López-Ortiz, The cost of cache-oblivious searching, in: Proceedings of the 44th Annual Symposium on Foundations of Computer Science, Cambridge, MA, 2003, pp. 271–282.
[14] M.A. Bender, R. Cole, E.D. Demaine, M. Farach-Colton, Scanning and traversing: maintaining data for traversals in a memory hierarchy, in: Proceedings of the 10th Annual European Symposium on Algorithms, Rome, Italy, in: Lecture Notes in Comput. Sci., vol. 2461, 2002, pp. 139–151.
[15] M.A. Bender, R. Cole, R. Raman, Exponential structures for efficient cache-oblivious algorithms, in: Proceedings of the 29th International Colloquium on Automata, Languages and Programming, Málaga, Spain, in: Lecture Notes in Comput. Sci., vol. 2380, 2002, pp. 195–207.
[16] M.A. Bender, E. Demaine, M. Farach-Colton, Cache-oblivious B-trees, in: Proceedings of the 41st Annual Symposium on Foundations of Computer Science, 2000, pp. 399–409.
[17] M.A. Bender, E.D. Demaine, M. Farach-Colton, Efficient tree layout in a multilevel memory hierarchy, in: Proceedings of the 10th Annual European Symposium on Algorithms, Rome, Italy, in: Lecture Notes in Comput. Sci., vol. 2461, 2002, pp. 165–173.
[18] M.A. Bender, Z. Duan, J. Iacono, J. Wu, A locality-preserving cache-oblivious dynamic dictionary, in: Proceedings of the 13th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA), 2002, pp. 29–38.
[19] G.S. Brodal, R. Fagerberg, Cache oblivious distribution sweeping, in: Proceedings of the 29th International Colloquium on Automata, Languages, and Programming, Malaga, Spain, in: Lecture Notes in Comput. Sci., vol. 2380, 2002, pp. 426–438.
[20] G.S. Brodal, R. Fagerberg, Funnel heap—a cache oblivious priority queue, in: Proceedings of the 13th Annual International Symposium on Algorithms and Computation, Vancouver, Canada, in: Lecture Notes in Comput. Sci., vol. 2518, 2002, pp. 219–228.
[21] G.S. Brodal, R. Fagerberg, On the limits of cache-obliviousness, in: Proceedings of the 35th Annual ACM Symposium on Theory of Computing, San Diego, CA, 2003, pp. 307–315.
[22] G.S. Brodal, R. Fagerberg, R. Jacob, Cache oblivious search trees via binary trees of small height (extended abstract), in: Proceedings of the 13th ACM–SIAM Symposium on Discrete Algorithms, 2002, pp. 39–48.
[23] M.R. Brown, R.E. Tarjan, Design and analysis of a data structure for representing sorted lists, SIAM J. Comput. 9 (1980) 594–614.
[24] D. Carlson, Software design using C++, http://cis.stvincent.edu/carlsond/swdesign/swd.html, 2001.

[25] D. Comer, The ubiquitous B-tree, ACM Comput. Surveys 11 (2) (1979) 121–137.
[26] J.V. den Bercken, B. Seeger, P. Widmayer, A generic approach to bulk loading multidimensional index structures, in: M. Jarke, M.J. Carey, K.R. Dittrich, F.H. Lochovsky, P. Loucopoulos, M.A. Jeusfeld (Eds.), VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25–29, 1997, Athens, Greece, Kaufmann, 1997, pp. 406–415.
[27] G. Franceschini, R. Grossi, Optimal cache-oblivious implicit dictionaries, in: Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP), in: Lecture Notes in Comput. Sci., vol. 2719, 2003, pp. 316–331.
[28] G. Franceschini, R. Grossi, Optimal worst-case operations for implicit cache-oblivious search trees, in: Proceedings of the 8th Workshop on Algorithms and Data Structures (WADS), 2003, pp. 114–126, in press.
[29] M. Frigo, C.E. Leiserson, H. Prokop, S. Ramachandran, Cache-oblivious algorithms, in: Proceedings of the 40th Annual Symposium on Foundations of Computer Science, New York, 1999, pp. 285–297.
[30] J. Gray, G. Graefe, The five minute rule ten years later, SIGMOD Record 26 (4) (1997).
[31] L.J. Guibas, R. Sedgewick, A dichromatic framework for balanced trees, in: Proceedings of the 19th Annual Symposium on Foundations of Computer Science, Ann Arbor, MI, 1978, pp. 8–21.
[32] S. Huddleston, K. Mehlhorn, A new data structure for representing sorted lists, Acta Informatica 17 (1982) 157–184.
[33] J. Iacono, Alternatives to splay trees with $o(\log n)$ worst-case access times, in: Proceedings of the 11th Symposium on Discrete Algorithms, 2001, pp. 516–522.
[34] A. Itai, A.G. Konheim, M. Rodeh, A sparse table implementation of priority queues, in: S. Even, O. Kariv (Eds.), Proceedings of the 8th Colloquium on Automata, Languages, and Programming, Acre (Akko), Israel, in: Lecture Notes in Comput. Sci., vol. 115, 1981, pp. 417–431.
[35] I. Kamel, C. Faloutsos, On packing R-trees, in: Proc. International Conference on Information and Knowledge Management, 1993, pp. 490–499.
[36] K. Kim, S.K. Cha, Sibling clustering of tree-based spatial indexes for efficient spatial query processing, in: Proc. ACM Internat. Conf. Information and Knowledge Management, 1998, pp. 398–405.
[37] D.E. Knuth, Sorting and Searching, The Art of Computer Programming, vol. 3, second ed., Addison–Wesley, Reading MA, 1998.
[38] R.L. Kruse, A.J. Ryba, Data Structures and Program Design in C++, Prentice-Hall, Upper Saddle River, NJ, 1998.
[39] P. Kumar, E. Ramos, I/O-efficient construction of Voronoi diagrams, Manuscript, 2003.
[40] R.E. Ladner, R. Fortna, B.-H. 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, in: Lecture Notes in Comput. Sci., vol. 2547, 2002, pp. 78–92.
[41] L. Arge, K. Hinrichs, J. Vahrenhold, J. Vitter, Efficient bulk operations on dynamic r-trees, in: ALENEX, 1999, pp. 328–348.
[42] R. Meville, D. Gries, Controlled density sorting, Inform. Process. Lett. 10 (1980) 169–172.
[43] J. Nievergelt, E.M. Reingold, Binary search trees of bounded balance, SIAM J. Comput. 2 (1973) 33–43.
[44] P. Morin, Putting your dictionary on a diet, Technical Report TR-02-07, Carleton University School of Computer Science, November 2002.
[45] H. Prokop, Cache-oblivious algorithms, Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, June 1999.
[46] W. Pugh, Skip lists: a probabilistic alternative to balanced trees, in: F. Dehne, J.-R. Sack, N. Santoro (Eds.), Proceedings of the Workshop on Algorithms and Data Structures, Ottawa, ON, Canada, in: Lecture Notes in Comput. Sci., vol. 382, 1989, pp. 437–449.
[47] N. Rahman, R. Cole, R. Raman, Optimized predecessor data structures for internal memory, in: Proceedings of the 5th Workshop on Algorithms Engineering, Aarhus, Denmark, 2001, pp. 67–78.
[48] V. Raman, Locality preserving dictionaries: theory and application to clustering in databases, in: ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems, 1999.
[49] J.E. Savage, Extending the Hong-Kung model to memory hierarchies, in: Proceedings of the 1st Annual International Conference on Computing and Combinatorics, in: Lecture Notes in Comput. Sci., vol. 959, 1995, pp. 270–281.
[50] R. Seidel, C.R. Aragon, Randomized search trees, Algorithmica 16 (4–5) (1996) 464–497.
[51] D.D. Sleator, R.E. Tarjan, Self-adjusting binary search trees, J. ACM 32 (3) (1985) 652–686.

[52] M. Thorup, Faster deterministic sorting and priority queues in linear space, in: Proc. of the 9th Annual ACM–SIAM Symposium on Discrete Algorithms, 1998, pp. 550–555.
[53] 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, Berkeley, CA, 1975, pp. 75–84.
[54] P. van Emde Boas, R. Kaas, E. Zijlstra, Design and implementation of an efficient priority queue, Math. Systems Theory 10 (2) (1977) 99–127.
[55] J.S. Vitter, External memory algorithms and data structures, in: J. Abello, J.S. Vitter (Eds.), External Memory Algorithms and Visualization, in: DIMACS Ser. Discrete Math. Theoret. Comput. Sci., Amer. Math. Soc., 1999, pp. 1–38.
[56] J.S. Vitter, E.A.M. Shriver, Algorithms for parallel memory, II: Hierarchical multilevel memories, Algorithmica 12 (2–3) (1994) 148–169.
[57] D.E. Willard, Inserting and deleting records in blocked sequential files, Technical Report TM81-45193-5, Bell Laboratories, 1981.
[58] D.E. Willard, Maintaining dense sequential files in a dynamic environment, in: Proceedings of the 14th Annual ACM Symposium on Theory of Computing, San Francisco, CA, 1982, pp. 114–121.
[59] D.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, Washington, DC, 1986, pp. 251–260.
[60] D.E. Willard, A density control algorithm for doing insertions and deletions in a sequentially ordered file in good worst-case time, Inform. and Comput. 97 (2) (1992) 150–204.