Tiered Vectors: Efficient Dynamic Arrays for Rank-Based Sequences

Michael T. Goodrich and John G. Kloss II

Johns Hopkins Univ., Baltimore, MD 21218 USA
{goodrich,jkloss},cs.jhu.edu

Abstract. We describe a data structure, the *tiered vector*, which is an implementation of the Vector ADT that provides $O(1/\epsilon)$ worst case time performance for rank-based retrieval and $O(n^\epsilon)$ amortized time perofor-mance for rank-based insertion and deletion, for any fixed $\epsilon > 0$. We also provide results from experiments involving the use of the tiered vector for $\epsilon = 1/2$ in JDSL, the Data Structures Library in Java.
Keywords: abstract data type, vector, dynamic array, Java, JDSL.

1 Introduction

An array is perhaps the most primitive data structure known; it is hard to imagine any non-trivial program that does not use one. Almost all high-level languages and assembly languages have some built-in concept for accessing elements by their indices in an array. But an array is a static data type; it does not allow for element insertions and deletions, just element replacements and accesses. There is nevertheless a great need for dynamic arrays as high-level programming structures, for they can free a programmer from having to artificially constrain his or her set of elements to be of a certain fixed size. This is in fact the motivation for the inclusion of a dynamic array data type in the Java language.

The Vector/Rank-Based-Sequence Abstract Data Type. A *Vector*, or *Rank-Based Sequence*, is a dynamic sequential list of elements. Each element e in a vector is assigned a *rank*, which indicates the number of elements in front of e in the vector. Rank can also be viewed as a current "address" or "index" for the element e. If there are n elements currently stored in a rank-based sequence, S, then a new element may be inserted at any rank r in $\{0, 1, 2, \ldots, n\}$, which forces all elements of rank $r, \ldots, n-1$ in S to have their respective ranks increased by one (there are no such elements if $r = n$, of course). Likewise, an existing element may be removed from any rank r in $\{0, 1, 2, \ldots, n-1\}$, which forces all elements of rank $r+1, \ldots, n-1$ in S to have their respective ranks decreased by one. Formally, we say that the data type *Vector* or *Rank-Based Sequence* (we use the terms interchangably) support the following operations:

***insertElemAtRank(r,e)*:** Insert an element e into the vector at rank r.
***removeElemAtRank(r)*:** Remove the element at rank r and return it.
***elemAtRank(r)*:** Retrieve the element e at rank r.

Standard Implementations. There are two standard implementations of the Vector abstract data type (ADT). In the most obvious implementation we use an array S to realize the vector. To retrieve an element of rank r from this vector we simply return the element located at the memory address $S[r]$. Thus, accesses clearly take constant time. Insertions and deletions, on the other hand, require explicit shifting of elements of rank above r. Thus, a vector update at rank r takes $O(n-r+1)$ time in this implementation, assuming that the array does not need to grow or shrink in capacity to accommodate the update. Even without a growth requirement, the time for a vector update is $O(n)$ in the worst case, and is $O(n)$ even in the average case. If the array is already full at the time of an insertion, then a new array is allocated, usually double the previous size, and all elements are copied into the new array. A similar operation is used any time the array should shrink, for efficiency reasons, because the number of elements falls far below the array's capacity. This is the implementation, for example, used by the Java Vector class.

The other standard implementation uses a balanced search tree to maintain a rank-based sequence, S. In this case ranks are maintained implicitly by having each internal node v in the search structure maintain the number of elements that are in the subtree rooted at v. This allows for both accesses and updates in S to be performed in $O(\log n)$ time. If one is interested in balancing access time and update time, this is about as good an implementation as one can get, for Fredman and Saks [5] prove an amortized lower bound of $\Omega(\log n/\log\log n)$ in the cell probe model for accesses and updates in a rank-based sequence.

In this paper we are interested in the design of data structures for realizing rank-based sequences so as to guarantee constant time performance for the *elemAtRank(r)* operation. This interest is motivated by the intuitive relationship between the classic array data structure and the Vector abstract data type. Constant time access is expected by most programmers of a Vector object. We therefore desire as fast an update time as can be achieved with this constraint. Our approach to achieving this goal is to use a multi-level dynamic array structure, which we call the "tiered vector."

Relationships to Previous Work. There are several hashing implementations that use a similar underlying structure to that of the tiered vector, although none in a manner as we do or in a way that can be easily adapted to achieve the performance bounds we achieve. Larson [1] implements a linear hashing scheme which uses as a base structure a directory that references a series of fixed size segements. Both the directory and segements are of size $l = 2^k$ allowing the use of a bit shift and mask operation to access any element within the hash table. However, Larson's method is a hashed scheme and provides no means of rank-order retrieval or update.

Sitarski [9] also uses a s^k fixed size directory-segement scheme for dynamic hash tables, which he terms "Hashed Array Trees." His method provides an efficient implementation for appending elements to an array, but does not provide an efficient method for arbitrary rank-based insertion or deletion into the array.

Our Results. We present an implementation of Vector ADT using a data structure we call the "tiered vector." This data structure provides, for any fixed constant $\epsilon > 0$, worst-case time performance of $O(1/\epsilon)$ for the *elemAtRank(r)* method, while requiring only $O(n^\epsilon)$ amortized time for the *insertElemAtRank(r,e)* and *removeElemAtRank(r)* methods (which sometimes run much faster than this, depending on r). Intuitively, keeping access times constant means we are essentially maintaining ranks explicitly in the representation of the vector. The main challenge, then, is in achieving fast update times under this constraint.

Besides providing the theoretical framework for the tiered vector data structure, we also provide the results of extensive experiments we performed in JDSL, the Data Structures Library in Java, on the tiered vector for $\epsilon = 1/2$. These results show, for example, that such a structure is competitive with the standard Java implementation for element accesses while being significantly faster than the standard Java Vector implementation for rank-based updates.

2 A Recursive Definition of the Tiered Vector

We define the tiered vector recursively. We begin with the base case, V^1, a 1-level tiered vector.

A 1-Level Tiered Vector. The base component, V^1, of the tiered vector is a simple extension of an array implementation of the well-known deque ADT. (The deque (or double-ended queue) is described, for example, by Knuth [7] as a linear list which provides constant time insert and delete operations at either rank the head or tail of this list.) This implementation provides for constant-time rank-based accesses and allows any insertion or deletion at rank r to be performed in $O(\min\{r, n-r\} + 1)$ time.

We use an array A of fixed size l to store the elements of the rank-based sequence S. We view A as a circular array and store indices h and t which respectively reference the head and tail elements in the sequence. Thus, to access the element at rank r we simply return $A[h + r \bmod l]$, which clearly takes $O(1)$ time. To perform an insertion at rank r we first determine whether $r < n - r$. If indeed $r < n - r$, then we shift each element of rank less than r down by 1; i.e., for $i = h, \ldots, h + r - 1 \bmod l$, we move $A[i]$ to $A[i + l - 1 \bmod l]$. Altnernatively, if $r \geq n - r$, then we shift each element of rank greater than or equal to r up by 1. Whichever of these operations we perform, we will have opened up the slot at rank r, $A[h + r \bmod l]$, where we can place the newly inserted element. (See Figure 1.) Of course, this implementation assumes that there is any empty "slot" in A. If there is no such slot, i.e., $n = l$, then we preface our computation by allocating a new array A of size $2l$, and copying all the elements of the old array to the first l slots of the new array.

Element removals are performed in a similar fashion, with elements being shifted up or down depending on whether $r < n - r$ or not. We can optionally also try to be memory efficient by checking if $n < l/4$ after the removal, and if so, we can reallocate the elements of A into a new array of half the size. This implementation gives us the following performance result:

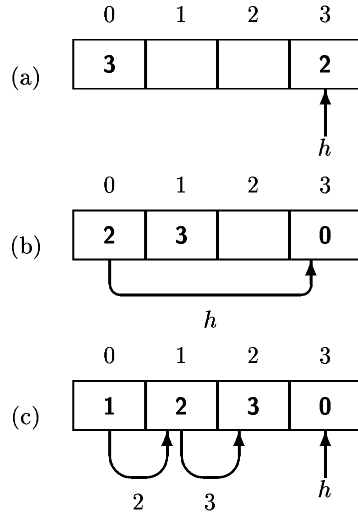

Fig. 1. A 1-level tiered vector V implemented with an array A of capacity 4. (a) An initial state: $V = (2, 3)$; (b) Element 0 is added at rank 0, so $V = (0, 2, 3)$, which causes the head pointer to move in A, but no shifting is needed; (c) Element 1 is added at rank 1, so $V = (0, 1, 2, 3)$, which causes 2 and 3 to shift up in A.

Lemma 1. *A 1-level tiered vector, V^1 can be maintained as a rank-based sequence S such that any access is performed in $O(1)$ worst-case time and any update at rank r is performed in $O(\min\{r, n - r\} + 1)$ amortized time.*

Proof. We have already described why accesses run in constant time and why an update at rank r runs in time $O(\min\{r, n - r\} + 1)$ if no resizing is needed. The amortized bound follows from two simple observations, by using the *accounting method* for amortized analysis (e.g., see [6]). First, note that any time we perform a size increase from l to $2l$ we must have done l insertions since the last resizing. Hence, we can charge growth resizing (which takes $O(l)$ time) to those previous insertions, at a constant cost per insertion. Second, note that any time we perform a size decrease from l to $l/2$ we must have done $l/4$ removals since the last resizing. Thus, we can charge growth resizing to those previous removals, at a constant cost per removal. This gives us the claimed amortized bounds.

Note in particular that insertions and removals at the head or tail of a sequence S run in constant amortized time when using a 1-level tiered vector V^1 to maintain S.

The General k-Level Tiered Vector. The k-level tiered vector is a set of m indexable $(k-1)$-level tiered vectors, $\{V_0^{k-1}, V_1^{k-1}, \ldots, V_{m-1}^{k-1}\}$. Each V_i^{k-1} vector is of exact size l where $l = 2^k$ for some integer parameter k, except possibly

the first and last non-empty vectors, which may hold fewer than l elements. The vectors themselves are stored in a 1-level tiered vector, V, which indexes the first non-empty vector to the last non-empty vector. The total number of elements a tiered vector may hold before it must be expanded is lm, and the number of non-empty V_i^{k-1} vectors is always at most $\lfloor n/l \rfloor + 2$.

Element Retrieval. Element retrieval in a tiered vector is similar to methods proposed by Larson [1] and Sitarski [9], complicated somewhat by double-ended nature of the top level of the vector V. To access any element of rank r in the k-level tiered vector V^k we first determine which V_i^{k-1} vector contains the element in question by calculating $i \leftarrow \lceil (r - l_0)/l \rceil$, where l_0 is the number of elements in the first non-empty vector in V (recall that we always begin vector indexing at 0). We then return the element in V_i^{k-1} by recursively requesting the element in that vector of rank r if $i = 0$ and rank $r - (i-1)l - l_0$ otherwise.

Element Insertion. Insertion into a tiered vector is composed of two phases: a *cascade* phase and a *recurse* phase. In the cascade phase we make room for the new element by alternately popping and pushing elements of lower-level queues to the closest end of the top-level vector, and in the recurse phase we recursively insert the new element into the appropriate $(k-1)$-level vector on the next level down.

Let us describe the cascade phase in more detail. Without loss of generality, let us assume that $r \geq n - r$, so we describe the cascade phase as a series of pops and pushes from the $(k-1)$-level vector currently containing the rank-r element in V to the last non-empty $(k-1)$-level vector in V. (The method for popping and pushing to the front of V when $r < n - r$ is similar, albeit in the opposite direction.) We begin by first determining the $(k-1)$-level vectors in which the elements at rank r and rank $n-1$ are located, where the element of rank $n-1$ indicates the last element in the tiered vector. Term these vectors as V_{sub}^{k-1} and V_{end}^{k-1}, respectively. These vectors are used as the bounds for a series of pair-wise *pop-push* operations. For each vector V_i^{k-1}, $sub \leq i < end$, we will pop its last item and push it onto the beginning of the vector V_{i+1}^{k-1}. Each such operation involves an insertion and removal at the beginning or end of a $(k-1)$-level tiered vector, which is a very fast operation, as we shall show each such operation takes only $O(k)$ time. Since there are a total of m vectors this cascading phase requires a maximum of $O(mk)$ operations.

In the *recurse* phase we simply recursively insert the element into V_{sub}^{k-1} at the appropriate rank r' (which is determined as described in the element retrieval description above). (See Figure 2.) Thus, if we let $I_k(r,n)$ denote the running time of inserting an element of rank r in a tiered vector of size n, then, assuming no resizing is needed, the total running time for this insertion is:

$$I_k(r,n) \leq \lfloor r/l \rfloor I_{k-1}(1,l) + I_{k-1}(r',l).$$

This implies that $I_k(1,n)$ is $O(k)$. More generally, we can show the following:

Lemma 2. *Insertion into a k-level tiered vector where expansion is not required can be implemented in* $O(\min\{\lceil r/n^{1/k} \rceil, k^2 n^{1/k}, \lceil (n-r)/n^{1/k} \rceil\} + k)$ *time.*

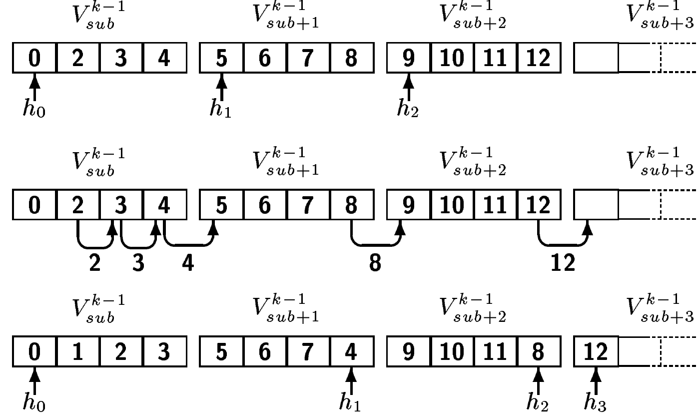

Fig. 2. Insertion of element **1** at rank r in a 2-level tiered vector.

Proof. If we choose l to be $O(n^{\frac{k-1}{k}})$, and maintain m to be $O(\lceil n/l \rceil)$, then

$$I_k(r,n) \text{ is } O(\min\{\lceil r/n^{1/k} \rceil, k^2 n^{1/k}, \lceil (n-r)/n^{1/k} \rceil\} + k),$$

by an induction argument that we leave to the reader.

Note that if $r = 1$, then the above time bound for insertion is $O(k)$. Also note that if $k \geq 1$ is any fixed constant, then this bound is $O(n^{1/k})$ for any r. Throughout the remainder of our algorithmic description we are going to maintain that the size m of the top-level vector is $O(n^{1/k})$.

Resizing During an Insertion. A special case occurs when the number of elements in the tiered vector, n equals the maximum space provided, ml. In this case the data structure must be expanded in order to accomodate new elements. However, we also wish to preserve the structure of the tiered vector in order to insure that the size, l, of the sublists is kept at $O(n^{k-1/k})$. We achieve this by first reseting the fixed length l to $l' \leftarrow 2l$ and then creating a new set of l'-sized $(k-1)$-level tiered vectors under the top-level vector V. We do this by recursively merging pairs of subvectors, so that the size of each subvector doubles in size. This implies, of course that number of non-empty $(k-1)$-level subvectors of V becomes $m' = \frac{1}{4}l'$. The total time for performing such a resizing is $O(n)$, assuming that m is $O(n^{1-\delta})$ for some constant $\delta > 0$, which is the case in our implementation. As in our description of expansions needed for a 1-level tiered vector, this linear amount of work can be amortized to the previous $n/2$ insertions, at a constant cost each. Thus, we have the following:

Theorem 1. *Insertion into a k-level tiered vector can be implemented in amortized time $O(\min\{\lceil r/n^{1/k} \rceil, k^2 n^{1/k}, \lceil (n-r)/n^{1/k} \rceil\} + k)$.*

Expansion is demonstrated in Figure 3, where a 2-level tiered vector of fixed subarray size 4 is expanded into a 2-level tiered vector of subarray size 8.

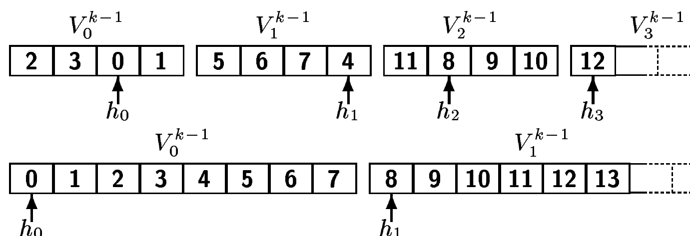

Fig. 3. Expansion and reordering of a 2-level tiered vector..

Element Deletion. Deletion is simply the reverse of insertion and uses a similar *cascade* and *recurse* process. Without loss of generality, let us again assume that $r \geq n - r$, so that any casing we need to perform is for ranks greater than r. As with the insertion operation, we begin by determining in which subvectors the elements at rank r and rank $n - 1$ are located and term these subarrays V_{sub}^{k-1} and V_{end}^{k-1}. Then for each pair of subarrays, V_i^{k-1} and V_{i+1}^{k-1}, $sub \leq i \leq end$, we will pop the head of V_{i+1}^{k-1} and push it onto the tail of V_i^{k-1}. Since this process is simply the reverse of insert's *cascade* phase, we are guaranteed a maximum of $O(m)$ operations.

During the second phase we perform a recursive removal in V_{sub}^{k-1} to close up the space vacated by the removed element. This implies a running time for deletion, without resizing, that is essentially the same as that for insertion.

A special case of delete occurs when the number of elements remaining in the tiered vector equals $\frac{1}{8}ml$. At this point we must reduce the size of the tiered vector inorder to preserve the desired asymptotic time bounds for both insertions and deletions. We first reset the fixed length l to $l' \leftarrow \frac{1}{2}l$ and then create a new set of size-l' subvectors, by a recursive splitting of the $(k-1)$-level subvectors. Note that by waiting until the size of a tiered vector goes below $\frac{1}{8}ml$ to resize, we avoid having resizing operations coming "on the heals" of each other. In particular, if we do perform such a shrinking resizing as described above, then we know we must have performed $n/4$ deletions since the last resizing; hence, may amortize the cost of this resizing by charging each of those previous deletions a constant amount. This give us the following:

Theorem 2. *Insertion and deletion updates in a k-level tiered vector can be implemented in amortized time* $O(\min\{\lceil r/n^{1/k}\rceil, k^2n^{1/k}, \lceil (n-r)/n^{1/k}\rceil\} + k)$ *while allowing for rank-based element access in* $O(k)$ *worst-case time.*

3 Implementation Decisions and Experiments

We implemented the scheme described above for $k = 2$ and performed several experiments with this implementation to test various design decisions. Our implementation used JDSL, the Data Structures Library in Java developed as a

prototype at Brown and Johns Hopkins University. This implementation was tested against the two best-known Java vector implementations: the Java vector implementation that is a part of the standard Java JDK language distribution and the dynamic array implementation included in JGL, the Generic Library in Java. All of our experiments were run on a Sun Sparc 5 computer in single-user mode.

Since our experimental setup used $k = 2$ we made a simplifying modfication in the definition of the tiered vector so that the top-level vector is a standard vector sequence $\mathcal{S}$ and each vector below it is also a standard vector sequence S_i. Moreover, we maintain each subvector S_i to have size exactly l except possibly the very last non-empty vector. This allows us to simplify the access code so that searching for an element of rank r simply involves computing the index $i \leftarrow \lceil r/l \rceil$ of the vector containing the search element and then computing $r - il$ as the rank in that vector to search for. Moreover, we maintain the number of possible bottom-level vectors in $\mathcal{S}$ to be a power of two, so that we may use a bit shifting and masking instead of division to determine which subvector S_i holds the rank r element. By storing the shift and bit mask values we can reduce the number of operations required to retrieve an element from a tiered vector to only two, thus holding access time to only twice that of normal array-based vector retrieval. These modifications have negligable effects on asymptotic running times, but they nevertheless proved useful in practice.

Subvector Size Test. The choice of size for subvectors in a 2-level tiered vector has significant impact on its performance. The following test demonstrates the optimal subvector size for the tiered vector. Initially we start with a subvector size of ten and for each successive test we increase the subvector size by ten up to ten thousand. For each test we preinsert ten thousand elements into the tiered vector and then time how long it takes to insert one hundred elements at the head of this vector. Thus for the first tests the majority of the time for insertion is spent in cascade operations whereas for the final tests the majority of the time is spent in recursive shift operations in subvectors. Each test is run ten times and the resulting time represents the average of these tests.

Theoretically, the optimal subvector size should be near 100; however, the perfomance graph of Figure 4 shows the actual optimal size is near 750. The likely reason for these results is that the cascade operations are computationally more expensive than the recursive shifting operations.

Access Test. The cost of performing an access in a tiered vector should clearly be higher than that of a simple vector, but a natural question to ask is how much worse is the 2-level tiered vector than a standard vector. The following test demonstrates the time taken to retrieve the first one hundred elements from a tiered vector, a Java Vector, and a JGL Array. In each successive test a set number of elements is preinserted into each vector, starting at one hundred elements and increasing in number each successive test by one hundred element increments up to ten thousand. We then test to see how much time it takes to retrieve the first one hundred elements from each vector. Each test is run one hundred times and the resulting times represent the averages of these tests. The

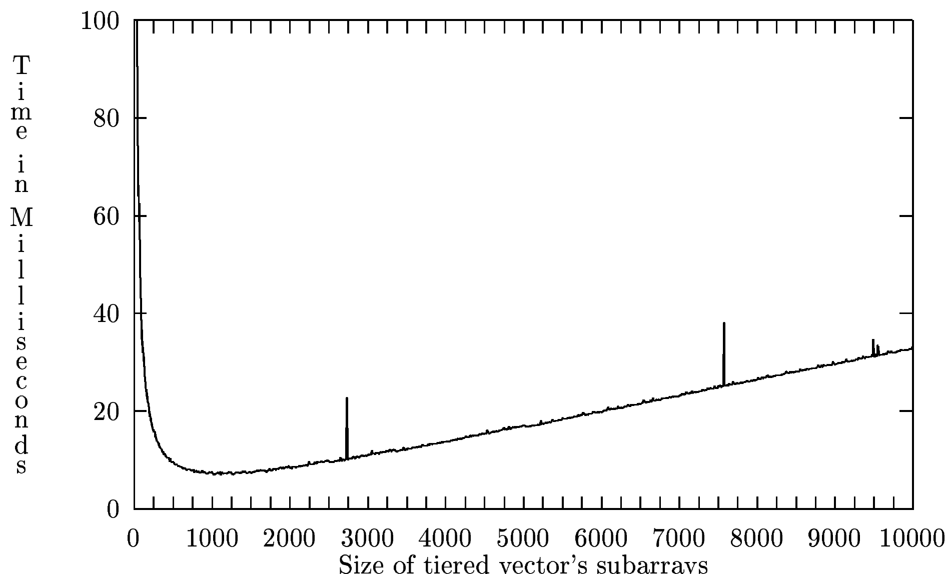

Fig. 4. Results of the subvector size experiment. Note that small sizes are very inefficient, but this inefficiency falls fast as the subarray size grows, it reaches an optimal value at 750 and then slowly rises after that.

choice of the first one hundred elements for retrieval was arbitrary. The results are shown in Figure 5.

Insertion Test. The claimed performance gain for tiered vectors is in the running times for element insertions and deletions. The following test demonstrates the time taken to insert one hundred elements at the head of a tiered vector, a Java Vector, and a JGL Array. The testing procedures are the same as the access test above. The choice of inserting at the head of each vector was to demonstrate worst case behavior in each.

Regarding the odd, step like behavior of the tiered vector, we note that sudden drops in insertion time occur when the vector initially contains near 64, 256, 1024, and 4098 elements. At these points the tiered vector is full and forced to expand, increasing it's subvector size by a factor of four. This expansion therefore reduces the initial number of cascade operations required for new insertions by a like factor of four. However, as the number of elements in the tiered vector increases the number of cascade operations increase linearly until the next forced expansion. The full results are shown in Figure 6.

Deletion Test. The following test demonstrates the time taken to remove one hundred elements from the head of a tiered vector, a Java Vector, and a JGL Array. The testing procedures are the same as the access test. The choice of deleting at the head of each vector is to demonstrate worst case behavior in each. The step like behavior of the tiered vector represents points of contraction, similar to the behavior in the insert tests. After a contraction the number of cas-

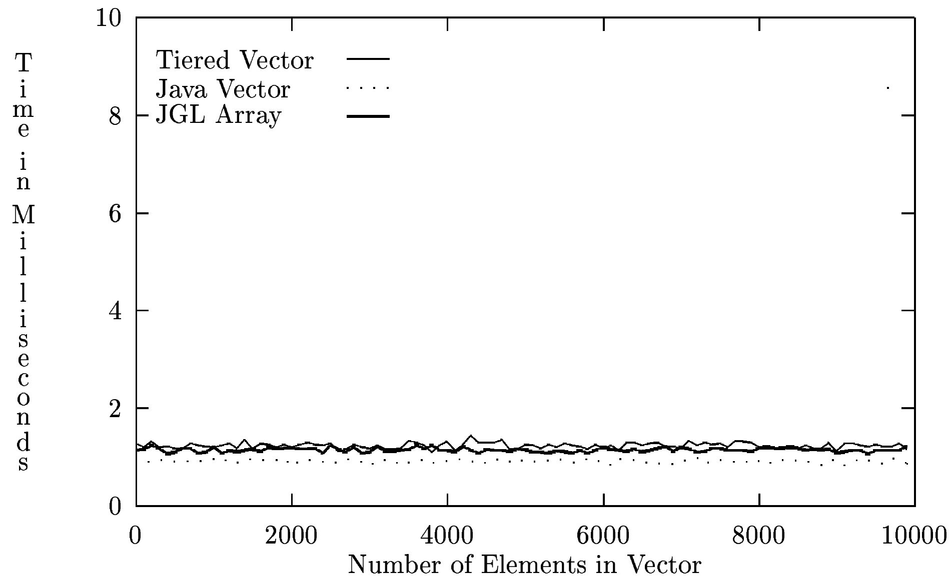

Fig. 5. Access times in tiered vectors and standard vectors. Note that the access times for tiered vectors are comparable with those for JGL arrays and only slightly worse than those for Java Vectors.

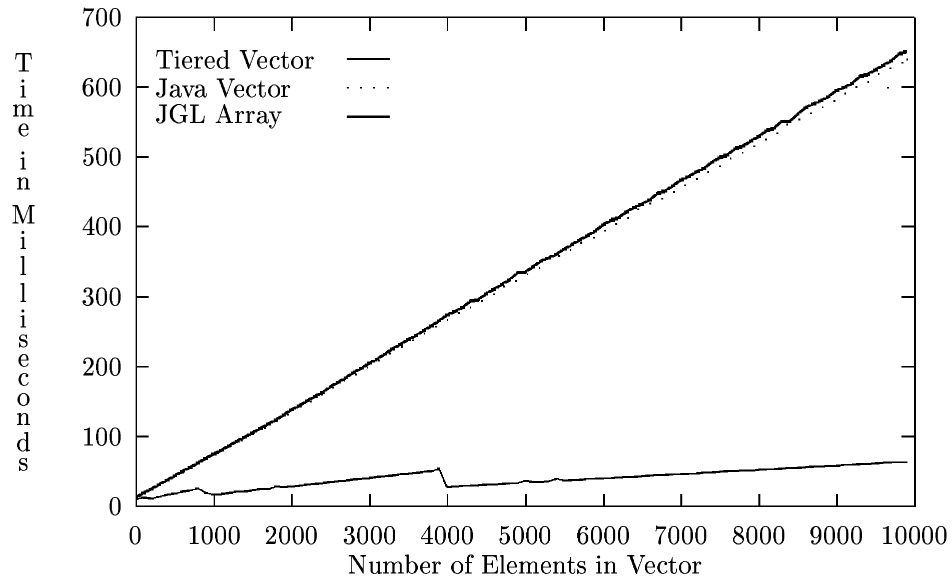

Fig. 6. The results for element insertions. The running times for standard vectors grow linearly, as expected, while those for tiered vectors are significantly faster.

cade operations required for deletion increases by a factor of four and gradually decreases as more elements are removed. The full results are shown in Figure 7.

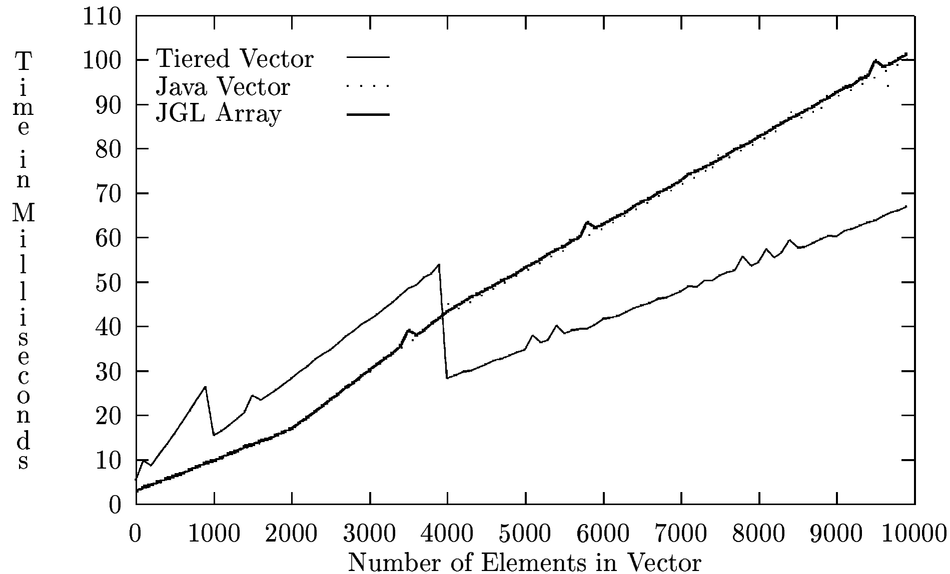

Fig. 7. The running times for deletion. The performance of tiered vectors is slightly inferior to standard vectors for small-sized lists, but is consistently superior for lists of more than 4096 elements.

Random Test. The following test demonstrates the time taken to insert one hundred elements randomly into a tiered vector, Java Vector, and JGL Array. The testing procedures are similar to the access test. During testing the vectors received the same set of random numbers to insert, though a different set of random numbers was generated for each test. Random numbers ranged from zero to the number of elements contained in the vector prior to testing. The results are given in Figure 8.

Acknowledgements

We would like to thank Rao Kosaraju and Roberto Tamassia for several helpful comments regarding the topics of this paper. The work of this paper was partially supported by the National Science Foundation under grant CCR-9732300.

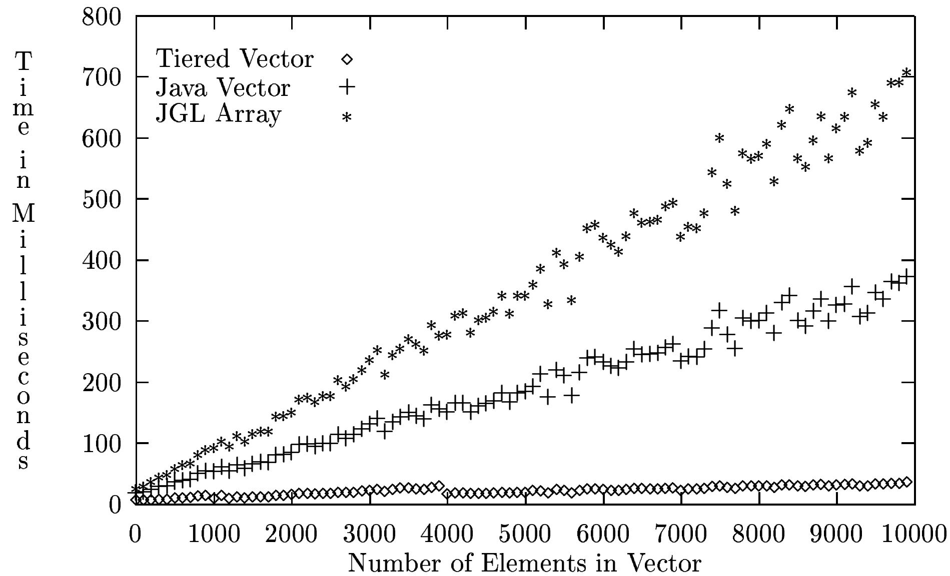

Fig. 8. Performance for random insertions. In this case the Java vector is superior to JGL's arrays, but the tiered vector is significantly faster than both.

References

1. P. Åke Larson. Dynamic hash tables. *Communications of the ACM*, 31(4), April 1988.
2. R. Bayer and K. Unterauer. Prefix B–Trees. *ACM Transactions on Database Systems*, 2(1):11–26, March 1977.
3. J. Boyer. Algorithm allery: Resizable data structures. *Dr. Dobb's Journal*, 23(1):115–116,118,129, January 1998.
4. A. Fraenkel, E. Reingold, and P. Saxena. Efficient management of dynamic tables. *Information Processing Letters*, 50:25–30, 1994.
5. M. L. Fredman and M. E. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, pages 345–354, Seattle, Washington, 15–17May 1989.
6. M. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java.* John Wiley & Sons, 1998.
7. D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison–Wesley, 3 edition, 1997.
8. D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison–Wesley, 3 edition, 1998.
9. E. Sitarski. Algorithm alley: HATs: Hashed array trees. *Dr. Dobb's Journal*, 21(11), September 1996.
10. H. Wedekind. On the selection of access paths in a database system. In *Proceedings of the IFIP Working Conference on Data Base Management.* North–Holland Publishing Company, 1974.