Fast Dynamic Arrays

Philip Bille[1], Anders Roy Christiansen[2], Mikko Berggren Ettienne[3], and Inge Li Gørtz[4]

1   **The Technical University of Denmark, Lyngby, Denmark**
    `phbi@dtu.dk`
2   **The Technical University of Denmark, Lyngby, Denmark**
    `aroy@dtu.dk`
3   **The Technical University of Denmark, Lyngby, Denmark**
    `miet@dtu.dk`
4   **The Technical University of Denmark, Lyngby, Denmark**
    `inge@dtu.dk`

───── **Abstract** ─────

We present a highly optimized implementation of tiered vectors, a data structure for maintaining a sequence of n elements supporting access in time $O(1)$ and insertion and deletion in time $O(n^\epsilon)$ for $\epsilon > 0$ while using $o(n)$ extra space. We consider several different implementation optimizations in C++ and compare their performance to that of *vector* and *set* from the standard library on sequences with up to 10^8 elements. Our fastest implementation uses much less space than set while providing speedups of $40\times$ for access operations compared to set and speedups of $10.000\times$ compared to vector for insertion and deletion operations while being competitive with both data structures for all other operations.

1   Introduction

We present a highly optimized implementation of a data structure solving the *dynamic array problem*, that is, maintain a sequence of elements subject to the following operations:

access(i): return the ith element in the sequence.

access(i, m): return the ith through $(i + m - 1)^{th}$ elements in the sequence.

insert(i, x): insert element x immediately after the ith element.

delete(i): remove the ith element from the sequence.

update(i, x): exchange the ith element with x.

This is a fundamental and well studied data structure problem [2, 4, 7, 8, 3, 1, 5, 6] solved by textbook data structures like arrays and binary trees. Many dynamic trees provide all the operations in $O(\lg n)$ time including 2-3-4 trees, AVL trees, splay trees, etc. while Dietz [2] gives a data structure that matches the lower bound of $\Omega(\lg n / \lg \lg n)$ showed by Fredman and Saks [4]. In this paper however, we focus on the problem where access must run in $O(1)$ time. Goodrich and Kloss present what they call *tiered vectors* [5] with a time complexity of $O(1)$ for access and update and $O(n^{1/l})$ for insert and delete for any constant integer $l \geq 2$, similar to the ideas presented by Frederickson in [3]. The data structure only uses $o(n)$ extra space beyond that required to store the actual elements. At the core, the data structure is a tree with out degree $n^{1/l}$ and *constant* height $l - 1$.

Goodrich and Kloss compare the performance of an implementation with $l = 2$ to that of *vector* from the standard library of Java and show that the structure is competitive for access operations while being significantly faster for insertions and deletions. Tiered vectors provide a performance trade-off between standard arrays and balanced binary trees for the dynamic array problem.

Our Contribution.   In this paper, we present what we believe is the first implementation of tiered vectors that supports more than 2 tiers. Our C++ implementation supports access and update in times that are competitive with the vector class from C++'s standard library while insert and delete run more than $10.000\times$ faster. It performs access and update more than $40\times$ faster than the set class from the standard library while insert and delete is only a few percent slower. Furthermore set uses more than $10\times$ more space than our implementation. All of this when working on large sequences of 10^8 32-bit integers.

To obtain these results, we significantly decrease the number of memory probes of the original tiered vector. Our best variant requires only half as many memory probes as the original tiered vector for access and update operations which is critical for the practical performance. Our implementation is cache efficient which makes all operations run fast in practice even on tiered vectors with several tiers.

We experimentally compare the different variants of tiered vectors. Besides the comparison to the two commonly used C++ data structures, vector and set, we compare the different variants of tiered vectors to find the best one. We show that the number of tiers have a significant impact on the performance which underlines the importance of tiered vectors supporting more than 2 tiers.

Our implementations are parameterized and thus support any number of tiers ≥ 2. It uses a number of tricks like *template recursion* to keep the code rather simple while enabling the compiler to generate highly optimized code.

2   Preliminaries

The first and i^{th} element of a sequence A are denoted $A[0]$ and $A[i-1]$ respectively and the ith through j^{th} elements are denoted $A[i-1, j-1]$. Let $A_1 \cdot A_2$ denote the concatenation of the sequences A_1 and A_2. $|A|$ denotes the number of elements in the sequence A. A circular shift of a sequence A by x is the sequence $A[|A|-x, |A|-1] \cdot A[0, |A|-x-1]$. Define the remainder of division of a by b as $a \mod b = a - qb$ where q is the largest integer such that $q \cdot b \leq a$. Define $A[i, j] \mod w$ to be the elements $A[i \mod w], A[(i+1) \mod w], \ldots, A[j \mod w]$, i.e. $A[4, 7] \mod 5 = A[4], A[0], A[1], A[2]$. Let $\lfloor x \rfloor$ denote the largest integer smaller than x.

3   Tiered Vectors

In this section we will describe how the tiered vector data structure from [5] works.

Data Structure.   An l-tiered vector can be seen as a tree T with root r, fixed height $l-1$ and out-degree w for any $l \geq 2$. A node $v \in T$ represents a sequence of elements $A(v)$ where $A(r)$ is the sequence represented by the tiered vector. The capacity $\text{cap}(v)$ of a node v is $w^{\text{height}(v)+1}$. For a node v with children $c_1, c_2, \ldots, c_w$, $A(v)$ is a circular shift of the concatenation of the elements represented by its children, $A(c_1) \cdot A(c_2) \cdot \ldots \cdot A(c_w)$. The circular shift is determined by an integer $\text{off}(v) \in [\text{cap}(v)]$ that is explicitly stored for all
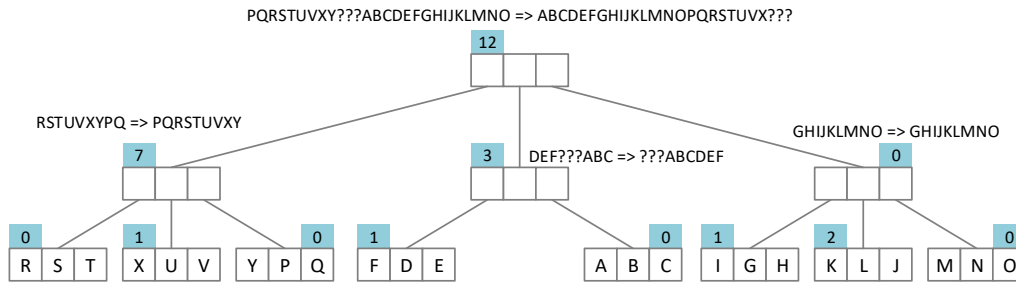
Figure 1 An illustration of a tiered vector with $l = w = 3$. The elements are letters, and the tiered vector represents the sequence ABCDEFGHIJKLMNOPQRSTUVX. The elements in the leaves are the elements that are actually stored. The number above each node is its offset. The strings above an internal node v with children c_1, c_2, c_3 is respectively $A(c_1) \cdot A(c_2) \cdot A(c_3)$ and $A(v)$, i.e. the elements v represents before and after the circular shift. ? specifies an empty element.

nodes. Thus the sequence of elements $A(v)$ of an internal node v can be reconstructed by recursively reconstructing the sequence for each of its children, concatenating these and then circular shifting the sequence by $\mathrm{off}(v)$. See Figure 1 for an illustration. A leaf v of T explicitly stores the sequence $A(v)$ in a circular array $\mathrm{elems}(v)$ with size w whereas internal nodes only store their offsets. Call a node v full if $|A(v)| = \mathrm{cap}(v)$ and empty if $|A(v)| = 0$. In order to support fast access, for all nodes v the elements of $A(v)$ are located in consecutive children of v that are all full, except the children containing the first and last element of $A(v)$ which may be only partly full.

Access & Update. To access an element $A(r)[i]$ at a given index i; one traverses a path from the root down to a leaf in the tree. In each node the offset of the node is added to the index to compensate for the cyclic shift, and the traversing is continued in the child corresponding to the newly calculated index. Finally the desired element is returned from the elements array of that leaf. Let $\mathsf{access}(v, i)$ return the element $A(v)[i]$, it can recursively be computed as:

v is internal: Compute $i' = (i + \mathrm{off}(v)) \mod \mathrm{cap}(v)$, let v' be the $\lfloor i'/w \rfloor^{th}$ child of v and return the element $\mathsf{access}(v', i' \mod \mathrm{cap}(v'))$.

v is leaf: Compute $i' = (i + \mathrm{off}(v)) \mod w$ and return the element $\mathrm{elems}(v)[i']$.

The time complexity is $\Theta(l)$ as we visit all nodes on a root-to-leaf path in T. To navigate this path we must follow $l - 1$ child pointers, lookup l offsets, and access the element itself. Therefore this requires $l - 1 + l + 1 = 2l$ memory probes.

The update operation is entirely similar to access, except the element found is not returned but substituted with the new element. The running time is therefore $\Theta(l)$ as well. For further use, let $\mathsf{update}(v, i, e)$ be the operation that sets $A(v)[i] = e$ and returns the element that was substituted.

Range Access. Accessing a range of elements, can obviously be done by using the access-operation multiple times, but this results in redundant traversing of the tree, since consecutive elements of a leaf often – but not always due to circular shifts – corresponds to consecutive elements of $A(r)$. Let $\mathsf{access}(v, i, m)$ report the elements $A(v)[i \ldots i + m - 1]$ in order. The operation can recursively be defined as:

v is internal: Let $i_l = (i + \mathrm{off}(v)) \mod \mathrm{cap}(v)$, and let $i_r = (i_l + m) \mod \mathrm{cap}(v)$. The children of v that contains the elements to be reported are in the range $[\lfloor i_l \cdot w / \mathrm{cap}(v) \rfloor, \lfloor i_r \cdot w / \mathrm{cap}(v) \rfloor] \mod w$, call these $c_l, c_{l+1}, \ldots, c_r$. In order, call $\mathsf{access}(c_l, i_l, \min(m, \mathrm{cap}(c_l) - i_l))$, $\mathsf{access}(c_i, 0, \mathrm{cap}(c_i))$ for $c_i = c_{l+1}, \ldots, c_{r-1}$, and $\mathsf{access}(c_r, e_{r-1}, 0, i_r \mod \mathrm{cap}(c_r))$.

v is leaf: Report the elements $\mathrm{elems}(v)[i, i + m - 1] \mod w$.

The running time of this strategy is $O(lm)$, but saves a constant factor over the naive solution.

Insert & Delete.   Inserting an element in the end (or beginning) of the array can simply be achieved using the update-operation. Thus the interesting part is fast insertion on an arbitrary position; this is where we utilize the offsets.

Consider a node v, the key challenge is to shift a big chunk of elements $A(v)[i, i + m - 1]$ one index right (or left) to $A(v)[i+1, i+m]$ to make room for a new element (without actually moving each element in the range). Look at the range of children $c_l, c_{l+1}, \ldots, c_r$ that covers the range of elements $A(v)[i, i + m - 1]$ to be shifted. All elements in $c_{l+1}, \ldots, c_{r-1}$ must be shifted. These children are guaranteed to be full, so make a circular shift by decrementing each of their offsets by one. Afterwards take the element $A(c_{i-1})[0]$ and move it to $A(c_i)[0]$ using the update operation for $l < i \leq r$. In c_l and c_r only a subrange of the elements might need shifting, which we do recursively. In the base case of this recursion, namely when v is a leaf, shift the elements by actually moving the elements one-by-one in $\mathrm{elems}(v)$.

Formally we define the $\mathsf{shift}(v, e, i, m)$ operation that (logically) shifts all elements $A(v)[i, i + m - 1]$ one place right to $A[i + 1, i + m]$, sets $A[i] = e$ and returns the value that was previously on position $A[i + m]$ as:

v is internal: Let $i_l = (i + \mathrm{off}(v)) \mod \mathrm{cap}(v)$, and let $i_r = (i_l + m) \mod \mathrm{cap}(v)$. The children of v that must be updated are in the range $[\lfloor i_l \cdot w / \mathrm{cap}(v) \rfloor, \lfloor i_r \cdot w / \mathrm{cap}(v) \rfloor] \mod w$ call these $c_l, c_{l+1}, \ldots, c_r$. Let $e_l = \mathsf{shift}(c_l, e, i_l, \min(m, \mathrm{cap}(c_l) - i_l))$. Let $e_i = \mathsf{update}(c_i, size(c) - 1, e_{i-1})$ and set $\mathrm{off}(c_i) = (\mathrm{off}(c_i) - 1) \mod \mathrm{cap}(c)$ for $c_i = c_{l+1}, \ldots, c_{r-1}$. Finally call $\mathsf{shift}(c_r, e_{r-1}, 0, i_r \mod \mathrm{cap}(c_r))$.

v is leaf: Let $e_o = \mathrm{elems}(v)[(i + m) \mod w]$. Move the elements $\mathrm{elems}(v)[i, (i + m - 1) \mod w]$ to $\mathrm{elems}(v)[i + 1, (i + m) \mod w]$, and set $\mathrm{elems}(v)[i] = e$. Return e_o.

An insertion $\mathsf{insert}(i, e)$ can then be performed as $\mathsf{shift}(root, e, i, size(root) - i - 1)$. The running time of an insertion is $T(l) = 2T(l - 1) + w \cdot l \Rightarrow T(l) = O(2^l w)$.

A deletion of an element can basically be done as an inverted insertion, thus deletion can be implemented using the shift-operation from before. A $\mathsf{delete}(i)$ can be performed as $\mathsf{shift}(r, \perp, 0, i)$ followed an update of the root's offset to $(\mathrm{off}(r) + 1) \mod \mathrm{cap}(r)$.

Space.   There are at most $O(w^{l-1})$ nodes in the tree and each takes up constant space, thus the total space of the tree is $O(w^{l-1})$. All leaves are either empty or full except the two leaves storing the first and last element of the sequence which might contain less than w elements. Because the arrays of empty leaves are not allocated the space overhead of the arrays is $O(w)$. Thus beyond the space required to store the n elements themselves, tiered vectors have a space overhead of $O(w^{l-1})$.

To obtain the desired bounds w is maintained such that $w = \Theta(n^\epsilon)$ where $\epsilon = 1/l$ and n is the number of elements in the tiered vector. This can be achieved by using global rebuilding to gradually increase/decrease the value of w when elements are inserted/deleted without asymptotically changing the running times. We will not provide the details here. We sum up the original tiered vector data structure in the following theorem:

▶ **Theorem 1** ([5])**.** *The original l-tiered vector solves the dynamic array problem for $l \geq 2$ using $\Theta(n^{1-1/l})$ extra space while supporting* access *and* update *in $\Theta(l)$ time and $2l$ memory probes. The operations* insert *and* delete *take $O(2^l n^{1/l})$ time.*

4 Improved Tiered Vectors

In this paper, we consider different new variants of the tiered vector. This section considers the theoretical properties of these approaches. In particular we are interested in the number of memory accesses that are required for the different memory layouts, since this turns out to have an effect on the experimental running time. In Section 5.1 we analyze the actual impact in practice through experiments.

4.1 Implicit Tiered Vectors

As the degree of all nodes is always fixed to the same value w (it may be changed for all nodes when the tree is rebuilt due to a full root), it is possible to layout the offsets and elements such that no pointers are necessary to navigate the tree. Simply number all nodes from left-to-right level-by-level starting in the root with number 0. Using this numbering scheme, we can store all offsets of the nodes in a single array and similarly all the elements of the leaves in another array.

To access an element, we only have to lookup the offset for each node on the root-to-leaf path which requires $l - 1$ memory probes plus the final element lookup, i.e. in total l which is half as many as the original tiered vector. The downside with this representation is that it must allocate the two arrays in their entirety from the beginning (or when rebuilding). This results in a $\Theta(n)$ space overhead which is worse than the $\Theta(n^{1-\epsilon})$ space overhead from the original tiered vector.

▶ **Theorem 2.** *The implicit l-tiered vector solves the dynamic array problem for $l \geq 2$ using $O(n)$ extra space while supporting* access *and* update *in $O(l)$ time requiring l memory probes. The operations* insert *and* delete *take $O(2^l n^{1/l})$ time.*

4.2 Lazy Tiered Vectors

We now combine the original and the implicit representation, to get both few memory probes and small space overhead. Instead of having one array storing all the elements of the leaves, we store for each leaf a pointer to a location with an array containing the leaf's elements. The array is lazily allocated in memory when elements are actually inserted into it.

The total size of the offset-array and the element pointers in the leaves is $O(n^{1-\epsilon})$. At most two leaves are only partially full, therefore the total space is now again reduced to $O(n^{1-\epsilon})$. To navigate a root-to-leaf path, we now need to look at $l - 1$ offsets, follow a pointer from a leaf to its array and access the element in the array, giving a total of $l + 1$ memory accesses.

▶ **Theorem 3.** *The lazy l-tiered vector solves the dynamic array problem for $l \geq 2$ using $\Theta(n^{1-1/l})$ extra space while supporting* access *and* update *in $\Theta(l)$ time requiring $l+1$ memory probes. The operations* insert *and* delete *take $O(2^l n^{1/l})$ time.*

5 Implementation

We have implemented a generic version of the tiered vector data structure such that the number of tiers and the size of each tier can be specified at compile time. To the best of our knowledge,

all prior implementations of the tiered vector are limited to the considerably simpler 2-tier version. Most of the performance optimizations applied in the 2-tier implementation do not easily generalize. We have implemented the following variants of tiered vectors:

- *Original.* The data structure described in Theorem 1.
- *Optimized Original.* As described in Theorem 1 but with the offset of a node v located in the parent of v, adjacent in memory to the pointer to v. Leaves only consists of an array of elements (since their parent store their offset) and the root's offset is maintained separately as there is no parent to store it in.
- *Implicit.* This is the data structure described in Theorem 2 where the tree is represented implicitly in an array storing the offsets and the elements of the leaves are located in a single array.
- *Packed Implicit.* This is the data structure described in Theorem 2 with the following optimization; The offsets stored in the offset array are are packed together and stored in as little space possible. The maximum offset of a node v in the tree is $n^{\epsilon(\mathrm{height}(v)+1)}$ and the number of bits needed to store all the offsets is therefore $\sum_{i=0}^{l} n^{1-i\epsilon} \log(n^{i\epsilon}) = \log(n) \sum_{i=0}^{l} i\epsilon n^{1-i\epsilon} \approx \epsilon n^{1-\epsilon} \log(n)$. Thus the $n^{1-\epsilon}$ offsets can be stored in approximately $\epsilon n^{1-\epsilon}$ words giving a space reduction of a constant factor ϵ. The smaller memory footprint could lead to better cache performance.
- *Lazy.* This is the data structure described in Theorem 3 where the tree is represented implicitly in an array storing the offsets and every leaf store a pointer to an array storing only the elements of that leaf.
- *Packed Lazy.* This is the data structure described in Theorem 3 with the following optimization; The offset and the pointer stored in a leaf is packed together and stored at the same memory location. On most modern 64-bit system – including the one we are testing on – a memory pointer is only allowed to address 48 bits. This means we have room to pack a 16 bit offset in the same memory location as the elements pointer, which results in one less memory probe during an access operation.
- *Non-Templated.* All other implementations used C++ templating for recursive functions in order to let the compiler do significant code optimizations. This implementation is template free and serves as a baseline to compare the performance gains given by templating.

In Section 7 we compare the performance of these implementations.

5.1   C++ Templates

As almost all other general purpose data structures in C++, we have used templates to support storing different types of data in our tiered vector. This is a well-known technique which we will not describe in detail.

However, we have also used *template recursion* which is basically like a normal recursion except that the recursion parameter must be a compile-time constant. This allows the compiler to unfold the recursion at compile-time eliminating all (recursive) function calls by inlining code, and allowing for better local code optimizations. In our case, we exploit that the height of a tiered vector is constant and therefore can be used for this.

To show the rather simple code resulting from this approach (disregarding the template stuff itself), we have included a snippet of the internals of our access operation:

```
template <class T, class Layer>
struct helper {
        static T& get(size_t node, size_t idx) {
            idx = (idx + get_offset(node)) % Layer::capacity;
            auto child = get_child(node, idx / Layer::child::capacity);
            return helper<T, typename Layer::child>::get(child, idx);
        }
}


template <class T, size_t W>
struct helper<T, Layer<W, LayerEnd> > {
        static T& get(size_t node, size_t idx) {
            idx = (idx + get_offset(node)) % L::capacity;
            return get_elem(node, idx);
        }
}
```

We also briefly show how to use the data structure. To specify the desired height of the tree, and the width of the nodes on each tier, we also use templating:

```
Tiered<int, Layer<8, Layer<16, Layer<32>>>> tiered;
```

This will define a tiered vector containing integers with three tiers. The height of the underlying tree is therefore 3 where the root has 8 children, each of which has 16 children each of which contains 32 elements. We call this configuration 8-16-32.

In this implementation of tiered vectors we have decided to let the number of children on each level be a fixed number as described above. This imposes a maximum on the number of elements that can be inserted. However, in a production ready implementation, it would be simple to make it grow-able by maintaining a single growth factor that should be multiplied on the number of children on each level. This can be combined with the templated solution since the growing is only on the number of children and not the height of the tree (per definition of tiered vectors the height is constant). This will obviously increase the running time for operations when growing/shrinking is required, but will only have minimal impact on all other operations (they will be slightly slower because computations now must take the growth factor into account).

In practice one could also, for many uses, simply pick the number of children on each level sufficiently large to ensure the number of elements that will be inserted is less than the maximum capacity. This would result in a memory overhead when the tiered vector is almost empty, but by choosing the right variant of tiered vectors and the right parameters this overhead would in many cases be insignificant.

6 Comparison with C++ STL Data Structures

In the following we have compared our best performing tiered vector (see next section) to the vector and the multiset class from the C++ standard library. The vector class directly supports the operations of a dynamic array. The multiset class is implemented as a red-black tree and is therefore interesting to compare with our data structure. Unfortunately, multiset does not directly support the operations of a dynamic array (in particular it has no notion of positions of elements). To simulate an access operation we instead find the successor of an element in the multiset. This requires a root-to-leaf traversal of the red-black tree, just as

an access operation in a dynamic array implemented as a red-black tree would. Insertion is simulated as an insertion into the multiset, which again requires the same computations as a dynamic array implemented as a red-black tree would.

Besides the random access, range access and insertion tests considered in the previous sections, we have also tested the operations *data dependent access*, insertion in the end, deletion, and *successor*. In the *data dependent access* tests, the next index to lookup depends on the value at the prior lookups. This ensures that the processor cannot successfully pipeline consecutive lookups, but must perform them in sequence. We test insertion in the end, since this is a very common use case. Deletion is performed by deleting elements at random positions. The *successor* operation returns the successor of an element and is not actually part of the dynamic array problem, but is included since it is a commonly used operation on a set in C++. It is simply implemented as a binary search over the elements in both the vector and tiered vector tests where the elements are now inserted in sorted order. The number of tests and operations is the same as in the other tests.

The results are summarized in Table 1 which shows that the vector performs slightly better than the tiered vector on all access and successor tests. As expected from the $\Theta(n)$ running time, it performs extremely poor on random insertion and deletion. For insertion in the end of the sequence, vector is also slightly faster than the tiered vector. The interesting part is that even though the tiered vector requires several extra memory lookups and computations, we have managed to get the running time down to less than the double of the vector for access, even less for data dependent and only a few percent slowdown for range access. As discussed earlier, this is most likely because the entire tree structure (without the elements) fits within the CPU cache, and because the computations required has been minimized.

Comparing our tiered vector to set, we would expect access operations to be faster since they run in $O(1)$ time compared to $O(\log n)$. On the other hand, we would expect insertion/deletion to be significantly slower since it runs in $O(n^{1/l})$ time compared to $O(\log n)$ (where $l = 4$ in these tests). We see our expectations hold for the access operations where the tiered vector faster by more than an order of magnitude. In random insertions however, the tiered vector is only 8% slower – even when operating on 100.000.000 elements. Both the tiered vector and set requires $O(\log n)$ time for the successor operation. In our experiment the tiered vector is 3 times faster for the successor operation.

Finally, we see the memory usage of vector and tiered vector is almost identical. This is expected since in both cases it is primarily the elements themselves that take up space. The set uses more than 10 times as much space, so this is also a considerable drawback of the red-black tree behind this structure.

To sum up, the tiered vectors performs better on all tests but insertion, but is even here highly competitive.

7   Tiered Vector Experiments

In this section we compare different variants of the tiered vector. We first consider how the performance of the different representations of the data structure listed in Section 5, and also how the height of tree and the capacity of the leaves affects the running time. Afterwards we compare it to some widely used C++ standard library containers.

Environment.   All experiments have been performed on a Intel Core i7-4770 CPU @ 3.40GHz with 32 GB RAM. The code has been compiled with GNU GCC version 5.4.0 with flags "-O3". The reported times are an average over 10 test runs.

Table 1 The table summarizes the performance of the implicit tiered vector compared to the performance of set and vector from the C++ standard library. dd-access refers to data dependent access.

	tiered vector	*set*	*set / tiered*	*vector*	*vector / tiered*
access	34.07 ns	1432.05 ns	42.03	21.63 ns	0.63
dd-access	99.09 ns	1436.67 ns	14.50	79.37 ns	0.80
range access	0.24 ns	13.02 ns	53.53	0.23 ns	0.93
insert	1.79 μs	1.65 μs	0.92	21675.49 μs	12082.33
insertion in end	7.28 ns	242.90 ns	33.38	2.93 ns	0.40
successor	0.55 μs	1.53 μs	2.75	0.36 μs	0.65
delete	1.92 μs	1.78 μs	0.93	21295.25 μs	11070.04
memory	408 MB	4802 MB	11.77	405 MB	0.99

(a) *access*   **(b)** *insert*
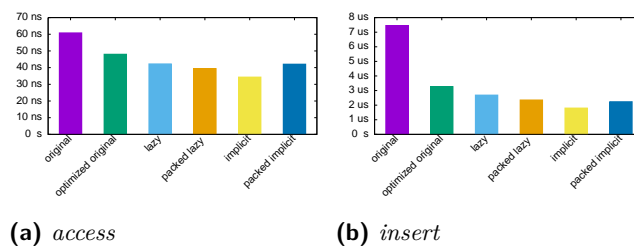
Figure 2 Figures (a) and (b) show the performance of the *original* (━), *optimized original* (━), *lazy* (━) *packed lazy* (━), *implicit* (━) and *packed implicit* (━) layouts.

Procedure. In all tests 10^8 32-bit integers are inserted in the data structure as a preliminary step to simulate that it has already been used[1]. For all the access and successor operations 10^9 elements have been accessed and the time reported is the average time per element. For range access, blocks of 10.000 elements have been used. For insertion/deletion 10^6 elements have been (semi-)randomly[2] added/deleted, though in the case of "vector" only 10.000 elements were inserted/deleted to make the experiments run within reasonable time.

7.1 Tiered Vector Variants Experiments

In this test we compare the performance of the implementations listed in Section 5 to that or the original data structure as described in Theorem 1.

Optimized Original. By co-locating the child offset and child pointer, the two memory lookups are at adjacent memory locations. Due to the cache lines in modern processors, this means the second memory lookup will often be answered directly by the fast L1-cache. As can be seen on Figure 2, this small change in the memory layout results in a significant improvement in performance for both access and insertion. In the latter case, the running time is more than halved.

---

[1] In order to minimize the overall running time of the experiments, the elements were not added randomly, but we show this does not give our data structure any benefits

[2] In order to not impact timing, a simple access pattern has been used instead of a normal pseudo-random generator.

(a) *access*   **(b)** *range access*   **(c)** *insert*
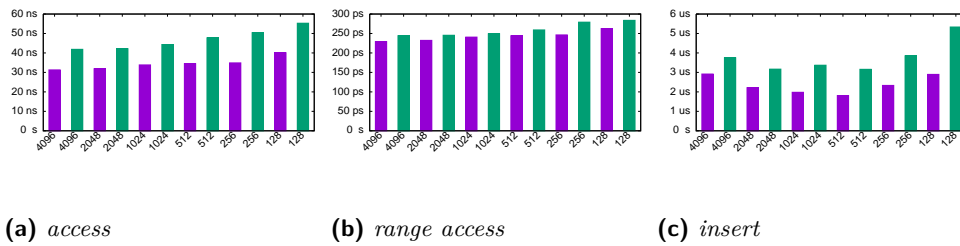
■ **Figure 3** Figures (a), (b) and (c) show the performance of the *implicit* (━) and the *optimized original* tiered vector (━) for different tree widths.

Lazy and Packed Lazy.   Figure 2 shows how the fewer memory probes required by the *lazy* implementation in comparison to the *original* and *optimized original* results in better performance. Packing the offset and pointer in the leaves results in even better performance for both access and insertion even though it requires a few extra instructions to do the actual packing and unpacking.

Implicit.   From Figure 2, we see the implicit data structure is the fastest. This is as expected because it requires fewer memory accesses than the other structures except for the packed lazy which instead has slight computational overhead due to the packing and unpacking.

As shown in Theorem 2 the implicit data structure has a bigger memory overhead than the lazy data structure. Therefore the packed lazy representation might be beneficial in some settings.

Packed Implicit.   Packing the offsets array could lead to better cache performance due to the smaller memory footprint and therefore yield better overall performance. As can be seen on Figure 2, the smaller memory footprint did not improve the performance in practice. The simple reason for this, is that the strategy we used for packing the offsets required extra computation. This clearly dominated the possible gain from the hypothesized better cache performance. We tried a few strategies to minimize the extra computations needed at the expense of slightly worse memory usage, but none of these led to better results than when not packing the offsets at all.

7.2   Width Experiments

This experiment was performed to determine the best capacity ratio between the leaf nodes and the internal nodes. The six different width configurations we have tested are: 32-32-32-4096, 32-32-64-2048, 32-64-64-1024, 64-64-64-512, 64-64-128-256, and 64-128-128-128. All configurations have a constant height 4 and a capacity of approximately 130 mio.

We expected the performance of access operations to remain unchanged, since the number of operations it must perform only depends on the height of the tree, and not the widths. We expect range access to perform better when the leaf size is increased, since more elements will be located in consecutive memory locations. For *insertion* there is not a clearly expected behavior as the time used to physically move elements in a leaf will increase with leaf size, but then less operations on the internal nodes of the tree has to be performed.

On Figure 3 we see access times are actually decreasing slightly when leaves get bigger. This is a bit unexpected, but is most likely due to small changes in the memory layout that results in slightly better cache performance. The same is the case for range access, but this

(a) *access(i)*   **(b)** *access(i, m)*   **(c)** *insert*
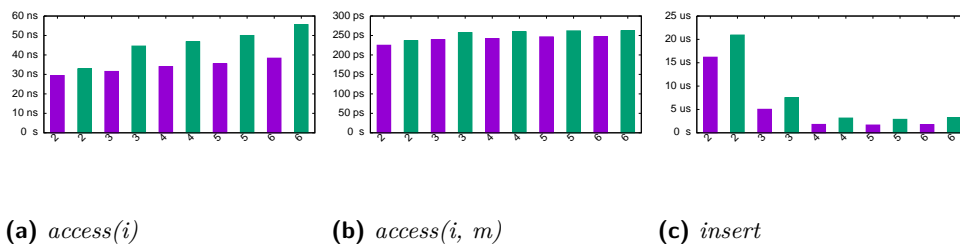
🟧 **Figure 4** Figures (a),(b) and (c) show the performance of the *implicit* (━) and the *optimized original* tiered vector (━) for different tree heights.

(a) *access*   **(b)** *range access*   **(c)** *insert(i,x)*
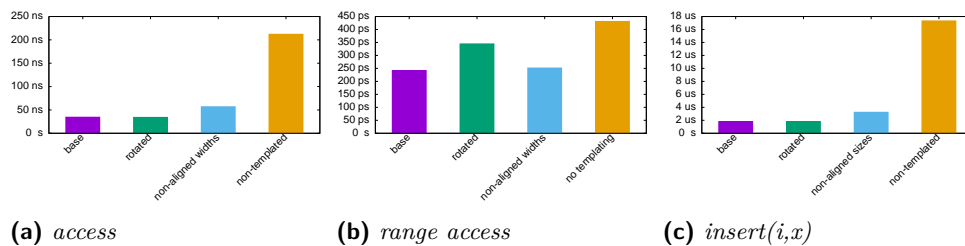
🟧 **Figure 5** Figures (a) and (b) show the performance of the *base* (━), *rotated* (━), *non-aligned sizes* (━), *non-templated* (━) layouts.

was expected. For insertion, we see there is a tipping point. For our particular instance, the best performance is achieved when the leaves have size around 512.

Based on this, we have performed the remaining tests with the 64-64-64-512 configuration (unless otherwise specified).

7.3 Height Experiments

In these tests we have studied how different heights affect the performance of access and insertion operations. We have tested the configurations 8196-16384, 512-512-512, 64-64-64-512, 16-16-32-32-512, 8-8-16-16-16-512. All resulting in the same capacity, but with heights in the range 2-6.

We expect the access operations to performance better for lower trees, since the number of operations that must be performed is linear in the height. On the other hand we expect insertion to perform significantly better with higher trees, since its running time is $O(n^{1/l})$ where l is one the height plus one.

On Figure 4 we see the results follow our expectations. However, the access operations only perform slightly worse on higher trees. We expect this to be because all internal nodes fit within the L3-cache. Therefore the dominant running time comes from the lookup of the element itself. (It is highly unlikely that the element requested by an access to a random position would be among the small fraction of elements that fit in the L3-cache).

Regarding insertion, we see significant improvements up until a height of 4 after that, increasing the height does not change the running time noticeably. This is most likely due to the hidden constant in $O(n^{1/l})$ increases rapidly with the height.

7.4 Configuration Experiments

In these experiments, we test a few hypotheses about how different changes impact the running time. The results are shown on Figure 5, the leftmost result (base) is our final and best implementation to which we compare our hypotheses.

Rotated: As already mentioned, the insertions performed as a preliminary step to the tests are not done at random positions. This means that all offsets are zero when our real operations start. The purpose of this test is the ensure that there are no significant performance gains in starting from such a configuration which could otherwise lead to misleading results. To this end, we have randomized all offsets (in a way such that the data structure is still valid, but the order of elements change) after doing the preliminary insertions but before timing the operations. As can be seen on Figure 5, the difference between this and the normal procedure is insignificant, thus we find our approach gives a fair picture.

Non-Aligned Sizes: In all our previous tests, we have ensured all nodes had an out-degree that was a power of 2. This was chosen in order to let the compiler simplify some calculations, i.e. replacing multiplication/division instructions by shift/and instructions. As Figure 5 shows, using sizes that are not powers of 2 results in significantly worse performance. Besides from showing that one should always pick powers of 2, it also indicates that not only the number of memory accesses during an operation is critical for our performance, but also the amount of computation we make.

Non-Templated: The non-templated results in Figure 2 the show that the change to templated recursion has had a major impact on the running time. It should be noted that some improvements have not been implemented in the non-templated version, but it gives a good indication that this has been quite useful.

8 Conclusion

This paper presents the first implementation of a generic tiered vector supporting any constant number of tiers. We have shown a number of modified version of the tiered vector, and employed several speed optimizations to the implementation. These implementations have been compared to vector and multiset from the C++ standard library. The benchmarks show that our implementation stays on par with vector for access and on update operations while providing a considerable speedup of more than $40\times$ compared to set. At the same time the asymptotic difference between the logarithmic complexity of multiset and the polynomial complexity of tiered vector for insertion and deletion operations only has little effect in practice. For these operations, our fastest version of the tiered vector suffers less than 10% slowdown. Arguably, our tiered array provides a better trade-off than the balanced binary tree data structures used in the standard library for most applications that involves big instances of the dynamic array problem.

References

1 Andrej Brodnik, Svante Carlsson, Erik D. Demaine, J. Ian Munro, and Robert Sedgewick. Resizable arrays in optimal time and space. In *Proceedings of the 6th International Workshop on Algorithms and Data Structures*, WADS'99, pages 37–48, London, UK, UK, 1999. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=645932.673194`.

2 Paul F. Dietz. Optimal algorithms for list indexing and subset rank. In *Proceedings of the Workshop on Algorithms and Data Structures*, WADS'89, pages 39–46, London, UK, UK, 1989. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=645928.672529`.

3 Greg N. Frederickson. Implicit data structures for the dictionary problem. *J. ACM*, 30(1):80–94, January 1983. `doi:10.1145/322358.322364`.

4   M. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing*, STOC'89, pages 345–354, New York, NY, USA, 1989. ACM. `doi:10.1145/73007.73040`.

5   Michael T. Goodrich and John G. Kloss. Tiered vectors: Efficient dynamic arrays for rank-based sequences. In Frank Dehne, Jörg-Rüdiger Sack, Arvind Gupta, and Roberto Tamassia, editors, *Algorithms and Data Structures: 6th International Workshop, WADS'99 Vancouver, Canada, August 11–14, 1999 Proceedings*, pages 205–216, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. `doi:10.1007/3-540-48447-7_21`.

6   Jyrki Katajainen. Worst-case-efficient dynamic arrays in practice. In Andrew V. Goldberg and Alexander S. Kulikov, editors, *Experimental Algorithms: 15th International Symposium, SEA 2016, St. Petersburg, Russia, June 5-8, 2016, Proceedings*, pages 167–183, Cham, 2016. Springer International Publishing. `doi:10.1007/978-3-319-38851-9_12`.

7   Jyrki Katajainen and Bjarke Buur Mortensen. Experiences with the design and implementation of space-efficient deques. In Gerth Stølting Brodal, Daniele Frigioni, and Alberto Marchetti-Spaccamela, editors, *Algorithm Engineering: 5th International Workshop, WAE 2001 Århus, Denmark, August 28–31, 2001 Proceedings*, pages 39–50, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. `doi:10.1007/3-540-44688-5_4`.

8   Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct dynamic data structures. In Frank Dehne, Jörg-Rüdiger Sack, and Roberto Tamassia, editors, *Algorithms and Data Structures: 7th International Workshop, WADS 2001 Providence, RI, USA, August 8–10, 2001 Proceedings*, pages 426–437, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. `doi:10.1007/3-540-44634-6_39`.