An Adaptive Packed-Memory Array

Michael A. Bender and Haodong Hu
Department of Computer Science
Stony Brook, NY 11794, USA
[bender,huhd]@cs.sunysb.edu

ABSTRACT

The *packed-memory array* (*PMA*) is a data structure that maintains a dynamic set of N elements in sorted order in a $\Theta(N)$-sized array. The idea is to intersperse $\Theta(N)$ empty spaces or *gaps* among the elements so that only a small number of elements need to be shifted around on an insert or delete. Because the elements are stored physically in sorted order in memory or on disk, the PMA can be used to support extremely efficient range queries. Specifically, the cost to scan L consecutive elements is $O(1+L/B)$ memory transfers.

This paper gives the first *adaptive packed-memory array* (*APMA*), which automatically adjusts to the input pattern. Like the original PMA, any pattern of updates costs only $O(\log^2 N)$ amortized element moves and $O(1+(\log^2 N)/B)$ amortized memory transfers per update. However, the APMA performs even better on many common input distributions achieving only $O(\log N)$ amortized element moves and $O(1+(\log N)/B)$ amortized memory transfers. The paper analyzes *sequential* inserts, where the insertions are to the front of the APMA, *hammer* inserts, where the insertions "hammer" on one part of the APMA, *random* inserts, where the insertions are after random elements in the APMA, and *bulk* inserts, where for constant $\alpha \in [0,1]$, N^α elements are inserted after random elements in the APMA. The paper then gives simulation results that are consistent with the asymptotic bounds. For sequential insertions of roughly 10^6 elements, the APMA has five times fewer element moves per insertion than the traditional PMA and running times that are more than five times faster.

Categories and Subject Descriptors: D.1.0 [**Programming Techniques**]: General; E.1 [**Data Structures**]: Arrays; E.1 [**Data Structures**]: Lists, stacks, and queues; E.5 [**Files**]: Sorting/searching; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval.

General Terms: Algorithms, Experimentation, Performance, Theory.

Keywords: Adaptive Packed-Memory Array, Cache Oblivious, Locality Preserving, Packed-Memory Array, Range Query, Rebalance, Sequential File Maintenance, Sequential Scan, Sparse Array.

1. INTRODUCTION

A classical problem in data structures and databases is how to maintain a dynamic set of N elements *in sorted order* in a $\Theta(N)$-sized array. The idea is to intersperse $\Theta(N)$ empty spaces or *gaps* among the elements so that only a small number of elements need to be shifted around on an insert or delete. These data structures effectively simulate a library bookshelf, where gaps on the shelves mean that books are easily added and removed.

Remarkably, such data structures can be efficient for any pattern of inserts/deletes. Indeed, it has been known for over two decades that the number of element moves per update is only $O(\log^2 N)$ both amortized [14] and in the worst case [18–20]. Since these data structures were proposed, this problem has been studied under different names, including ***sparse arrays*** [14, 15], ***sequential file maintenance*** [18–20], and ***list labeling*** [10–13]. The problem is also closely related to the ***order-maintenance*** problem [1, 10, 12, 17].

Recently there has been renewed interest in these sparse-array data structures because of their application in I/O-efficient and cache-oblivious algorithms. The I/O-efficient and cache oblivious version of the sparse array is called the ***packed memory array (PMA)*** [2, 3]. The PMA maintains N elements in sorted order in a $\Theta(N)$-sized array. It supports the operations insert, delete, and scan. Let B be the number of elements that fit within a memory block. To insert an element y after a given element x or to delete x costs $O(\log^2 N)$ amortized element moves and $O(1+(\log^2 N)/B)$ amortized memory transfers. The PMA maintains the ***density invariant*** that in any region of size S (for S greater than some small constant value), there are $\Theta(S)$ elements stored in it. To scan L elements after a given element x costs $\Theta(1+L/B)$ memory transfers.

The PMA has been used in cache-oblivious B-trees [2–6, 9], concurrent cache-oblivious B-trees [8], cache-oblivious string B-tree [6], and scanning structures [1]. A sparse array in the same spirit as the PMA was independently proposed and used in the locality-preserving B-tree of [16], although the asymptotic space bounds are superlinear and therefore inferior to the linear space bounds of the earlier sparse-array data structures [14, 18–20] and the PMA [2, 3].

The primary use of the PMA in the literature has been for sequential storage in memory/disk of all the elements of a (cache-oblivious or traditional) B-tree. An early paper suggesting this idea was [16]. The PMA maintains locality of reference at all granularities and consequently supports extremely efficient sequential scans/range queries of the elements. The concern with traditional B-trees is that the 2K or 4K sizes of disk blocks are too small to amortize the cost of disk seeks. Consequently, on modern disks, random block accesses are well over an order-of-magnitude slower than sequential block accesses. Thus, locality-preserving B-trees and cache-oblivious B-trees based on PMAs support range queries that run an order of magnitude faster than those of traditional B-trees [6]. Moreover, since the elements are maintained strictly in sorted order, these structures do not suffer from aging unlike most file systems and databases.

The PMA is an efficient and promising data structure, but

This research was supported in part by NSF Grants ACI-0324974 and CCR-0208670.

PODS'06, June 26–28, 2006, Chicago, Illinois, USA.

it also has weaknesses. The main weakness is that the PMA performs relatively poorly on some common insertion patterns such as sequential inserts. For sequential inserts, the PMA performs near its worst in terms of the number of elements moved per insert. The PMA's difficulty with sequential inserts is that the insertions "hammer" on one part of the array, causing many elements to be shifted around. Although $O(\log^2 N)$ amortized elements moves and $O(1+(\log^2 N)/B)$ amortized memory transfers is surprisingly good considering the stringent requirements on the data order, it is relatively slow compared with traditional B-tree inserts. Moreover, sequential inserts are common, and B-trees in databases are frequently optimized for this insertion pattern. It would be better if the PMA could perform near its best, not worst, in this case.

In contrast, one of the PMA's strengths is its performance on common insertion patterns such as random inserts. For random inserts, the PMA performs extremely well with only $O(\log N)$ element moves per insert and only $O(1+(\log N)/B)$ memory transfers. This performance surpasses the guarantees for arbitrary inserts.

Results. This paper proposes an ***adaptive packed-memory array*** (abbreviated ***adaptive PMA*** or ***APMA***), which overcomes these deficiencies of the ***traditional PMA***. Our structure is the first PMA that adapts to insertion patterns, and it gives the largest decrease in the cost of sparse arrays/sequential-file maintenance in almost two decades. The APMA retains the same amortized guarantees as the traditional PMA, but adapts to common insertion patterns, such as sequential inserts, random inserts, and bulk inserts, where chunks of elements are inserted at random locations in the array.

We give the following results for the APMA:

- We first show that the APMA has the "rebalance property", which ensures that any pattern of insertions cost only $O(1+(\log^2 N)/B)$ amortized memory transfers and $O(\log^2 N)$ amortized element moves. Because the elements are kept in sorted order in the APMA, as with the PMA, scans of L elements costs $O(1+L/B)$ memory transfers. Thus, the adaptive PMA guarantees performance at least as good as that of the traditional PMA.

We next analyze the performance of the APMA under some common insertion patterns.

- We show that for ***sequential inserts***, where all the inserts are to the front of the array, the APMA makes only $O(\log N)$ amortized element moves and $O((\log N/B)+1)$ amortized memory transfers.
- We generalize this analysis to ***hammer inserts***, where the inserts hammer on any single element in the array.
- We then turn to ***random inserts***, where each insert occurs after a randomly chosen element in the array. We establish that the insertion cost is again only $O(\log N)$ amortized element moves and $O((\log N/B)+1)$ amortized memory transfers.
- We generalize all these previous results by analyzing the case of ***bulk inserts***. In the bulk-insert insertion pattern, we pick a random element in the array and perform N^α inserts after that element for $\alpha \in [0,1]$. We show that for all values of $\alpha \in [0,1]$, the APMA also only performs $O(\log N)$ amortized element moves and $O(1+(\log N)/B)$ amortized memory transfers.
- We next perform simulations and experiments, measuring the performance of the APMA on these insertion patterns. For sequential insertions of roughly 1.4 million elements, the APMA has over four times fewer element moves per insertion than the traditional PMA and running times that are nearly seven times faster. For bulk insertions of 1.4 million elements, where $f(N) = N^{0.6}$, the APMA has over three times fewer element moves per insertion than the traditional PMA and running times that are over four times faster.

2. ADAPTIVE PMA

In this section we introduce the adaptive PMA. We first explain how the adaptive PMA differs from the traditional PMA. We then show that both PMAs have the same amortized bounds, $O(\log^2 N)$ element moves and $O(1+(\log^2 N)/B)$ memory transfers per insert/delete. Thus, adaptivity comes at no extra asymptotic cost.

Description of Traditional and Adaptive PMAs. We first describe how to insert into both the adaptive and traditional PMAs. Henceforth, ***PMA*** with no preceding adjective refers to either structure. When we insert an element y after an existing element x in the PMA, we look for a neighborhood around element x that has sufficiently low ***density***, that is, we look for a subarray that is not storing too many or too few elements. Once we find a neighborhood of the appropriate density, we ***rebalance*** the neighborhood by spacing out the elements, including y. In the traditional PMA, we rebalance by spacing out the elements evenly. In the adaptive PMA, we may rebalance the elements *unevenly*, based on previous insertions, that is, we leave extra gaps near elements that have recently had inserts after them.

We now give some terminology. We divide the PMA into $\Theta(N/\log N)$ ***segments***, each of size $\Theta(\log N)$, and we let the number of segments be a power of 2. We call a contiguous group of segments a ***window***. We view the PMA in terms of a tree structure, where the nodes of the tree are windows. The root node is the window containing all segments, and a leaf node is a window containing a single segment. A node in the tree that is a window of 2^i segments has two children, a left child that is the window of the first 2^{i-1} segments and a right child that is the window of the last 2^{i-1} segments.

We let the height of the tree be h, so that $2^h = \Theta(N/\log N)$ and $h = \lg N - \lg\lg N + O(1)$. The nodes at each height ℓ have an ***upper density threshold*** τ_ℓ and a ***lower density threshold*** ρ_ℓ, which together determine the acceptable density of keys within a window of 2^ℓ segments. As the node height *increases*, the upper density thresholds *decrease* and the lower density thresholds *increase*. Thus, for constant minimum and maximum densities $D_{\min}$ and $D_{\max}$, we have

$$D_{\min} = \rho_0 < \cdots < \rho_h < \tau_h < \cdots < \tau_0 = D_{\max}. \qquad (1)$$

The density thresholds on windows of intermediate powers of 2 are arithmetically distributed. For example, the maximum density threshold of a segment can be set to 1.0, the maximum density threshold of the entire array to 0.5, the minimum density threshold of the entire array to 0.2, and the minimum density of a segment to 0.1. If the PMA has 32 segments, then the maximum density threshold of a single segment is 1.0, of two segments is 0.9, of four segments is 0.8, of eight segments is 0.7, of 16 segments is 0.6, and of all 32 segments is 0.5.

More formally, upper and lower density thresholds for nodes and height ℓ are defined as follows:

$$\tau_\ell = \tau_h + (\tau_0 - \tau_h)(h-\ell)/h \quad (2)$$

$$\rho_\ell = \rho_h - (\rho_h - \rho_0)(h-\ell)/h. \quad (3)$$

Moreover,

$$2\rho_h < \tau_h, \quad (4)$$

because when we double the size of an array that becomes too dense, the new array must be within the density threshold.[1]

We now give more details about how to insert element y after an existing element x. If there is enough space in the leaf (segment) containing x, then we rearrange the elements within the leaf to make room for y. If the leaf is full, then we find the closest ancestor of the leaf whose density is within the permitted thresholds and rebalance. To delete an element x, we remove x from its segment. If the segment falls below its density threshold, then, as before, we find the smallest enclosing window whose density is within threshold and rebalance. If the *entire* array is above the maximum density threshold (resp., below the minimum density threshold), then we recopy the keys into a PMA of twice (resp., half) the size.

We introduce further notation. Let ***Cap***(u_ℓ) denote the number of array positions in node u_ℓ of height ℓ. Since there are 2^ℓ segments in the node, the capacity is $\Theta(2^\ell \log N)$. Let ***Gaps***(u_ℓ) denote the number of gaps, i.e., unfilled array positions in node u_ℓ. Let ***Density***(u_ℓ) denote the fraction of elements actually stored in node u_ℓ, i.e., $\text{Density}(u_\ell) = 1 - \text{Gaps}(u_\ell)/\text{Cap}(u_\ell)$.

Rebalance. We ***rebalance*** a node u_ℓ of height ℓ if u_ℓ is within threshold, but we detect that a child node $u_{\ell-1}$ is outside of threshold. Any node whose elements are rearranged in the process of a rebalance is ***swept.*** Thus, we ***sweep*** a node u_ℓ of height ℓ when we detect that a child node $u_{\ell-1}$ is outside of threshold, but now u_ℓ need not be within threshold. Note that with this rebalance scheme, this tree can be implicitly rather than explicitly maintained. In this case, a rebalance consists of two scans, one to the left and one to the right of the insertion point until we find a region of the appropriate density.

In a traditional PMA we rebalance evenly, whereas in the adaptive PMA we rebalance unevenly. The idea of the APMA is to store a smaller number of elements in the leaves in which there have been many recent inserts. However, since we must maintain the bound of $O(\log^2 N)$ amortized element moves, we cannot let the density of any child node be too high or too low.

PROPERTY 1. *(rebalance property) After a rebalance, if each node u_ℓ (except the root of the rebalancing subtree) has density within u_ℓ's parent's thresholds, then we say that the rebalance satisfies the* ***rebalance property****. We say that a node u_ℓ is* ***within balance*** *or* ***well balanced*** *if u_ℓ is within its parent's thresholds.*

[1]There are straightforward ways to generalize (4) to further reduce space usage. Introducing this generalization here leads to unnecessary complication in presentation.

The following theorem shows if each rebalance satisfies the rebalance property, then we achieve good update bounds. The proof is essentially that in [2, 3], but the rebalance property applies to a wide set of rebalancing schemes.

THEOREM 1. *If the rebalance in a PMA satisfies the rebalance property, then updates take $O(\log^2 N)$ amortized element moves and $O(1+(\log^2 N)/B)$ amortized memory transfers.*

PROOF. Let u_ℓ be a node at level ℓ. A rebalance of u_ℓ is trigged by an insert or delete that pushes one descendant node u_i at each height $i = 0, \ldots, \ell-1$ above its upper threshold τ_i or below its lower threshold ρ_i. (If this were not the case, then we would rebalance a node of a lower height than ℓ.)

Consider one particular such node u_i. Before the sweep of u_i's parent u_{i+1},

$$\text{Density}(u_i) > \tau_i \quad \text{or} \quad \text{Density}(u_i) < \rho_i\,.$$

After the sweep of u_{i+1}, by the rebalance property,

$$\rho_{i+1} \le \text{Density}(u_i) \le \tau_{i+1}\,.$$

Therefore we need at least

$$(\tau_i - \tau_{i+1})\text{Cap}(u_i)$$

inserts or at least

$$(\rho_{i+1} - \rho_i)\text{Cap}(u_i)$$

deletes before the next sweep of node u_{i+1}. Therefore the amortized size of a sweep of node u_{i+1} per insert into child node u_i is at most

$$\begin{aligned} & \max\left\{\frac{\text{Cap}(u_{i+1})}{(\tau_i - \tau_{i+1})\text{Cap}(u_i)}, \frac{\text{Cap}(u_{i+1})}{(\rho_{i+1} - \rho_i)\text{Cap}(u_i)}\right\} \\ = \; & \max\left\{\frac{2}{\tau_i - \tau_{i+1}}, \frac{2}{\rho_{i+1} - \rho_i}\right\} \\ = \; & O(\log N). \end{aligned}$$

When we insert an element into the PMA, we actually insert into $h = \Theta(\log N)$ such nodes u_i, one at each level in the tree. Therefore the total amortized size of a rebalance per insertion into the PMA is $O(\log^2 N)$. Thus, the amortized number of element moves per insert is $O(\log^2 N)$. Because a rebalance is composed of a constant number of sequential scans, the amortized number of memory transfers per insert is $O(1+(\log^2 N)/B)$, as promised.

Prediction. In a ***predictor*** data structure, we store a small collection of elements that directly precede the newly inserted elements. For each of these ***marker*** elements, we count the number of recently inserted elements that directly follow the marker. We store the current leaf node where each marker element resides.

We give some terminology for prediction: For an element x, let ***insert number*** $I(x)$ denote a count from 0 to $\log N$ estimating the number of inserts after x in the last $O(\log^2 N)$ inserts:

$$I(x) = \min\{\log N, \text{ estimate of the number of inserts after } x \text{ roughly from the last } O(\log^2 N) \text{ inserts }\}.$$

Furthermore, if x is not in the predictor, then we set $I(x) = 0$. For a node u_ℓ at level ℓ, let insert number $I(u_\ell)$ be the sum of the insert numbers of elements in u_ℓ. When rebalancing a node, we reallocate elements unevenly among its descendant leafs according to their insert numbers. The larger the insert number, the fewer elements are allocated.

We implement the predictor as a circular linked list (array) containing $\beta \log N$ cells, for constant β. Two pointers, a head pointer and a tail pointer, indicate the front and the back of the linked list. Each cell in the circular linked list contains three pieces of data: an element x, a pointer to the leaf node in the PMA where the element x currently resides, and a counter recording the number of elements recently inserted after element x (see Figure 1). The predictor is essentially a priority queue where elements are removed from the priority queue in FIFO order. The head cell keep the newest element and the tail cell keep the oldest element in the predictor.

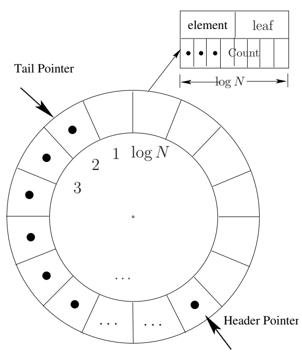

Figure 1: Predictor: each cell contains an element x, a leaf node, and a count number.

When a new element is inserted after an element x, we first check whether x already exists in the predictor. If so, we increase x's count number by 1; then we move x's cell one position forward in the predictor and move the displaced cell back one position to fill the space vacated by x's cell. If the element x does not exist in the predictor, then we create a new cell for x, which is pointed to by the head pointer. There are two special cases. One case is when the element x exists in the predictor and its count number is already at the maximum $\log N$. The other case is when element x is not in the predictor and there is no free cell space for the element x. In both cases, we remove the element with the lowest priority (pointed to by the tail pointer) instead of increasing x's count number or creating a new cell. As long as the count number of the tail cell drops to zero, a new free cell space is available for new inserts. Thus, with this structure, random noise in inserts does not hurt the prediction.

Uneven Rebalance. Now we present the algorithm for uneven rebalance. Assume that nodes $u_{\ell-1}$ and $v_{\ell-1}$ are left and right children of u_ℓ at level ℓ and that there are m ordered elements $\{x_1, x_2, \ldots, x_m\}$ stored in u_ℓ. The uneven rebalance performs as follows:

- If $I(x_i) = 0$ for all $i \in [1, m]$, then we perform an even rebalance for this node u_ℓ.
- Otherwise, we perform an uneven rebalance. Our uneven rebalance is designed so that, the bigger the insert numbers, the more gaps we leave. Specifically, we minimize the quantity

$$\left| \frac{I(u_{\ell-1})}{\mathrm{Gaps}(u_{\ell-1})} - \frac{I(v_{\ell-1})}{\mathrm{Gaps}(v_{\ell-1})} \right|, \qquad (5)$$

subject to the constraint that the rebalance property must be satisfied. When we rebalance, we ***split at an element*** x_i, meaning that we put elements $\{x_1, \ldots, x_i\}$ in $u_{\ell-1}$ and $\{x_{i+1}, \ldots, x_m\}$ in $v_{\ell-1}$. The objective is to find the index i to minimize

$$\left| \frac{\sum_{j=1}^{i} I(x_j)}{\mathrm{Cap}(u_{\ell-1}) - i} - \frac{\sum_{j=i+1}^{m} I(x_j)}{\mathrm{Cap}(v_{\ell-1}) - (m-i)} \right|, \qquad (6)$$

subject to the constraints that

$$i \in \left[\mathrm{Cap}(u_{\ell-1})\rho_\ell, \mathrm{Cap}(u_{\ell-1})\tau_\ell\right] \qquad (7)$$

// density of left child within parents' threshold

$$i \in \left[m - \mathrm{Cap}(v_{\ell-1})\tau_\ell, m - \mathrm{Cap}(v_{\ell-1})\rho_\ell\right]. \qquad (8)$$

// density of right child within parents' threshold

- We recursively allocate elements in $u_{\ell-1}$ and $v_{\ell-1}$'s child nodes and proceed down the tree until we reach the leaves. Once we know the number of elements in each leaf, we rebalance u_ℓ in one scan.

For example, in the insert-at-head case, the insert numbers of right descendants are always 0. Thus, minimizing the simplified objective quantity $|I(u_{\ell-1})/\mathrm{Gaps}(u_{\ell-1})|$ means maximizing $\mathrm{Gaps}(u_{\ell-1})$.

Now we show how to implement the rebalance so that there is no asymptotic overhead in the bookkeeping for the rebalance. Specifically, the number of element moves in the uneven rebalance is dominated by the size of the rebalancing node, as described is the following theorem:

THEOREM 2. *To rebalance a node u_ℓ at level ℓ unevenly requires $O(Cap(u_\ell))$ operations and $O(1 + Cap(u_\ell)/B)$ memory transfers.*

PROOF. There are three steps to rebalancing a node u_ℓ unevenly. First, we check the predictor to obtain the insert numbers of the elements located in all descendant nodes of u_ℓ. Because the size of the predictor is $O(\log N)$, this step takes $O(\log N)$ operations and $O(1 + (\log N)/B)$ memory transfers. Second, we recursively determine the number of elements to be stored in u_ℓ's children, grandchildren, etc., down to descendent leaves. Naively, this procedure uses $O(\ell \mathrm{Cap}(u_\ell))$ operations and $O(1 + \ell \mathrm{Cap}(u_\ell)/B)$ memory transfers; below we show how to perform this procedure in $O(\mathrm{Cap}(u_\ell))$ operations and $O(1 + \mathrm{Cap}(u_\ell)/B)$ memory transfers. Third, we scan the node u_ℓ putting each element into the correct leaf node. Thus, this last step also takes $O(\mathrm{Cap}(u_\ell))$ operations and $O(1 + \mathrm{Cap}(u_\ell)/B)$ memory transfers.

We now show how to implement the second step efficiently. We call the elements in the predictor ***weighted*** elements and the remaining elements ***unweighted***. Recall that only weighted elements have nonzero insert numbers. In the first step, we obtain all information about which elements are weighted. Then, we start the second step, which is recursive. At the first recursive level, we determine which elements are allocated to the left and right children of u_ℓ, i.e., we find the index i minimizing (6). At first glance, it seems necessary to check all indices i in order to get the minimum, which takes $O(\text{Cap}(u_\ell))$ operations, but we can do better. Observe that when the index i is in a sequence of unweighted elements between two weighted elements, the numerator in (6) does not change. Only the denominator changes, and it does so continuously. So in order to minimize (6) at the first recursive level, it is not necessary to check all elements in node u_ℓ. It is enough to check which two contiguous weighted elements the index i is between such that (6) is minimized. Since there are at most $O(\log N)$ weighted elements, the number of operations at each recursive level is at most $O(\log N)$. Furthermore, because there are ℓ recursive levels, the number of operations in the whole recursive step is at most $O(\ell \log N)$, which is less than $O(\text{Cap}(u_\ell))$. By storing these weighted elements contiguously, we obtain $O(1+\text{Cap}(u_\ell)/B)$ memory transfers.

3. SEQUENTIAL AND HAMMER INSERTIONS

In this section we first analyze the adaptive PMA for the sequential insert pattern, where inserts occur at the front of the PMA. Then we generalize the result to hammer inserts.

Sequential Inserts. We prove the following theorem:

THEOREM 3. *For sequential inserts, the APMA has $O(\log N)$ amortized element moves and $O(1+\log N/B)$ amortized memory transfers.*

We give some notation. In the rest of this section, we assume that u_ℓ is the leftmost node at level ℓ and $v_{\ell-1}$ is the right child of u_ℓ. Recall that leaves have height 0. Suppose that we insert N elements in the front of an array of size cN ($c > 1$). Since we always insert elements at the front, rebalances occur only at the leftmost node u_ℓ ($0 \le \ell \le h$). If we know the number of sweeps of u_ℓ in the process of inserting these N elements, then we also know the total number of moves.

In order to bound the number of sweeps at each level, we need more notation. Let $\mathcal{N}_\kappa(\ell,i)$ be the number of sweeps of the leftmost node u_κ at level κ between the $(i-1)$th sweep and the ith sweep of node u_ℓ. We imagine a virtual parent node u_{h+1} of the root node u_h, where u_{h+1} has size $2cN$. Thus, the time when the root node u_h reaches its upper threshold τ_h, after we insert N elements, is the time when the virtual parent node performs the first rebalance. Thus, $\mathcal{N}_\kappa(h+1,1)$ is the number of sweeps of node u_κ at level κ during the insertion of these N elements ($0 \le \kappa \le h$). Since each sweep of u_κ costs $2^\kappa \log N$ moves, the total number of moves is:

$$\sum_{\kappa=0}^{h} \mathcal{N}_\kappa(h+1,1)2^\kappa \log N.$$

This quantity is the sum of the sweep costs at each level, until the virtual node needs its first rebalance. Thus, the amortized number of element moves is

$$\frac{1}{N}\sum_{\kappa=0}^{h} \mathcal{N}_\kappa(h+1,1)2^\kappa \log N. \tag{9}$$

PROOF OF THEOREM 3: We bound $\mathcal{N}_\kappa(\ell,1)$, the number of sweeps of the leftmost child u_κ at level κ until the ancestor node u_ℓ performs the first rebalance. We decompose this process into three phases. *Phase i of node u_ℓ* ($1 \le i \le 3$), starts after the $(i-1)$th sweep of $u_{\ell-1}$ and ends at the ith sweep of $u_{\ell-1}$. At the end of the last phase, u_ℓ performs its first rebalance, which is the third sweep of $u_{\ell-1}$. Thus, we have at most three sweeps of node $u_{\ell-1}$ before the first rebalance of u_ℓ:

$$\mathcal{N}_\kappa(\ell,1) \le \mathcal{N}_\kappa(\ell-1,1)+\cdots+\mathcal{N}_\kappa(\ell-1,3).$$

Now we prove the above claim analyzing the densities in each phase.

I) We consider the densities of child nodes $u_{\ell-2}$ and $v_{\ell-2}$ of $u_{\ell-1}$ at the end of phase 1. The first rebalance of $u_{\ell-1}$ occurs when $u_{\ell-2}$ reaches its upper threshold $\tau_{\ell-2}$. For sequential inserts, we allocate as many free spaces as possible to $u_{\ell-2}$, while ensuring that $u_{\ell-2}$ and $v_{\ell-2}$ have densities between $\rho_{\ell-1}$ and $\tau_{\ell-1}$. Thus, after the first rebalance, which happens after $\tau_{\ell-2}\text{Cap}(u_{\ell-2})$ inserts, we have densities:

$$\begin{aligned}\text{Density}(u_{\ell-2}) &= \rho_{\ell-1}\\ \text{Density}(v_{\ell-2}) &= \tau_{\ell-2}-\rho_{\ell-1}.\end{aligned}$$

It is immediate that the density setting of $u_{\ell-2}$ is legal; we now explain why the above density setting of $v_{\ell-2}$ is legal, i.e., satisfies the rebalance property. Notice that $\rho_{\ell-1} \le \tau_{\ell-2}-\rho_{\ell-1} \le \tau_{\ell-1}$, since $2\rho_{\ell-1} \le \tau_{\ell-1} < \tau_{\ell-2}$ by (1) and (4) and $\tau_{\ell-2}-\tau_{\ell-1} = O(1/\log N) < \rho_{\ell-1}$ by (1) and (2).

II) We now consider the densities of child nodes $u_{\ell-2}$ and $v_{\ell-2}$ at the end of phase 2. When $u_{\ell-2}$ reaches its threshold again, phase 2 of node u_ℓ ends. After $u_{\ell-1}$ does the second rebalance, which happens after $(\tau_{\ell-2}-\rho_{\ell-1})\text{Cap}(u_{\ell-2})$ inserts, we have densities:

$$\begin{aligned}\text{Density}(u_{\ell-2}) &= 2\tau_{\ell-2}-\rho_{\ell-1}-\tau_{\ell-1}\\ \text{Density}(v_{\ell-2}) &= \tau_{\ell-1}.\end{aligned}$$

It is immediate that the density setting of $v_{\ell-2}$ is legal; we now show that the density setting of $u_{\ell-2}$ is legal. Notice that $\rho_{\ell-1} < 2\tau_{\ell-2}-\rho_{\ell-1}-\tau_{\ell-1} < \tau_{\ell-1}$, because $2\rho_{\ell-1} < \tau_{\ell-2} < \tau_{\ell-2}+(\tau_{\ell-2}-\tau_{\ell-1})$ by (1) and (4) and $2(\tau_{\ell-2}-\tau_{\ell-1}) = O(1/\log N) < \rho_{\ell-1}$ by (1) and (2).

III) Now we consider the densities of child nodes $u_{\ell-2}$ and $v_{\ell-2}$ at the end of phase 3. When $u_{\ell-2}$ reaches its threshold a third time, which happens after $(\tau_{\ell-1}-\tau_{\ell-2}+\rho_{\ell-1})\text{Cap}(u_{\ell-2})$ inserts, phase 3 of node u_ℓ ends. When $u_{\ell-1}$ does the third sweep, the density of $u_{\ell-1}$ is $(\tau_{\ell-2}+\tau_{\ell-1})/2 > \tau_{\ell-1}$, so $u_{\ell-1}$ is above threshold. Thus, the end of phase 3 is the first rebalance of u_ℓ.

Thus, there are at most three sweeps of $u_{\ell-1}$ before the first rebalance of u_ℓ, that is,

$$\mathcal{N}_\kappa(\ell,1) \le \mathcal{N}_\kappa(\ell-1,1)+\mathcal{N}_\kappa(\ell-1,2)+\mathcal{N}_\kappa(\ell-1,3). \tag{10}$$

We cannot simply use the bound $\mathcal{N}_\kappa(\ell,1) \le 3\mathcal{N}_\kappa(\ell-1,1)$ for our analysis, since this bound naively leads to $O(N^{\log(3/2)})$ amortized moves, which is far from our goal of $O(\log N)$.

To establish our bound, we prove the following recurrences for phase 2 and phase 3:

$$\mathcal{N}_\kappa(\ell-1,2) \le 2\mathcal{N}_\kappa(\ell-2,2), \tag{11}$$

and

$$\mathcal{N}_\kappa(\ell-1,3) \le \mathcal{N}_\kappa(\ell-1,2). \tag{12}$$

Solving (10), (11), and (12) will yield the desired bound.

We already showed (10); now we show (11). We proceed by breaking phase 2 into two subphases. The first subphase begins when phase 2 begins, i.e., after the first rebalance of $u_{\ell-1}$, and it ends after the next sweep of $u_{\ell-2}$. The second subphase begins when the first subphase ends, and it ends after the next another sweep of $u_{\ell-2}$. We will show that at the end of subphase 2, $u_{\ell-2}$ is above threshold, meaning that subphase 2 ends with a sweep of $u_{\ell-1}$, i.e., phase 2 ends as well.

- At the beginning of subphase 1, $u_{\ell-3}$ has density $\rho_{\ell-2}$, and the sweep of $u_{\ell-2}$ is triggered once the density of $u_{\ell-3}$ reaches $\tau_{\ell-3}$. At the end of subphase 1, after $(\tau_{\ell-3}-\rho_{\ell-2})\mathrm{Cap}(u_{\ell-3})$ inserts, the density of $u_{\ell-3}$ and $v_{\ell-3}$ are:

$$\begin{aligned}\mathrm{Density}(u_{\ell-3}) &= 2\rho_{\ell-1}-\rho_{\ell-2}+\tau_{\ell-3}-\tau_{\ell-2},\\ \mathrm{Density}(v_{\ell-3}) &= \tau_{\ell-2}.\end{aligned}$$

  It is immediate that the density of $v_{\ell-3}$ is legal; we show that the density of $u_{\ell-3}$ is legal too. Notice that $\rho_{\ell-2} < 2\rho_{\ell-1}-\rho_{\ell-2}+\tau_{\ell-3}-\tau_{\ell-2} < \tau_{\ell-2}$, because $2\rho_{\ell-2} < 2\rho_{\ell-1}$ and $\tau_{\ell-2} < \tau_{\ell-3}$ by (1) and $2\rho_{\ell-1} < \tau_{\ell-1} < \tau_{\ell-2}$ and $\tau_{\ell-3}-\tau_{\ell-2} = O(1/\log N) < \rho_{\ell-2}$ by (1) and (4).

  We now show that the number of sweeps of u_κ in subphase 1 is equal to $\mathcal{N}_\kappa(\ell-2,2)$. Observe that subphase 1 is exactly phase 2 of the node $u_{\ell-1}$ because they both start with the node $u_{\ell-3}$ having density $\rho_{\ell-2}$ and end with the node $u_{\ell-3}$ having density $\tau_{\ell-3}$. Although in subphase 1 and phase 2 of node $u_{\ell-1}$, node $v_{\ell-3}$ has different densities, this difference does not matter because the density of $v_{\ell-3}$ does not affect when subphase 1 and phase 2 of node $u_{\ell-1}$ end.

- At the beginning of subphase 2, $u_{\ell-3}$ has density $2\rho_{\ell-1}-\rho_{\ell-2}+\tau_{\ell-3}-\tau_{\ell-2} > \rho_{\ell-2}$, and the subsequent sweep of $u_{\ell-2}$ is triggered once the density of $u_{\ell-3}$ reaches $\tau_{\ell-3}$ again. Since the density of $v_{\ell-3}$ is $\tau_{\ell-2}$, the density of $u_{\ell-2}$ is $(\tau_{\ell-3}+\tau_{\ell-2})/2 > \tau_{\ell-2}$ at the end of subphase 2, so $u_{\ell-2}$ is above its upper threshold. Thus, the end of subphase 2 is the sweep of $u_{\ell-1}$.

  We now prove that the number of sweeps of u_κ in subphase 2 is less than $\mathcal{N}_\kappa(\ell-2,2)$, because both subphase 2 and phase 2 of node $u_{\ell-1}$ end with node $u_{\ell-3}$ reaching its upper threshold $\tau_{\ell-3}$, but subphase 2 starts with node $u_{\ell-3}$ having density greater than $\rho_{\ell-2}$ while phase 2 of node $u_{\ell-1}$ starts with node $u_{\ell-3}$ having density $\rho_{\ell-2}$.

Thus, there are at most two subphases in phase 2 of node u_ℓ and each subphase has the number of sweeps of node u_κ at most $\mathcal{N}_\kappa(\ell-2,2)$, which shows (11). Since Recurrence (11) has the base case $\mathcal{N}_\kappa(\kappa,2)=1$, we obtain the solution

$$\mathcal{N}_\kappa(\ell-1,2) \le 2^{\ell-\kappa-1}. \tag{13}$$

Now we establish the recurrence in (12). Both phase 2 and phase 3 end with node $u_{\ell-2}$ reaching its upper threshold $\tau_{\ell-2}$, while phase 3 starts with the node $u_{\ell-2}$ having density $2\tau_{\ell-2}-\tau_{\ell-1}-\rho_{\ell-1} > \rho_{\ell-1}$. phase 2 starts with node $u_{\ell-2}$ having density $\rho_{\ell-1}$.

We now establish the desired bound. Plugging (13) and (12) into (10), we have

$$\begin{aligned}\mathcal{N}_\kappa(\ell,1) &\le \mathcal{N}_\kappa(\ell-1,1)+\mathcal{N}_\kappa(\ell-1,2)+\mathcal{N}_\kappa(\ell-1,3)\\ &\le \mathcal{N}_\kappa(\ell-1,1)+2\mathcal{N}_\kappa(\ell-1,2)\\ &\le \mathcal{N}_\kappa(\ell-1,1)+2\cdot 2^{\ell-\kappa-1}\\ &\le 2^{\ell-\kappa+1}.\end{aligned}$$

Finally, the amortized number of moves is

$$\frac{1}{N}\sum_{\kappa=0}^{h}\mathcal{N}_\kappa(h+1,1)2^{\kappa}\log N = \sum_{\kappa=0}^{h}\mathcal{N}_\kappa(h+1,1)2^{\kappa-h}$$
$$\le \sum_{\kappa=0}^{h}(2^{h-\kappa+2})2^{\kappa-h} = \sum_{\kappa=0}^{h}4 = O(\log N).$$

Observe that after any insert the elements are moved from a contiguous group, and the moves can be performed with a constant number of scans. Therefore the amortized number of memory transfer is $O(1+(\log N)/B)$. $\square$

Hammer Inserts. We now consider the hammer insertion distribution, where we always insert the elements at the same rank. We show that the analysis from sequential insertion distribution (Theorem 3) applies here.

THEOREM 4. *When inserted elements have fixed rank (hammer inserts), the APMA has $O(\log N)$ amortized element moves and $O(1+(\log N)/B)$ amortized memory transfers.*

PROOF. In the hammer-insert case, we always insert new elements after a given element x. Notice that in the rebalancing subtree rooted at u_ℓ, there is a unique path from the leaf node containing the element x to the root node u_ℓ. Let node u_i ($i \le \ell$) be the ancestor of x at level i, and let v_i be u_i's sibling.

Intuitively, we want to use the same argument as in the proof of Theorem 3, except that u_{i-1} and sibling v_{i-1} may be either left or right children of u_i. Beyond this change, we want to use the same analysis. This approach almost works, but requires a generalization. In particular, as we show, Recurrences (10) and (12) still hold, but there is one value of ℓ for which Recurrence (11) does not.

We examine the density of u_i after one rebalance; we demonstrate that $\mathrm{Density}(u_i)$ can be different in the sequential-insert case and the hammer-insert case. With sequential inserts, a rebalance tries to put as many elements as possible in v_i without disobeying the upper and lower density thresholds. With hammer inserts, we also want to put as many elements as possible in v_i without disobeying the density thresholds. But now we have the additional constraint that node x remains in u_i. This constraint means that our rebalance may not be able to be as uneven as in the sequential-insert case. However, if we have this constraint, then x will always be at one end of u_i, either at the very beginning or very end. We now look at higher and lower nodes in the tree. For nodes nearer the root, i.e., u_j ($j > i$), the position of x does not constrain the allowed splits, and the elements in u_j are exactly the same as for sequential inserts. For nodes nearer the leaves, i.e., u_j ($j < i$), the insertion pattern of u_j matches the sequential-insert case, although with different densities.

Thus, Recurrence (11) is still true except for one intermediate node u_i. However, $\mathcal{N}_\kappa(i,2) \le \beta\mathcal{N}_\kappa(i-1,2)$ is true for some constant β. Thus, the solution for Recurrence (11) is $\mathcal{N}_\kappa(\ell-1,2) \leq 2^{\ell-\kappa-2}\beta$. Thus, the solution for Recurrence (10) is $\mathcal{N}_\kappa(\ell,1) \leq 2^{\ell-\kappa}\beta$, and the theorem follows.

4. RANDOM AND BULK INSERTIONS

In the previous section, we analyzed the sequential and hammer insertion distributions, where the inserts hammer on one part of the PMA. In this section we analyze the random insertion distribution, where we insert after random elements in the array. Then we generalize all of these distributions and consider the bulk insertion distribution.

The bulk insertion distribution for function N^α, $0 \leq \alpha \leq 1$, is defined as follows: pick a random element and insert N^α elements after it; then pick another element and repeat. Bulk insert generalizes all distributions seen so far: For $\alpha = 0$, we have random inserts, and for $\alpha = 1$, we have sequential or hammer inserts.

Random Inserts. We now give the performance for the PMA and APMA with random inserts.

THEOREM 5 ([7, 14]). *For random inserts, with high probability the original PMA and the APMA have $O(\log N)$ element moves and $O(1+(\log N)/B)$ memory transfers.*

Even simpler rebalance schemes perform well under random inserts, as shown in [7, 14]. These papers show that there are $O(\log N)$ moves with high probability for random inserts, even with the most basic of rebalances: When we insert an element y after an element x, we simply push the elements to the right or left to make room for y. The maximum number of element moves is $O(\log N)$ with high probability. Thus, for the PMA, as long as the density thresholds in the leaves is a constant less than 1, we need no big rebalances in the tree.

Bulk Inserts. For bulk inserts, we have the following:

THEOREM 6. *For bulk inserts with $f(N) = N^\alpha$ ($0 \leq \alpha \leq 1$), the APMA achieves $O(\log N)$ amortized element moves and $O(1+(\log N)/B)$ memory transfers.*

The intuition for Theorem 6 is as follows: Conceptually, we divide the virtual tree into a top tree with $\Theta(N/(f(N)\log N))$ leaves, each of which is the root of a bottom tree B_i with $\Theta(f(N))$ leaves, i.e., $\Theta(f(N)\log N)$ array positions. Thus, we split the virtual tree at height $h' = \lceil \alpha \log N \rceil$. Bulk inserts can be analyzed by looking at the process as a combination random and hammer inserts: random inserts in the top tree A with big leaf nodes of size $f(N)\log N$ and hammer inserts in a bottom tree B_i of size $f(N)\log N$. In an insertion, we randomly choose a leaf node of the top tree A and do a hammer insert at the bottom subtree of the chosen leaf node of A.

We first show that $f(N) = N^\alpha$ ($0 \leq \alpha \leq 1$) hammer inserts into B_i costs $O(\log N)$ amortized moves when the nodes all are well balanced. Then, we explain that these $f(N)$ inserts trigger at most one rebalance in the top tree A. Thus, from the point of view of A, there is a big element of size $f(N)$ inserted, and this big insert costs $O(\log N)$ amortized moves in the leaf node.

We prove the following lemma for $f(N) = N^\alpha$.

LEMMA 7. *Consider inserting $f(N) = N^\alpha$ elements after an element x in subtree B_i of size $N^\alpha \log N$. Suppose that at the beginning of these insertions, each node in B_i is well balanced. Then, the amortized number of moves is $O(\log N)$ and the amortized number of memory transfers is $O(1+(\log N)/B)$.*

The proof appears in the full version. Based on Lemma 7, Theorem 6 is proved as follows.

PROOF OF THEOREM 6: We consider each bottom subtree B_i. Suppose that an ancestor of the root of B_i does a rebalance. Then the root of B_i has density at most $\tau_{h'+1}$. Thus, we can insert at least $(\tau_{h'} - \tau_{h'+1})\Theta(N^\alpha \log N) = \Theta(N^\alpha)$ elements without triggering sweeps above level h', i.e., inserting N^α elements in B_i triggers at most one rebalance in top subtree A.

Now we consider a *round* of N^α inserts into some bottom subtree B_i. We show that there are $O(\log N)$ amortized element moves in the APMA. Recall that we use the predictor to store recent inserts. For the first N^α inserts, the predictor only uses one cell. When the next N^α inserts start to hammer, the predictor uses the second cell to store new elements. After the count number in the second cell reaches $\log N$, which means there are $\log N$ new elements at the second position, the count number in the first cell begins to decrease. Thus, at most $2\log N$ inserts remove the first cell, meaning that the hammer-insert pattern starts after the first $2\log N$ inserts. Thus, we divide the N^α inserts in the round into two parts: the first $2\log N$ ones and the $N^\alpha - 2\log N$ subsequent ones. This is one dividing point.

The second dividing point is when some insert triggers a rebalance in the top subtree A. We assume the second dividing point is after the first one. The alternative is similar to the following analysis, although somewhat easier. These two dividing points split the round into three parts. We analyze the cost of the rebalance in the bottom subtree B_i for these parts as follows:

1. The rebalance cost for the first part, the insertion of the first $2\log N$ elements, is at most $3N^\alpha \log N$. To see why, observe that there exists a node u' of size N^α, such that these $2\log N$ elements trigger at most one rebalance above u', by an argument similar to that above. This rebalance is within B_i, and therefore costs at most $N^\alpha \log N$. Thus, the total cost is the cost of this rebalance plus the cost of the rebalances below u', at most $(2\log N - 1)N^\alpha$.
2. The second part is from the $(2\log N)$th element insert to the element insert triggering the rebalance in the top subtree A. The total cost is at most the worst-case cost in Lemma 7, which is $O(N^\alpha \log N)$.
3. The third part is from the element insert triggering the rebalance in the top subtree A to the last element insert of these N^α elements. From Lemma 7, the cost is less than the cost to insert all N^α elements in subtree B_i whose ancestor did the rebalance, which is $O(N^\alpha \log N)$.

Thus, without counting the rebalance cost in the top subtree A, the average cost for each round is $3O(N^\alpha \log N)/N^\alpha = O(\log N)$. If we can show that the average cost in the top subtree A is also $\log N$, then the theorem is proved.

From the view point of top subtree A, the bulk insert is similar to random inserts of "big elements" of size N^α in A, because big element triggers at most one rebalance in A and a leaf node of size $N^\alpha \log N$ is a black box that has $O(\log N)$ amortized moves. So the bulk inserts is: randomly choose a leaf node in A, a black-box operation to insert N^α elements in the

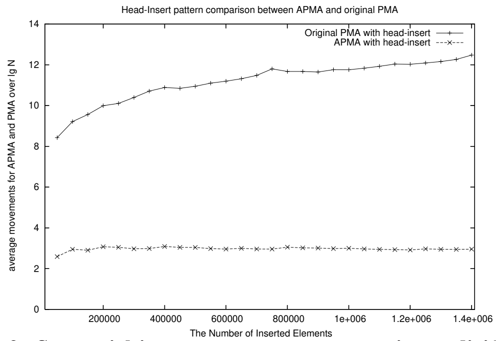

Figure 2: Sequential inserts: average moves per insert divided by $\lg N$. The array size grows to two million and 1.4 million elements are inserted.

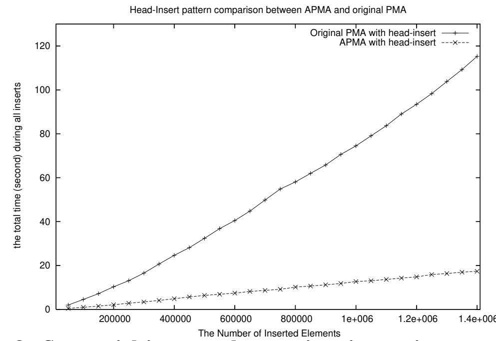

Figure 3: Sequential inserts: the running time to insert up to 1.4 million elements.

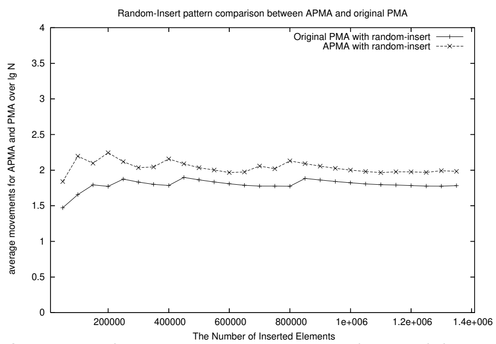

Figure 4: Random inserts: average moves per insert divided by $\lg N$. The array size grows to two million and 1.4 million elements are inserted.

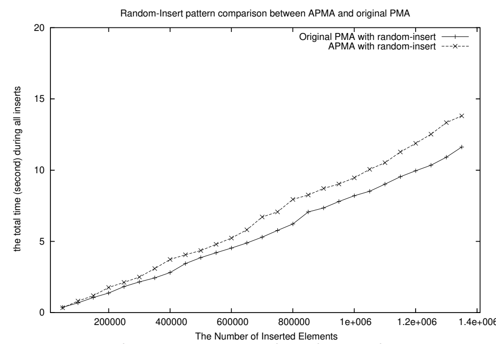

Figure 5: Random inserts: the running time to insert up to 1.4 million elements.

leaf node, each with $O(\log N)$ moves. If the leaf node reaches its threshold, then a rebalance is triggered at most once in A. Thus, as in Theorem 5, we have $O(\log N)$ element moves in the top subtree A. As before, the memory-transfer bound follows because all rebalances are to contiguous groups of elements. $\square$

5. EXPERIMENTAL RESULTS

In this section we describe our simulation and experimental study. We show that our results are consistent with the asymptotic bounds from the previous sections and suggest the constants involved. We also demonstrate that the bookkeeping for the adaptive structure has little computational overhead.

We ran our experiments as follows: For each insert pattern, we began with an empty array and added elements until the array contained roughly 1.4 million elements. We began our measurements once the array had size at least 100,000. We recorded the amortized number of element moves per insert as well as the running times. We considered the sequential, hammer, random, and bulk insertion distributions from the previous sections. We also added noise to the distributions, combining, for example, the hammer and random distributions, showing that the predictor is resilient to this noise. Each graph plots the intermediate data points in a single run.

We ran our experiments on a Pentium 4 CPU 3.0GHZ, with 2GB of RAM, running Windows XP professional, and a 100G ATA disk drive. Our file contained up to 2^{21} keys, and the total memory used was up to 1.4 GB. We implemented a search into the PMA as a simple binary search. The binary search was appropriate since our experiments were small enough that they did not involve paging to disk. Consequently, the search time was dominated by the insertion time into the PMA.

The adaptive PMA is ultimately targeted for used in cache-oblivious and locality-preserving B-trees, where the search time becomes relatively more expensive because the data structures do not fit in main memory. In this case the binary search will be too slow because it lacks sufficient data locality. (The number of memory transfers for the PMA insert is $O(1+(\log N)/B)$, which is dominated by the cost of a binary search, $O(\log \lceil N/B \rceil)$, as well as the optimal external-memory search cost, $O(1+\log_B N)$.) Thus, our next round of experiments on larger data sets is to be run with the objective of speeding up inserts in the cache-oblivious B-tree.

Sequential inserts. We first compared the adaptive and traditional PMAs on sequential insertions. For sequential inserts of roughly 1.4 million elements, the APMA has four times fewer element moves per insertion than the traditional PMA and running times that are nearly seven times faster.

Figure 2 shows the average number of element moves in the PMAs. The x-axis indicates the number of inserted elements up to 1.4 million. The y-axis indicates the number of element moves divided by $\lg N$. For both the adaptive and traditional PMA, we choose the upper and lower density thresholds as follows: $\tau_0 = 0.92$, $\tau_h = 0.7$, $\rho_h = 0.3$, and $\rho_0 = 0.08$. In our experiments, we double when the array gets too full. Thus, before doubling, the array has density over 0.7 and after, the array has density over 0.35. (By increasing the array size by only a $(1+\varepsilon)$-factor for constant ε, we can make the density

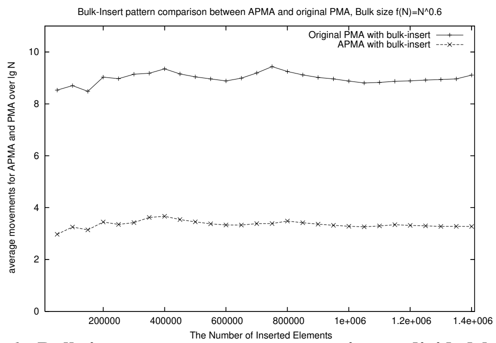

Figure 6: Bulk inserts: average moves per insert divided by $\lg N$. The array size grows to two million and 1.4 million elements are inserted.

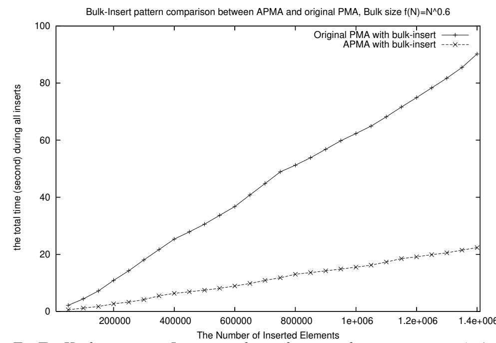

Figure 7: Bulk inserts: the running time to insert up to 1.4 million elements.

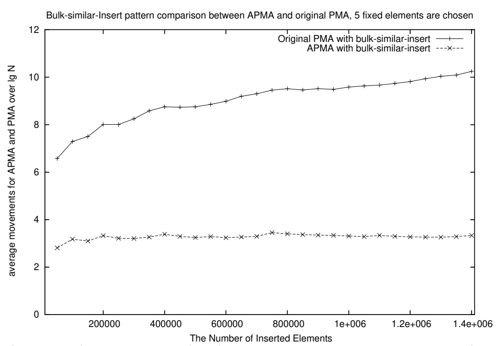

Figure 8: Multiple sequential inserts: average moves per insert divided by $\lg N$. The array size grows to two million and 1.4 million elements are inserted.

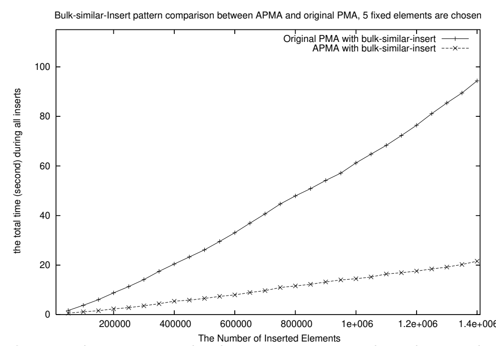

Figure 9: Multiple sequential inserts: the running time to insert up to 1.4 million elements.

of the entire array at least $(1+\varepsilon)\rho_h$ with only a small additive increase in the number of elements moved. Thus, we can have an array whose density is always arbitrary close to 70% full.) The roughly flat line shows the performance of the APMA. These experiments suggest that the constant in front of the $\lg N$ (see Theorem 3) is roughly 2.5 for the density thresholds chosen. Because we are measuring number of element moves, these results are machine independent. Figure 3 gives the running times for our experiment. Observe that the APMA runs almost 7 times faster even though the amortized number of element moves is only 4 times smaller. Hence, the overhead for the adaptive PMA is small. We suspect that this decrease has to do with caching issues; the APMA has a smaller working set than the traditional PMA.

Random inserts. For random insertions the traditional PMA performs slightly better than the APMA because there is seemingly no advantage in uneven rebalalances. For sequential insertions of 1.4 million elements with the same density thresholds and axes as in Figures 2 and 3, both the adaptive and traditional PMAs have the same asymptotic performance (see Theorem 5). The traditional PMA's constant seem to be less than 12% smaller. Figures 4 and 5 show that both the amortized number of element moves and the running times are comparable, with the traditional PMA performing slightly better, as expected. Figure 5 indicates that the bookkeeping overhead for the APMA is small.

Bulk inserts. We next investigated the bulk-insert distribution, comparing both the adaptive and traditional PMAs. For sequential insertions of 1.4 million elements, the APMA has roughly 3.2 times fewer element moves per insertion than the traditional PMA and running times that are over 4.7 times faster. Figure 6 shows the average number of elements moves in the PMAs with the same thresholds as in Figure 2 and bulk parameter $N^{0.6}$. The roughly flat line shows the performance of the APMA. These experiments suggest that the constant in front of the $\lg N$ (see Theorem 6) is roughly 2.7 for the chosen density thresholds and bulk parameter. Figure 7 shows the running times of the traditional and adaptive PMAs.

Multiple sequential inserts. We next consider a distribution that performs sequential inserts into multiple parts of the array at once. We first choose R random elements and then insert one element at a time after one of these chosen elements. As long as the number of chosen elements R is less than the number of elements stored in the predictor, most predictions are good and the performance of APMA remains $O(\log N)$. Figures 8 and 9 compare the performance of the traditional and adaptive PMAs when we choose 5 fixed elements. The APMA in this case has a performance only slightly worse than that in the sequential-insert case while tradition PMA still performs much worse.

Half random and half sequential inserts. Finally, we analyze a distribution that adds noise to sequential inserts. We decide randomly whether to insert a new element at the front of the PMA or after a random element. Thus, roughly half of the inserted elements form random noise. Figures 10 and 11 compare the performance of the traditional PMA and APMA. The roughly flat curve in Figure 10 is the performance of APMA, which is slightly worse than that in random inserts and better than that in sequential inserts, while the performance of tradi-

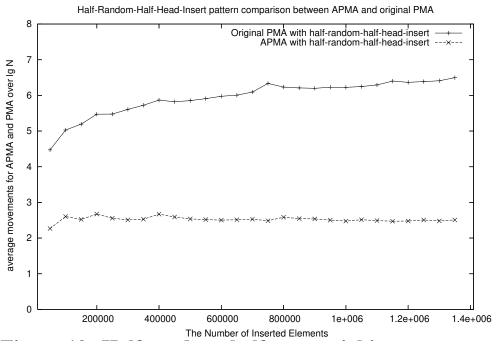

Figure 10: Half random, half sequential inserts: average moves per insert divided by $\lg N$. The array size grows to two million and 1.4 million elements are inserted.

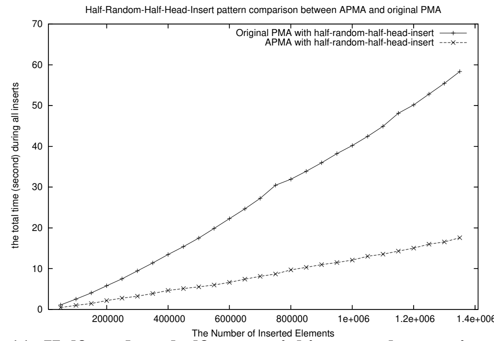

Figure 11: Half random, half sequential inserts: the running time to insert up to 1.4 million elements.

tional PMA is about 3 times worse than that of random inserts.

6. CONCLUSION

We introduced an adaptive packed-memory array. The adaptive PMA guarantees a performance at least as good as that of the traditional PMA, while simultaneously adapting to common insertion distributions. Thus, the adaptive PMA always achieves at most $O(\log^2 N)$ amortized element moves and $O(1+(\log^2 N)/B)$ memory transfers per update, but it achieves only $O(\log N)$ amortized element moves and $O(1+(\log N)/B)$ memory transfers for sequential inserts, hammer inserts, random inserts, and bulk inserts. Our simulations and experiments are consistent with these asymptotic bounds. Several open problems remain. For example, can we show some type of working-set property for an adaptive PMA? Perhaps such an investigation will require study into the design of other predictors. The next step in this research is to use the adaptive PMA in a cache-oblivious B-tree and to measure the speedup obtained for updates.

7. ACKNOWLEDGMENTS

We are grateful to Ziyang Duan for proposing an early uneven-rebalance strategy. We are grateful to Bradley Kuszmaul for suggesting that we focus on sequential inserts. We are grateful to Yue Wang for important discussions and for reading early drafts of this paper.

8. REFERENCES

[1] M. A. Bender, R. Cole, E. D. Demaine, and M. Farach-Colton. Scanning and traversing: Maintaining data for traversals in a memory hierarchy. In *Proc. 10th Annual European Symposium on Algorithms (ESA)*, volume 2461 of *Lecture Notes in Computer Science*, pages 139–151, 2002.

[2] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proc. 41st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 399–409, 2000.

[3] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. *SIAM Journal on Computing*, 35(2):341–358, 2005.

[4] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proc. 13th Annual Symposium on Discrete Mathematics (SODA)*, pages 29–38, 2002.

[5] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. *Journal of Algorithms*, 3(2):115–136, 2004.

[6] M. A. Bender, M. Farach-Colton, and B. Kuszmaul. Cache-oblivious string B-trees. In *Proc. 25th Symposium on Principles of Database Systems (PODS)*, 2006.

[7] M. A. Bender, M. Farach-Colton, and M. Mosteiro. Insertion sort is $O(n\log n)$. In *Proc. 3rd International Conference on Fun with Algorithms (FUN)*, pages 16–23, 2004.

[8] M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuszmaul. Concurrent cache-oblivious B-trees. In *Proc. 17th Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 228–237, 2005.

[9] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height. In *Proc. 13th Annual Symposium on Discrete Algorithms (SODA)*, pages 39–48, 2002.

[10] P. F. Dietz. Maintaining order in a linked list. In *Proc. Symposium on the Theory of Computing (STOC)*, pages 122–127, 1982.

[11] P. F. Dietz, J. I. Seiferas, and J. Zhang. A tight lower bound for on-line monotonic list labeling. In *Proc. 4th Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 824 of *Lecture Notes in Computer Science*, pages 131–142, 1994.

[12] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proc. 19th Annual Symposium on Theory of Computing (STOC)*, pages 365–372, 1987.

[13] P. F. Dietz and J. Zhang. Lower bounds for monotonic list labeling. In *Proc. 2nd Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 447 of *Lecture Notes in Computer Science*, 1990.

[14] A. Itai, A. G. Konheim, and M. Rodeh. A sparse table implementation of priority queues. In *Proc. 8th Internationl Colloquium on Automata, Languages, and Programming (ICALP)*, volume 115 of *Lecture Notes in Computer Science*, pages 417–431, 1981.

[15] I. Katriel. Implicit data structures based on local reorganizations. Master's thesis, Technion – Israel Inst. of Tech., Haifa, May 2002.

[16] V. Raman. Locality-preserving dictionaries: theory and application to clustering in databases. In *Proc. 18th Symposium on Principles of Database Systems (PODS)*, pages 337–345, 1999.

[17] A. K. Tsakalidis. Maintaining order in a generalized linked list. *Acta Informatica*, 21(1):101–112, 1984.

[18] D. Willard. Maintaining dense sequential files in a dynamic environment (extended abstract). In *Proc. 14th Annual Symposium on Theory of Computing (STOC)*, pages 114–121, 1982.

[19] D. E. Willard. Good worst-case algorithms for inserting and deleting records in dense sequential files. In *Proc. International Conference on Management of Data (SIGMOD)*, pages 251–260, 1986.

[20] D. E. Willard. A density control algorithm for doing insertions and deletions in a sequentially ordered file in good worst-case time. *Information and Computation*, 97(2):150–204, 1992.