Tiered Vector:
An Efficient Dynamic Array for JDSL

Michael T. Goodrich
Johns Hopkins University
goodrich@cs.jhu.edu

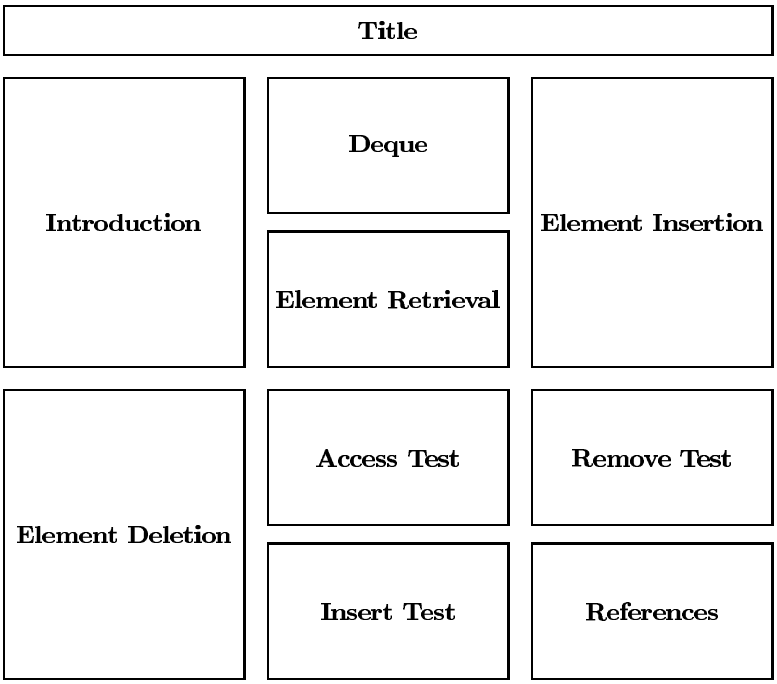
John G. Kloss II
Johns Hopkins University
jkloss@cs.jhu.edu

August 27, 1998

Abstract

We demonstrate the Tiered Vector, an implementation of the Vector ADT that provides O(1) worst case for rank based retrieval and O($\sqrt{n}$) amortized time for insertion and deletion. We also provide results from experiments involving the use of the Tiered Vector in JDSL, the Data Structures Library in Java.

Keywords: JDSL, Data Structure, Vector, Dynamic Array



1 Introduction

A Vector is a dynamic sequential list of elements that can expand or contract in size. When the number of elements in a Vector becomes larger or much smaller than the memory allocated, new memory is allocated (or reallocated) to accomodate the change in size and elements are reassigned to this memory. Each element e in a Vector is assigned an index or *rank*, which indicates the number of elements in front of e in the Vector. Rank can also be viewed as a current "address" for the element e. Each rank is assigned sequentially– that is, there are no gaps in the rank ordering. However an element may be inserted or deleted at any existing rank r. Such an operation forces all elements of rank $r+1, \ldots, n$ to be shifted either left or right, respectively.

In a standard implementation of the Vector Abstract Data Type (ADT) we would 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 $S[r]$. This is clearly constant time. However, both insertion and deletion may take $O(n)$ time with the worst case occurring when an element has rank 0 since either operation forces all elements at ranks $1, \ldots, n$ to be shifted.

1.1 Vector Abstract Data Type

The Vector ADT supports the following operations:

***insertElemAtRank(r,e)*:** Inserts an element e into the Vector at rank r.

***removeElemAtRank(r)*:** Removes the element stored at rank r and returns it.

***elemAtRank(r)*:** Retrieves the element e at rank r.

A Vector should guarantee constant time performance for the *elemAtRank(r)* operation, where n is the number of elements held in the Vector.

The above methods are defined, for example in the RankedSequence interface as part of the Data Structures Library JDSL. This interface allows Vectors (and other ranked lists) to be easily used as components for

- M-ary Trees. Because Vectors dynamically resize and have constant time access to elements they are useful as tree nodes in M-ary Trees.
- Sorting routines. For sorting routines that cannot be done in place a Vector can be used as backing store for the routine, resizing to the size of the sorting array.
- Database tables. In many database routines, the size of the database tables cannot be determined during creation. A Vector provides an easy method of storing dynamically expanding tables.

1.2 Relationships to Previous Work

A standard implementation of the Vector ADT is provided in the Java class libraries. It uses a bit copy routine to shift elements in its underlying array. This implementation provides the standard $O(n)$ time bounds for insertion and deletion. Java also provides a **capacityIncrement** parameter which allows the Vector to double in size or grow by a fixed k amount upon every expansion. This specified capacity use should be done with caution, however, for the fixed method of expansion is know to be $O(n^2)$ over n operations [7].

Our approach to implementing the Vector ADT is to use a 2-level array we call the "Tiered Vector". Several hashing implementations 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 [8] 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 also uses a s^k fixed size directory-segement scheme which he terms Hashed Array Trees [9]. Sitarski was primarily interested in providing an efficient implementation for appending elements to an array. He does not, however, provide an efficient method for insertion or deletion into the array ($O(n)$ time bounds for both).

The Tiered Vector may also be visualized as a highly compressed B-Tree [1] [6] and the searching mechanism is somewhat similar. In a B-Tree each node v maintains up to d keys and $d+1$ pointers, each pointer lying between key pairs and referencing a child of v. Searching the tree takes place by comparing a search key k with the d keys in a node until we find a key pair (d_i, d_{i+1}) such that $d_i \leq k \leq d_{i+1}$ and following the link which lies between the key pair. In the Tiered Vector there is only one internal node n with exactly l keys, each key paired with a link to an exteranl node with exactly l keys. Search comparisons are based upon higher-bit equality of keys. That is, if a search key k has the same higher order bits as a key l_i in n then we follow that link. Furthermore, the keys in n do not necessarily represent actual keys in the external nodes. They instead act as "indices" and in this sense are related to the internal node keys of the B^+-Tree as documented by Bayer and Wedekind [2] [10]. The main differences between a Tiered Vector and a B-Tree or B^+-Tree, then, is that there is only one internal node whose size is not a fixed constant. Furthermore, the sizes of external nodes in our case are kept very similar, so as to allow fast access.

1.3 Our Results

We present a simple variation on the Vector, termed the Tiered Vector. This data structure provides constant time bounds for the *elemAtRank(r)* retrieval, yet requires only $O(\sqrt{n})$ time for *insertElemAtRank(r,e)* and *removeElemAtRank(r)*. Furthermore, though expansion and contractionof the Tiered Vector is linear in time, the amortized time over a series of n insert and delete operations remains $O(\sqrt{n})$.

2 Indexable Circular Deque

The major component of the Tiered Vector is a set $\mathcal{S}$ of indexable circular deques. The deque is described by Knuth [5] as a linear list which provides constant time insert and delete operations at either the head or tail of this list. A circular deque S is a list which is held in a sequential section of memory of fixed size l. S maintains a pointer h, which references the index in memory of the head of this list as well as a pointer t, which references the tail. These values are decremented or incremented in order to insert or delete elements from the head or tail of S. We will let $|S|$ denote the number of elements in S. A circular deque is considered full when $|S| = l$.

An indexable circular deque assigns a *rank* to each element within the list, where the head element has rank 0, its proceeding element rank 1, etc. An element's rank in S does not necessarily correspond to its index in memory. For example, consider the deque S in Figure 1a. The rank 0 element **2**, which is pointed to

by h, is located at $S[3]$ whereas the rank 1 element **3** is at $S[0]$. To access any element of rank r, $0 \leq r < l$, in S we calculate the translation value $r' \leftarrow (h + r) \bmod l$ and retrieve the desired element at location $S[r']$. For our purposes, the size of each deque in $\mathcal{S}$ is equal to 2^k where k is an integer value. This choice of size allows us to use a method similar to Sitarski's [9] in that modulus operations may be performed using a bit mask of k lower-order bits.

Insertion into the head of a circular deque is easily performed in constant time by decrementing h and inserting the element at $S[h]$. If decrementing h results in $h < 0$ we set $h \leftarrow (l - 1)$. Code segment 1a, *insertFirst(S,e)*, demonstrates how this operation may be coded. An example of *insertFirst(S,e)* is illustrated in Figure 1b where h is shifted from index 0 to index 3 via the insertion of **0** at rank 0 in S. A similar procedure *removeFirst(S)* (used below) is an analogue of the code in 1a.

Constant time insertion and deletion at the tail of a circular deque is obtained through the use of a tail pointer. The method is analogous to that of insertion and deletion at the head. The method, *removeLast* which demostrates this operation is shown in code section 1b. The opposite operation, *insertLast* is nearly identical.

procedure *insertFirst(S,e)*
  $h' \leftarrow (h - 1) \bmod l$
  $S[h'] \leftarrow e$
  $h \leftarrow h'$

Code 1a: insertFirst

procedure *removeLast(S)*
  $e \leftarrow S[t]$
  $t \leftarrow (t - 1) \bmod l$
  return e

Code 1b: removeLast

3 Tiered Vector

The Tiered Vector is a set of l indexable circular deques, $\mathcal{S} = \{S_1, S_2, \ldots, S_l\}$. As was mentioned above, each deque is of size l where $l = 2^k$ for some integer parameter k. Thus the total number of elements a Tiered Vector may hold before it must be expanded is l^2. Given n elements, $\mathcal{S}$ partitions these elements into $\lceil n/l \rceil$ sets where the deque S_1 contains the elements of rank $0, 1, \ldots, l-1$, S_2 contains $l, l+1, \ldots, 2l-1$, etc. All remaining sets $S_{\lceil n/l \rceil + 1}, \ldots, S_l$ are empty.

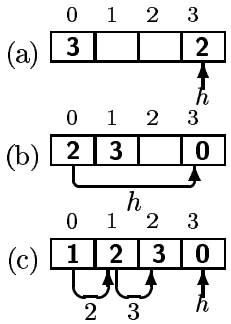


Fig 1. Circular deque S of size 4

3.1 Element Retrieval

Element retrieval in a Tiered Vector is very similar to methods proposed by Larson [8] and Sitarski [9]. To access any element of rank r in the Vector we first determine in which deque the element is located by calculating $i \leftarrow \lfloor r/l \rfloor$. We then calculate its location in deque S_i via the translation r' as mentioned above and retrieve the desired element at location $S_i[r']$. Since the number of deques in $\mathcal{S}$ is $l = 2^k$ we may use a bit shift instead of division to determine which deque 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.

3.2 Element Insertion

Insertion into a Tiered Vector is composed of two phases– *Pop-Push* and *Shift*. The purpose of the two phase process is to reduce the number of operations to $O(l)$ (which we prove to be equal to $O(\sqrt{n})$ below), as opposed to the $O(n)$ operations required by normal Vector insertion.

In the *Pop-Push* phase we first determine the deques 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 deques S_{sub} and S_{end}. These deques are used as the bounds for a series of pair-wise *pop-push* operations. For each deque S_i, $top \leq i < end$, we will pop its tail item and push it onto the head of deque S_{i+1}. As was demonstrated above, each such operation takes only constant time. Since there are a total of l deques this phase requires a maximum of $O(l)$ operations.

In the *Shift* phase we then shift the elements of S_{sub} to the left or right inorder to make room for the new element at rank r. We are assured that such space exists by the *Pop-Push* phase. Since each deque holds a maximum of l elements this operation takes $O(l)$ time. Figure 1c demonstrates this operation where the element **1** is inserted at rank 1 forcing the elements **2** and **3** to be shifted to the right. The implementation of insertion is demonstrated in code segment 2b. Figure 2 demonstrates the insertion operation where element **1** is inserted at rank 1.

A special case occurs when the number of elements in the Tiered Vector, n equals the maximum space provided, l^2. In this case the data structure must be expanded inorder to accomodate new elements. However, we also wish to preserve the structure of the Tiered Vector in order to insure only $O(l)$ operations are performed. We achieve this by first reseting the fixed length l to $l' \leftarrow 2l$ and then creating a new set of l' subarrays, $\mathcal{S}'$ where the first $\frac{1}{4}l'$ elements of $\mathcal{S}'$ are the result of a pairwise merge and rank ordering of the subarrays in $\mathcal{S}$. Code segment 3 demonstrates this procedure.

```
procedure expand( )
  l' ← 2l
  S' ← new set of arrays of size l' where each
       array S'_i in S' is of size l'
  foreach S'_i in S'
    realign rank order in S_2i to match indices
    realign rank order in S_2i+1 to match indices
    S'_i ← S_2i ∪ S_2i+1
    h'_i ← i * l
  S ← S'
  l ← l'
```

Code 3: expand

Theorem 1 *Insertion into a Tiered Vector where expansion is not required takes $O(\sqrt{n})$ time.*

Expansion is demonstrated in Figure 3, where a Tiered Vector of fixed subarray size 4 is expanded into a Vector of subarray size 8. From the *Expand* operation we see that n always lies between $\frac{1}{4}l^2$ and l^2. Since insertion takes $O(l)$ operations and n is $O(l^2)$, this implies that insertion takes $O(\sqrt{n})$ time.

Insertion requiring an *expand* operation, however, takes $O(n)$ time since we are forced to reorder n elements in the process of creating $\mathcal{S}$. However, we can show that the amortized time for insert is $O(\sqrt{n})$ by using the *accounting method* as described by Goodrich and Tamasia [7].

Theorem 2 *Insertion into a Tiered Vector takes amortized time $O(\sqrt{n})$.*

We shall assign each reorder operation a debit of 1 for each element so reordered. Thus the total cost of a reorder operation is a debit of n. For each insert operation we assign a credit of 2. Since we only perform

an *Expand* after we have made $\frac{3}{4}$ inserts it is apparent that after O(n) operations we have completely paid for the cost of the expansion.

```
procedure InsertElemAtRank(r,e)
  if r > number of elements or r < 0 then
    error "Index Out of Bounds"
  if number of elements > max space
    Expand
  sub ← ⌊r/l⌋
  end ← ⌊n/l⌋
  if sub < end then
    head ← removeLast(S_sub)
    tail ← null
    i ← sub + 1
    foreach S_i, i < end
      tail ← removeLast(S_i)
      insertFirst(S_i,head)
      head ← tail
      i ← i + 1
    insertFirst(S_end,head)
  r' ← (h_sub + r) mod l
  if h_sub = 0 or r' < h_sub then
    Slide all elements in S_sub of rank
    greater than or equal to r' and less
    than (|S_sub| − r') mod l to the right
    by one
  else
    Slide all elements in S_sub of rank less
    than r' and greater than or equal to
    h_sub to the left by one
  S_sub[r'] ← e
```

Code 2a: insertElemAtRank

```
procedure RemoveElemAtRank(r)
  if r ≥ number of elements or r < 0then
    error "Index Out of Bounds"
  if number of elements < 1/8 max space
    Contract
  sub ← ⌈r/l⌉
  end ← ⌈n/l⌉
  r' ← (h_sub + r) mod l
  ret ← S_sub[r']
  head ← null
  tail ← null
  if top < end then
    i ← end
    foreach S_i, i > top
      head ← removeFirst(S_i)
      insertLast(S_i,tail)
      tail ← head
      i ← i − 1
  if h_sub = 0 or r' < h_sub then
    Slide all elements in S_sub of rank r'+1
    to h_sub + |S_sub| to the left
  else
    Slide all elements in S_sub of rank
    greater than or equal to h_sub and less
    than r' to the right by one
    h_sub ← h_sub + 1
  return ret
```

Code 2b: removeElemAtRank

3.3 Element Deletion

Deletion is simply the reverse of insertion and uses a similar *Pop-Push* and *Shift* process. Again, we first determine in which subarrays the elements at rank r and rank $n-1$ are located and term these subarrays S_{sub} and S_{end}. Then for each pair of subarrays, S_i and S_{i+1}, $sub \leq i \leq end$, we will pop the head of S_{i+1} and push it onto the tail of S_i. Since this process is simply the reverse of insert's *Pop-Push* phase, we are guaranteed a maximum of O(l) operations.

During the second phase we again perform the opposite of insert's *Shift* phase. After popping the rank r element from S_{sub} we shift a maximum of $l-1$ elements to the left or right to close the space vacated by the removed element. Since S_{sub} contains at

```
procedure compress( )
  l' ← 1/2 l
  S' ← new set of arrays of size l' where each
       array S'_i in S' is of size l'
  foreach S_i in S
    realign rank order in S_i to match indices
    S'_{2i} ← ranks 0, 1, ..., 1/2(l − 1) in S_i
    S'_{2i+1} ← ranks 1/2 l, 1/2(l + 1), ..., l in S_i
    h'_{2i} ← 0
    h'_{2i+1} ← 0
  S ← S'
  l ← l'
```

Code 4: compress

most l elements, this operation takes $O(l)$ time. The implementation of delete is in code segment 2b.

A special case of delete occurs when the number of elements remaining in the Tiered Vector equals $\frac{1}{8}l^2$. At this point we must reduce the size of the Tiered Vector inorder to preserve the $O(\sqrt{n})$ time bounds for both insert and delete. We first reset the fixed length l to $l' \leftarrow \frac{1}{2}l$ and then create a new set of l' subarrays, $\mathcal{S}'$ where the first $\frac{1}{2}l'$ elements of $\mathcal{S}'$ are the result of a rank reordering and splitting of each full subarray in $\mathcal{S}$. This procedure in demonstrated in code segment 4.

It should be noted that we don't call compress when $|\mathcal{S}|$ equals $\frac{1}{4}l$. Instead we follow a method suggested by Boyer [3] so that after a compression $\frac{1}{2}l^2$ inserts must be made before an expansion is required. If, instead we had chosen to compress at $|\mathcal{S}|$ equal to $\frac{1}{4}$ the new Tiered Vector would have all its slots filled and a single insert operation would cause another expansion.

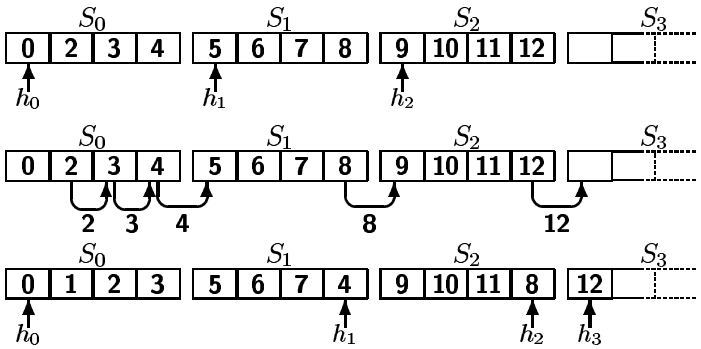


Fig 2. Insertion of element **1** at rank 1

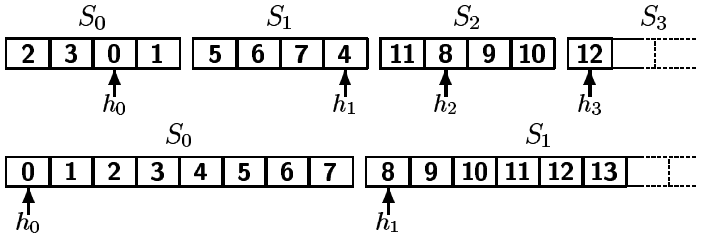
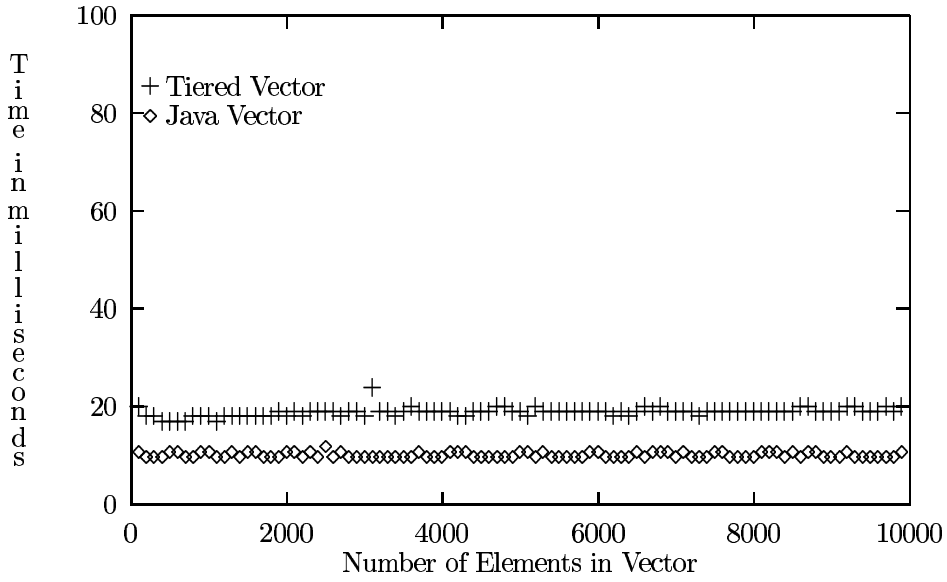


Fig 3. Expansion and reordering of a Tiered Vector after a call to *Expand*

4 Tests

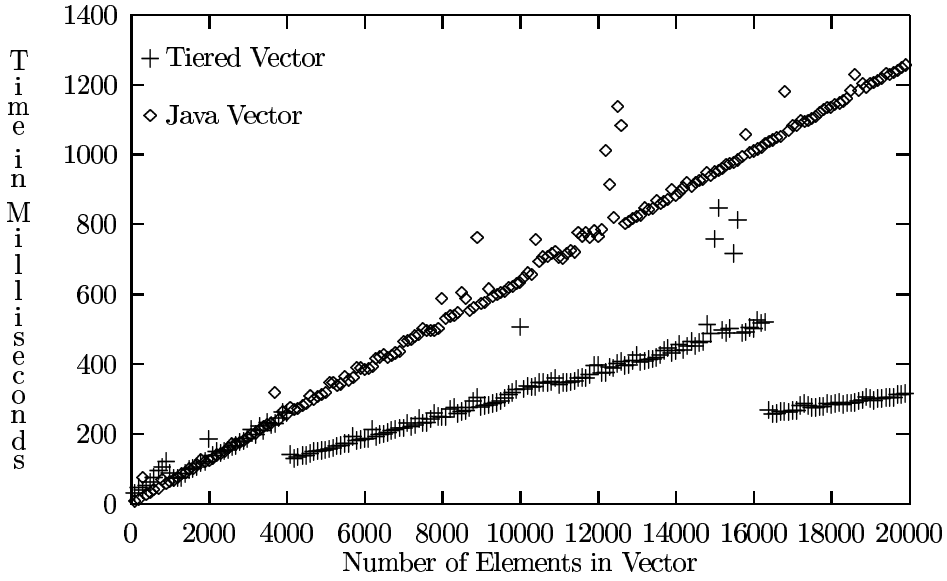
4.1 Access Test

In the test below one hundred random elements where accessed from both the Java Vector and the Tiered Vector.



4.2 Insertion Test

In the test below one hundred elements where inserted at the head (rank 0) of both a Java Vector and the Tiered Vector. "Number of Elements" indicates the number of elements contained in either Vector prior to insertion.



4.3 Deletion Test

To be submitted later.

5 Acknowledgements

We would like to thank Rao Kosaraju and Roberto Tamassia for several helpful comments regarding the topics of this paper.

References

[1] R. Bayer and C. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Inf.* v.1, 3(1972), 173-189.

[2] R. Bayer and K. Unterauer. Prefix B-Trees. *ACM Trans. Database Syst.* 2, 1(March 1977), 11-26.

[3] John Boyer. Algorithm Alley: Resizable Data Structures. *Dr. Dobb's Journal*, 23(1), p.115-116,118,129, January 1998.

[4] Aviezri Fraenkel, Edward Reinglod, and Prashant Saxena. Efficient management of dynamic tables. *Information Processing Letters.* v.50, 25-30, 1994.

[5] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Third edition, 1997.

[6] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Third edition, 1998.

[7] Michael Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java*. John Wiley & Sons, 1998.

[8] Per-Åke Larson. Dynamic Hash Tables. *Communications of the ACM*, 31(4), April 1988.

[9] Edward Sitarski. Algorithm Alley: HATs: Hashed Array Trees. *Dr. Dobb's Jounral*, 21(11), September 1996.

[10] H. Wedekind. On the Selection of Access Paths in a Database System. *Proc. IFIP Working Conf. Data Base Management.* North-Holland Publishing Co., New York, 1974.