Experiences with the design and implementation of space-efficient deques

Jyrki Katajainen and Bjarke Buur Mortensen

Department of Computing, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
{jyrki, rodaz}@diku.dk
http://www.diku.dk/research-groups/performance-engineering/

Abstract. A new realization of a space-efficient deque is presented. The data structure is constructed from three singly resizable arrays, each of which is a blockwise-allocated pile (a heap without the order property). The data structure is easily explainable provided that one knows the classical heap concept. All core deque operations are performed in $O(1)$ worst-case time. Also, general modifying operations are provided which run in $O(\sqrt{n})$ time if the structure contains n elements. Experiences with an implementation of the data structure show that, compared to an existing library implementation, the constants for some of the operations are unfavourably high, whereas others show improved running times.

1 Introduction

A ***deque*** (***double-ended queue***) is a data type that represents a sequence which can grow and shrink at both ends efficiently. In addition, a deque supports random access to any element given its ***index***. Insertion and erasure of elements in the middle of the sequence are also possible, but these should not be expected to perform as efficiently as the other operations. A deque is one of the most important components of the C++ standard library; sometimes it is even recommended to be used as a replacement for an array or a vector (see, e.g., [13]).

Let `X` be a deque, `n` an index, `p` a valid iterator, `q` a valid dereferenceable iterator, and `r` a reference to an element. Of all the deque operations four are fundamental:

operation	***effect***
`X.begin()`	returns a random access iterator referring to the first element of `X`
`X.end()`	returns a random access iterator referring to the one-past-the-end element of `X`
`X.insert(p, r)`	inserts a copy of element referred to by `r` into `X` just before `p`
`X.erase(q)`	erases the element referred to by `q` from `X`

We call the insert and erase operations collectively the ***modifying operations***. The semantics of the ***sequence operations***, as they are called in the C++ standard, can be defined as follows:

operation	*operational semantics*
`X[n]`	`*(X.begin() + n)` (no bounds checking)
`X.at(n)`	`*(X.begin() + n)` (bounds-checked access)
`X.front()`	`*(X.begin())`
`X.back()`	`*(--(X.end()))`
`X.push_front(r)`	`X.insert(X.begin(), r)`
`X.pop_front()`	`X.erase(X.begin())`
`X.push_back(r)`	`X.insert(X.end(), r)`
`X.pop_back()`	`X.erase(--(X.end()))`

For a more complete description of all deque operations, we refer to the C++ standard [7, Clause 23], to a textbook on C++, e.g., [12], or to a textbook on the Standard Template Library (STL), e.g., [10].

In this paper we report our experiences with the design and implementation of a deque which is space-efficient, supports fast sequence operations, and has relatively fast modifying operations. Our implementation is part of the Copenhagen STL which is an open-source library under development at the University of Copenhagen. The purpose of the Copenhagen STL project is to design alternative/enhanced versions of individual STL components using standard performance-engineering techniques. For further details, we refer to the Copenhagen STL website [5].

The C++ standard states several requirements for the complexity of the operations, exception safety, and iterator validity. Here we focus on the time- and space-efficiency of the operations. According to the C++ standard all sequence operations should take $O(1)$ time in the worst case. By ***time*** we mean the sum of operations made on the elements manipulated, on iterators, and on any objects of the built-in types. Insertion of a single element into a deque is allowed to take time linear in the minimum of the number of elements between the beginning of the deque and the insertion point and the number of elements between the insertion point and the end of the deque. Similarly, erasure of a single element is allowed to take time linear in the minimum of the number of elements before the erased element and the number of elements after the erased element.

In the Silicon Graphics Inc. (SGI) implementation of the STL [11], a deque is realized using a number of data blocks of fixed size and an index block storing pointers to the beginning of the data blocks. Only the first and the last data block can be non-full, whereas all the other data blocks are full. Adding a new element at either end is done by inserting it into the first/last data block. If the relevant block is full, a new data block is allocated, the given element is put there, and a pointer to the new block is stored in the index block. If the index block is full, another larger index block is allocated and the pointers to the data blocks are moved there. Since the size of the index block is increased by a constant factor, the cost of the index block copying can be amortized over the push operations. Hence, the push operations are supported in $O(1)$ amortized time and all the other sequence operations in $O(1)$ worst-case time. Thus this realization is not fully compliant with the C++ standard. Also, the

space allocated for the index block is never freed so the amount of extra space used is not necessarily proportional to the number of elements stored.

Recently, Brodnik et al. [3] announced the existence of a deque which performs the sequence operations in $O(1)$ worst-case time and which requires never more than $O(\sqrt{n})$ extra space (measured in elements and in objects of the built-in types) if the deque stores n elements. After reading their conference paper, we decided to include their deque realization in the Copenhagen STL. For the implementation details, they referred to their technical report [4]. After reading the report, we realized that some implementation details were missing; we could fill in the missing details, but the implementation got quite complicated. The results of this first study are reported in [8]. The main motivation of this first study was to understand the time/space tradeoff better in this context. Since the results were a bit unsatisfactory, we decided to design the new space-efficient data structure from scratch and test its competitiveness with SGI's deque.

The new design is described in Sections 2–5. For the sequence operations, our data structure gives the same time and space guarantees as the proposal of Brodnik et al. [4]. In addition, using the ideas of Goodrich and Kloss II [6] we can provide modifying operations that run in $O(\sqrt{n})$ time. Our solution is based on an efficient implementation of a resizable array, i.e., a structure supporting efficient inserts and erases only at one end, which is similar to that presented by Brodnik et al. [3, 4]. However, after observing that "deques cannot be efficiently implemented in the worst case with two stacks, unlike the case of queues", they use another paradigm for realizing a deque. While their observation is correct, we show that a deque can be realized quite easily using three resizable arrays. One can see our solution as a slight modification of the standard "two stacks" technique relying on global rebuilding [9]. All in all, our data structure is easily explainable which was one of the design criteria of Brodnik et al.

The experimental results are reported in Section 6. Compared to SGI's deque, for our implementation shrink operations at the ends may be considerably slower, grow operations at the ends are only a bit slower, access operations are a bit faster, and modifying operations are an order of magnitude faster.

2 Levelwise-allocated piles

A ***heap***, as defined by Williams [14], is a data structure with the following four properties:

Shape property: It is a left-complete binary tree, i.e., a tree which is obtained from a complete binary tree by removing some of its rightmost leaves.

Capacity property: Each node of the tree stores one element of a given type.

Representation property: The tree is represented in an array $\mathbf{a}[0\,..\,n)$ by storing the element at the root of the tree at entry 0, the elements at its children at entries 1 and 2, and so on.

Order property: Assuming that we are given an ordering on the set of elements, for each branch node the element stored there is no smaller than the element stored at any children at that node.

Our data structure is based on a heap but for us the order property is irrelevant. For the sake of clarity, we call the data structure having only the shape, capacity, and representation properties a ***static pile***.

The main drawback of a static pile is that its size n must be known beforehand. To allow the structure to grow and shrink at the back end, we allocate space for it levelwise and store only those levels that are not empty. We call this stem of the data structure a ***levelwise-allocated pile***. We also need a separate array, called here the ***header***, for storing the pointers to the beginning of each level of the pile. Theoretically, the size of this header is $\lceil \log_2(n+1) \rceil$, but a fixed array of size, say 64, will be sufficient for all practical purposes. The data structure is illustrated in Figure 1. Observe that element $\mathbf{a}[k]$, $k \in \{0, 1, \ldots, n-1\}$, has index $k - 2^{\lfloor \log_2(k+1) \rfloor} + 1$ at level $\lfloor \log_2(k+1) \rfloor$.

The origin of the levelwise-allocated pile is unclear. The first author of this paper gave the implementation of a levelwise-allocated heap as a programming exercise for his students in May 1998, but the idea is apparently older. Bojesen [2] used the idea in the implementation of dynamic heaps in his heaplab. According to his experiments the practical performance of a levelwise-allocated heap is almost the same as that of the static heap when used in Williams' heapsort [14].

If many consecutive grow and shrink operations are performed at a level boundary, it might happen that the memory for a level is repeatedly allocated and deallocated. We can assume that both of these memory-allocation operations require constant time, but in practice the constant is high (see [1, Appendix 3]). To amortize the memory-allocation costs, we do not free the space reserved by the highest level h until all the elements from level $h-1$ have been erased. Also, it is appropriate to allocate the space for the first few levels (8 in our actual implementation) statically so that the extra costs caused by memory allocation can be avoided altogether for small piles.

For a data structure storing n elements, the space allocated for elements is never larger than $4n+O(1)$. Additionally, the extra space for $O(\log_2 n)$ pointers is needed by the header. If we ignore the costs caused by the dynamization of the header — as pointed out in practice there are no costs — a levelwise-allocated pile provides the same operations equally efficiently as a static pile. In addition, the grow and shrink operations at the back end are possible in $O(1)$ worst-case time. For instance, to locate an element only a few arithmetic operations are needed for determining its level and its position at that level; thereafter only two memory accesses are needed. To determine the level, at which element $\mathbf{a}[k]$ lies, we have to compute $\lfloor \log_2(k+1) \rfloor$. Since the computation of the whole-number logarithm of a positive integer fitting into a machine word is an AC^0 instruction, we expect this to be fast. In our programs, we have used the whole-number logarithm function available in our C library (`<cmath>`) which turned out to be faster than our home-made variants.

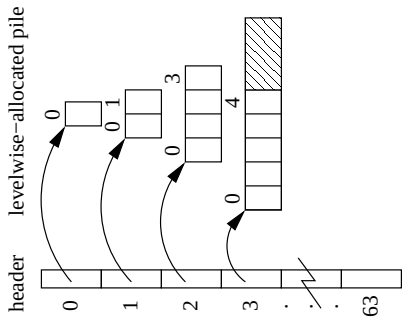

Fig. 1. A levelwise-allocated pile storing 12 elements.

3 Space-efficient singly resizable arrays

A ***singly resizable array*** is a data structure that supports the grow and shrink operations at the back end plus the location of an arbitrary element, all in constant worst-case time. That is, a levelwise-allocated pile could be used for implementing a singly resizable array. In this section we describe a realization of a singly resizable array that requires only $O(\sqrt{n})$ extra space if the data structure

contains n elements. The structure is similar to that presented by Brodnik et al. [3, 4], but we use a pile to explain its functioning.

Basically, our realization of a singly resizable array is nothing but a pile where each level ℓ is divided into blocks of size $2^{\lceil \ell/2 \rceil}$ and where space is allocated only for those blocks that contain elements. Therefore, we call it a ***blockwise-allocated pile***. Again to avoid the allocation/deallocation problem at block boundaries, we maintain the invariant that there may exist only at most one empty block, i.e., the last empty block is released when the block prior to it gets empty. The pointers to the beginning of the blocks are stored separately in a ***levelwise-allocated twin-pile***; we call this structure a twin-pile since the number of pointers at level ℓ is $2^{\lfloor \ell/2 \rfloor}$. Therefore, in a twin-pile two consecutive levels can be of the same size, but the subsequent level must be twice as large. The data structure is illustrated in Figure 2.

Since the block sizes grow geometrically, the size of the largest block is proportional to $\sqrt{n}$ if the structure stores n elements. Also, the number of blocks is proportional to $\sqrt{n}$. In the twin-pile there are at most two non-full levels. Hence, $O(\sqrt{n})$ extra space is used for pointers kept there. In the blockwise-allocated pile there are at most two non-full blocks. Hence, $O(\sqrt{n})$ extra space is reserved there for elements.

The location of an element is almost equally easy as in the levelwise-allocated pile; now only three memory accesses are necessary. Resizing is also relatively easy to implement. When the size is increased by one and the corresponding block does not exist in the blockwise-allocated pile, a new block is allocated (if there is no empty block) and a pointer to the beginning of that block is added to the end of the twin-pile as described earlier. When the size is decreased by one and the corresponding block gets empty, the space for the preceding empty block is released (if there is any); the shrinkage in the twin-pile is handled as described earlier.

One crucial property, which we use later on, is that a space-efficient singly resizable array of a given size can be constructed in reverse order and it can be used simultaneously during such a construction already after the first element is moved into the structure. Even if part of the construction is done in connection with each shrink operation, more precisely before it, the structure retains its usability during the whole construction. Furthermore, in this organization space need only be allocated for non-empty blocks in the blockwise-allocated pile and for non-empty levels in the twin-pile.

4 Space-efficient doubly resizable arrays

A ***doubly resizable array*** is otherwise as a singly resizable array but it can grow and shrink at both ends. We use two singly resizable arrays to emulate a doubly resizable array. We call the singly resizable arrays A and B, respectively, and the doubly resizable array emulated by these D. Assume that A and B are connected together such that A implements the changes at the front end of D and B those at the back end of D. From this the indexing of the elements is

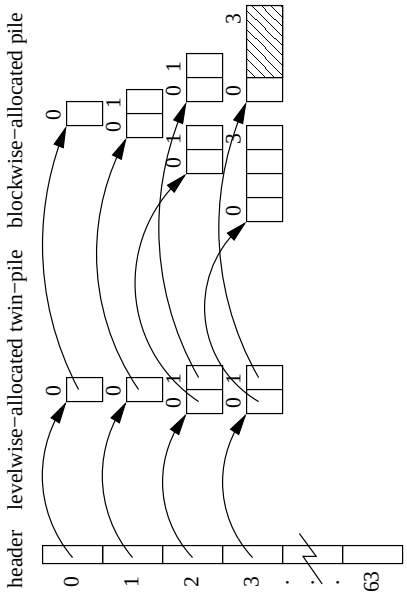

Fig. 2. A space-efficient singly resizable array storing 12 elements.

easily derived. This emulation works perfectly well unless A or B gets empty. Next we describe how this situation can be handled time- and space-efficiently.

Assume that A gets empty, and let m denote the size of B when this happens. The case where B gets empty is handled symmetrically. The basic idea is to halve B, move the first half of its elements (precisely $\lfloor m/2 \rfloor$ elements) to A, and the remaining half of its elements (precisely $\lceil m/2 \rceil$ elements) to a new B. This reorganization work is distributed for the next $\lfloor m/d \rfloor$ shrink operations to the structure D, where $d \geq 2$ is an even integer to be determined experimentally.

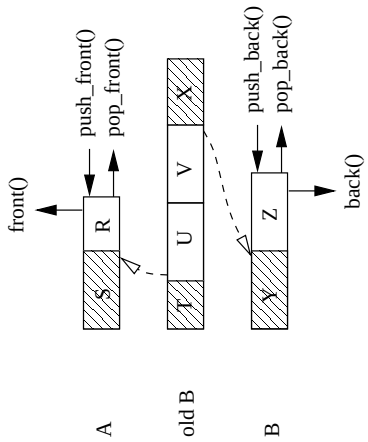

Fig. 3. Illustration of the reorganization.

If $\lfloor m/d \rfloor \cdot d < m$, $\lfloor (m \bmod d)/2 \rfloor$ elements are moved to A and $\lceil (m \bmod d)/2 \rceil$ elements to B before the $\lfloor m/d \rfloor$ reorganization steps are initiated. In each of the $\lfloor m/d \rfloor$ reorganization steps, $d/2$ elements are moved from old B to A and $d/2$ elements from old B to B. The construction of A and B is done in reverse order so that they can be used immediately after they receive the first bunch of elements.

Figure 3 illustrates the reorganization. The meaning of the different zones in the figure is as follows. Zone U contains elements still to be moved from old B

to A and zone S receives the elements coming from zone U. Zone R contains the elements already moved from zone T in old B to A; some of the elements moved may be erased during the reorganization and some new elements may have been inserted into zone R. Zone V contains the remaining part of old B to be moved to zone Y in B. Zone Z in B contains the elements received from zone X in old B; zone Z can receive new elements and loose old elements during the reorganization. The elements of the doubly resizable array D appear now in zones R, U, V, and Z.

If a reorganization process is active prior to the execution of a shrink operation (involving D), the following steps are carried out.

1. Take $d/2$ elements from zone U (from the end neighbouring zone T) and move them into zone S (to the end neighbouring zone R) in reverse order.
2. Take $d/2$ elements from zone V (from the end neighbouring zone X) and move them into zone Y (to the end neighbouring zone Z).

In these movements in the underlying singly resizable arrays, new blocks are allocated when necessary and old blocks are deallocated when they get empty. This way only at most a constant number of non-full blocks in the middle of the structures exists and the zones S, T, X, and Y do not consume much extra space. The same space saving is done for levels in the twin-piles.

Even if all modifying operations (involving D) done during a reorganization make A or B smaller, both of them can service at least $\lfloor m/2 \rfloor$ operations, because the work done in a single reorganization is divided for $\lfloor m/d \rfloor$ shrink operations and $d \geq 2$. Therefore, there can never be more than one reorganization process active at a time. To represent D, at most three singly resizable arrays are used. If D contains n elements, the size of A and B cannot be larger than n. Furthermore, if the size of old A or old B was m just before the reorganization started, $m \leq 2n$ at all times since the reorganization is carried out during the next $\lfloor m/d \rfloor$ shrink operations and $d \geq 2$. Hence, the number of blocks and the size of the largest block in all the three substructures is proportional to $\sqrt{n}$. That is, the bound $O(\sqrt{n})$ for the extra space needed is also valid for doubly resizable arrays.

When we want to locate the element with index k in D, we have to consider two cases. First, if no reorganization process is active, the element is searched for from A or B depending on their sizes. Let $|Z|$ denote the size of zone Z. If $k < |A|$, the element with index $|A|-k-1$ in A is returned. If $k \geq |A|$, the element with index $k-|A|$ in B is returned. Second, if a reorganization process is active, the element is searched for from zones R, U, V, and Z depending on their sizes. If $k < |R|$, the element with index $|A|-k-1$ in A is returned. If $|R| \leq k < |R|+|U|+|V|$, the element with index $k-|R|$ in old B is returned. If $|R|+|U|+|V| \leq k < |R|+|U|+|V|+|Z|$, the element with index $k-|R|-|U|$ in B is returned. The case where old A exists is symmetric. Clearly, the location requires only a constant number of comparisons and arithmetic operations plus an access to a singly resizable array.

5 Space-efficient deques

The main difference between a doubly resizable array and a deque is that a deque must also support the modifying operations. Our implementation of a space-efficient doubly resizable array can be directly used if the modifying operations simply move the elements in their respective singly resizable arrays one location backwards or forwards, depending on the modifying operation in question. This also gives the possibility to complete a reorganization process if there is one that is active. However, this will only give us linear-time modifying operations.

More efficient working is possible by implementing the blocks in the underlying singly resizable arrays circularly as proposed by Goodrich and Kloss II [6]. If the block considered is full, a ***replace*** operation, which removes the first element of the block and inserts a new element at the end of the block, is easy to implement in $O(1)$ time. Only a cursor to the current first element need to be maintained; this is incremented by one (modulus the block size) and the earlier first element is replaced by the new element. A similar replacement that removes the last element of the block and adds a new element to the beginning of the block is equally easy to implement. If the block is not full, two cursors can be maintained after which replace, insert, and erase operations are all easy to accomplish.

In a space-efficient singly resizable array an insert operation inserting a new element just before the given position can be accomplished by moving the elements (after that position) in the corresponding block one position forward to make place for the new element, by adding the element that fell out of the block to the following block by executing a replace operation, and by continuing this until the last block is reached. In the last block the insertion reduces to a simple insert operation. The worst-case complexity of this operation is proportional to the number of blocks plus the size of the largest block.

An erase operation erasing the element at the given position can be carried out symmetrically. The elements after that position in the corresponding block are moved backward to fill out the hole created, the hole at the end is filled out by moving the first element of the following block here, and this filling process is repeated until the last block is reached, where the erasure reduces to a simple erase operation. Clearly, the worst-case complexity is asymptotically the same as that for the insert operation.

In a space-efficient doubly resizable array, the repeated replace strategy is applied inside A, old A/B, or B depending on which of these the modifying operation involves. Furthermore, the modifying operation involving old B (old A) should propagate to B (A). If a reorganization process is active, one step of the reorganization is executed prior to erasing an element.

Since the blocks are realized circularly, for each full block one new cursor pointing to the current first element of the circular block must be stored in the twin-piles. This will only double their size. There are a constant number of non-full blocks (at the ends of zones R, U, V, and Z); for each of these blocks one more cursor is needed for indicating the location of its last element, but these cursors require only a constant amount of extra space.

To summarize, all sequence operations run in $O(1)$ worst-case time. If the deque stores n elements, the total number of blocks in A, B, and old A/B is proportional to $\sqrt{n}$; similarly, the size of the largest block is proportional to $\sqrt{n}$. In connection with every modifying operation, in one block $O(\sqrt{n})$ elements are moved one position forwards or backwards and at most $O(\sqrt{n})$ blocks are visited. Therefore, the modifying operations run in $O(\sqrt{n})$ time.

6 Experimental results

In this section we report the results of a series of benchmarks where the overall goal was to measure the cost of being space-efficient. This is done by comparing the efficiency of our implementation to the efficiency of SGI's implementation for the core deque operations. For reference, we have included the results for SGI's vector in our comparisons.

All benchmarks were carried out on a dual Pentium III system with 933 Mhz processors (16 KB instruction cache, 16 KB data cache and 256 KB second level cache) running RedHat Linux 6.1 (kernel version 2.2.16-3smp). The machine had 1 GB random access memory. The compiler used was `gcc` (version 2.95.2) and the C++ standard library shipped with this compiler included the SGI STL (version 3.3). All optimizations were enabled during compilation (using option `-O6`). The timings have been performed on integer containers of various sizes by using the `clock()` system call. For the constant time operations, time is reported per operation and has been calculated by measuring the total time of executing the operations and dividing this by the number of operations performed.

The first benchmarks were done to determine the best value for d. The results of these suggested that the choice of d was not very important for the performance of our data structure. For instance, doing a test using $d = 4$ in which we made just as many operations (here `pop_back`s) as there were elements to be restructured improved the time per `pop_back` by approximately 10 ns compared to $d = 2$. Increasing d to values higher than 32 did not provide any significant improvement in running times. This indicates that our data structure does not benefit noticeably from memory caching. The improvements in running times come exclusively from shortening the restructuring phase. Our choice for the value of d has thus become 4.

Next we examined the performance of the sequence operations. The best case for our data structure is when neither singly resizable array used in the emulation becomes empty, since reorganization will then never be initiated. The worst case occurs when an operation is executed during a reorganization. The following table summarizes the results for `push_back` and `pop_back` operations. Results for `push_front` and `pop_front` operations were similar for SGI's deque and our deque and are omitted; vector does not support these operations directly.

container	`push_back` *(ns)*	`pop_back` *(ns)*
`std::deque`	85	11
`std::vector`	115	2
space-efficient deque	113	35
space-efficient deque (with reorganization)	113	375

The performance of `push_back` operations for our deque is on par with that for SGI's vector, which suffers from the need to reallocate its memory from time to time. Compared to SGI's deque there is an overhead of approximately 30 percent. This overhead is expected, since there is more bookkeeping to be done for our data structure. The overhead of SGI's deque for reallocating its index block is small enough to outperform our deque.

Looking at the `pop_back` operations SGI's deque and vector are about 3 and 15 times faster than our deque when no reorganization is involved. The reason for this is that these two structures do not deallocate memory until the container itself is destroyed. For SGI's deque, the index block is never reallocated to a smaller memory block, and the same goes for the entire data block of SGI's vector. In fact, for SGI's vector the `pop_back` operation reduces to executing a single subtraction, resulting in a very low running time. This running time was verified to be equal to the running time of a single subtraction by running Bentley's instruction benchmark program, see [1, Appendix 3]. When reorganization is involved, for our deque `pop_back` operations are approximately 34 times slower than SGI's deque. What is particularly expensive is that restructuring requires new memory to be allocated and elements to be moved in memory.

Accessing an element in a vector translates into an integer multiplication (or a shift), an addition, and a load or store instruction, whereas access to a deque is more complicated. Even though SGI's deque is simple, experiments reported in [8] indicated that improving access times compared to SGI's deque is possible if we rely on shift instructions instead of division and modulus instructions. The following table gives the average access times per operation when performing repeated sequential accesses and repeated random accesses, respectively.

container	*sequential access (ns)*	*random access (ns)*
`std::deque`	117	210
`std::vector`	2	60
space-efficient deque	56	160
space-efficient deque (with reorganization)	58	162

From the results it is directly seen that, due to its contiguous allocation of memory, a vector benefits much more from caching than the other structures when the container is traversed sequentially. For SGI's deque the running time is reduced by about a factor two and for our deque the running time is reduced by about a factor three, SGI's deque being a factor two slower than our deque. As regards random access, SGI's deque is approximately 1.3 times slower than

our deque, even though the work done to locate an element in our data structure comprises more instructions. To locate the data block to which an element belongs, SGI's deque needs to divide an index by the block size. The division instruction is expensive compared to other instructions (see [1, Appendix 3]). In our deque, we must calculate for instance $2^{\lfloor k/2 \rfloor}$ (the number of blocks at level `k`), which can be expressed using left (`<<`) and right (`>>`) shifts as `1 << (k >> 1)`. Furthermore, because our data blocks are circular, we need to access elements in these blocks modulus the block size. Since our data block sizes are always powers of two, accessing element with index `i` in a block of size `b` starting at index `h` can be done by calculating `(h + i) & (b - 1)` instead of `(h + i) % b`. Modulus is just as expensive as division and avoiding it makes access in circular blocks almost as fast as access in vectors.

The improved time bounds for insert and erase operations achieved by using circular blocks are clearly evident from the benchmark results. The table below gives the results of a test inserting 1 000 elements in the middle of the container. Results for the erase operation were similar and are omitted.

container	1 000 `insert`***s (s)*** ***initial size*** 10 000	1 000 `insert`***s (s)*** ***initial size*** 100 000	1 000 `insert`***s (s)*** ***initial size*** 1 000 000
`std::deque`	0.07	1.00	17.5
`std::vector`	0.015	0.61	12.9
space-efficient deque	0.003	0.01	0.04

With 100 000 elements in the container before the 1 000 insert operations, SGI's deque is 100 times slower than our deque, and SGI's vector is 15 times slower. The difference between $O(n)$ and $O(\sqrt{n})$ is even more clear when n is 1 000 000. SGI's deque and vector are outperformed approximately by a factor 436 and factor 321, respectively.

References

1. J. Bentley, *Programming Pearls*, 2nd Edition, Addison-Wesley, Reading, Massachusetts (2000).
2. J. Bojesen, Heap implementations and variations, Written Project, Department of Computing, University of Copenhagen, Copenhagen, Denmark (1998). Available at `http://www.diku.dk/research-groups/performance-engineering/resources.html`.
3. A. Brodnik, S. Carlsson, E. D. Demaine, J. I. Munro, and R. Sedgewick, Resizable arrays in optimal time and space, *Proceedings of the 6th International Workshop on Algorithms and Data Structures*, *Lecture Notes in Computer Science* **1663**, Springer-Verlag, Berlin/Heidelberg, Germany (1999), 37–48.
4. A. Brodnik, S. Carlsson, E. D. Demaine, J. I. Munro, and R. Sedgewick, Resizable arrays in optimal time and space, Technical Report CS-99-09, Department of Computer Science, University of Waterloo, Waterloo, Canada (1999). Available at `ftp://cs-archive.uwaterloo.ca/cs-archive/CS-99-09/`.

5. Department of Computing, University of Copenhagen, The Copenhagen STL, Website accessible at `http://cphstl.dk/` (2000–2001).
6. M. T. Goodrich and J. G. Kloss II, Tiered vectors: Efficient dynamic arrays for rank-based sequences, *Proceedings of the 6th International Workshop on Algorithms and Data Structures*, *Lecture Notes in Computer Science* **1663**, Springer-Verlag, Berlin/Heidelberg, Germany (1999), 205–216.
7. ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission), *International Standard ISO/IEC 14882: Programming Languages — C++*, Genève, Switzerland (1998).
8. B. B. Mortensen, The deque class in the Copenhagen STL: First attempt, Copenhagen STL Report 2001-4, Department of Computing, University of Copenhagen, Copenhagen, Denmark (2001). Available at `http://cphstl.dk/`.
9. M. H. Overmars, *The Design of Dynamic Data Structures*, *Lecture Notes in Computer Science* **156**, Springer-Verlag, Berlin/Heidelberg, Germany (1983).
10. P. J. Plauger, A. A. Stepanov, M. Lee, and D. R. Musser, *The C++ Standard Template Library*, Prentice Hall PTR, Upper Saddle River, New Jersey (2001).
11. Silicon Graphics, Inc., Standard template library programmer's guide, Worldwide Web Document (1990–2001). Available at `http://www.sgi.com/tech/stl/`.
12. B. Stroustrup, *The C++ Programming Language*, 3rd Edition, Addison-Wesley Publishing Company, Reading, Massachusetts (1997).
13. H. Sutter, Standard library news, part 1: Vectors and deques, *C++ Report* **11**,7 (1999). Available at `http://www.gotw.ca/publications/index.htm`.
14. J. W. J. Williams, Algorithm 232: Heapsort, *Communications of the ACM* **7** (1964), 347–348.