Experiences with the design and implementation of space-efficient deques

Jyrki Katajainen
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 $O(1)$ time in the worst case. 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 shows 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***. By default, the index of the first element is zero. 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., [Sutter 1999]). For the ease of reference, the complete declaration of the deque class, as specified by the ISO/IEC standard for the C++ programming language [ISO and IEC 1998, Clause 23], is given in Appendix A.

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:

Copenhagen STL Report 2001-7, April 2001.

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 [ISO and IEC 1998, Clause 23], to a textbook on C++, e.g., [Stroustrup 1997], or to a textbook on the Standard Template Library (STL), e.g., [Plauger et al. 2001].

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 the 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 [Department of Computing, University of Copenhagen 2000–2001].

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 manipulated elements, on iterators, and on any objects of the built-in types [Stroustrup 1997, Section 4.1.1]. 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 [Silicon Graphics, Inc. 1990–2001], 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. By doubling the size of the index block, 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 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. [1999a] 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 built-in objects) 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 [Brodnik et al. 1999b]. 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 [Mortensen 2001]. 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. [1999b]. In addition, using the ideas of Goodrich and Kloss II [1999] 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. [1999a, 1999b]. 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 by using three resizable arrays. One can see our solution as a slight modification of the standard "two stacks" technique relying on global rebuilding [Overmars 1983]. 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 the grow and shrink operations at the ends may be a constant factor slower, access operations are a bit faster, and

modifying operations are an order of magnitude faster.

2. Levelwise-allocated piles

The term ***heap***, originally defined by Williams [1964], is a data structure with the following four properties:

Shape property: A heap 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 that 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 the space for it levelwise and store only the levels that are not empty. For this purpose we need a separate array, called here the ***header***, for storing the pointers to the beginning of each level. 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, called the ***levelwise-allocated pile***, is illustrated in Figure 2.1. 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. Also, Bojesen [1998] 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 [Williams 1964]. In contrast, the heap implementation relying on the standard doubling technique can be considerably slower due to the extra copying of elements.

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 [Bentley 2000, 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

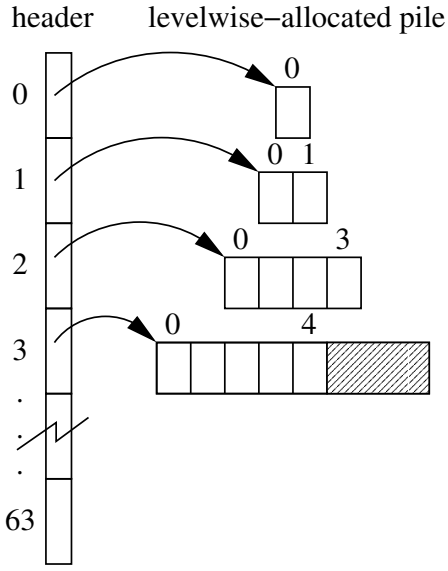

Figure 2.1. A levelwise-allocated pile of 12 elements.

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. 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. We expect that the header is kept in cache at all times so the first memory access should be fast. 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.

3. 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. Now we describe a realization of a singly resizable array that requires only $O(\sqrt{n})$ extra space (for elements and point-

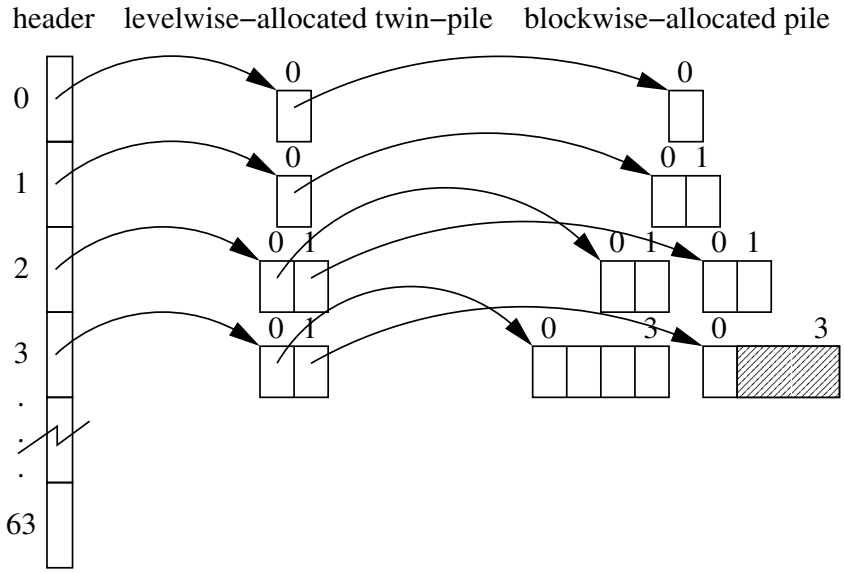

Figure 3.1. A singly resizable array of 12 elements.

ers) if the data structure contains n elements. The structure is similar to that presented by Brodnik et al. [1999a, 1999b], 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. This organization induces a natural order among the blocks. Again to avoid the allocation/deallocation problem at block boundaries, we maintain the invariant that there may exist only at most one empty blocks, 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 the resulting pile 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 level after them must be double as large. The whole data structure is illustrated in Figure 3.1.

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. Since the size of the twin-pile is relatively small — even for large n — compared to the cache sizes of contemporary computers, we expect that both the

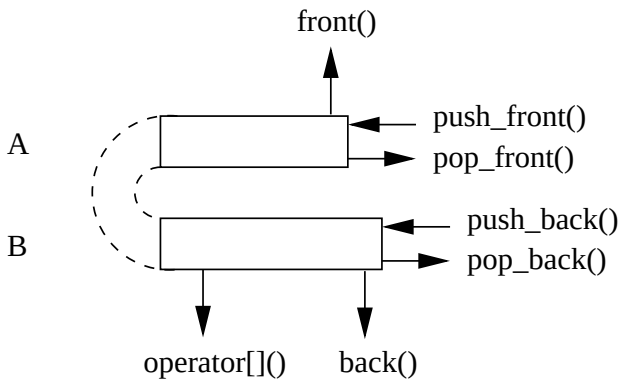

Figure 4.1. Our implementation of a doubly resizable array.

header and the twin-pile are kept in cache most of the time. Therefore, the two extra memory accesses should not make the cache performance much worse, but of course if the data structure is used simultaneously with other structures, a worsening of the cache behaviour is possible. In particular, our implementation of a deque uses three resizable arrays at the same time, so there some extra cache misses are probable.

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 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. If part of the reorganization is done in connection with each grow and shrink operation, the structure can be used with no problems if the number of shrink operations does not exceed the given size. Furthermore, this organization is space-efficient since space must be allocated only for those blocks in the blockwise-allocated pile and for those levels in the twin-pile that are not empty.

4. Doubly resizable arrays

A ***doubly resizable array*** is otherwise as a singly resizable array but it can grow and shrink at both ends (see Figure 4.1). We use two resizable arrays to implement a doubly resizable array. We call the singly resizable arrays A and B, respectively, and the doubly resizable array realized by these D. For illustrative purposes, assume that the arrays A and B are

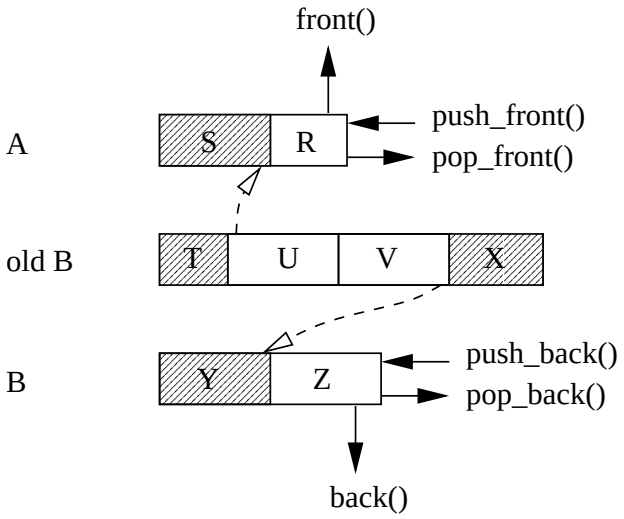

Figure 4.2. Illustration of a reorganization.

connected together such that A can implement 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 easily derived. This simulation 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 when 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 is done during the next $\lfloor m/d \rfloor$ operations to the structure D (including the operation that triggered off the reorganization), where $d \geq 2$ is an even integer to be determined experimentally. If $\lfloor m/d \rfloor \cdot d < m$, $\lfloor m/2 \rfloor \bmod (d/2)$ elements are moved to A and $\lceil m/2 \rceil \bmod (d/2)$ 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 the old B to A and $d/2$ elements from the old B to the new 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 4.2 illustrates the reorganization operation. The meaning of the different zones in the figure is as follows. Zone U contains elements still to be moved from the old B to A and zone S receives the elements coming from zone T. Zone R contains the elements already moved from zone T in the 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 the old B to be moved to zone Y in B. Zone Z in B contains the elements received from zone X in the

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 in this order.

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

1. Move $d/2$ elements from zone U (from the end neighbouring zone T) to zone S (to the end neighbouring zone R). If zone U gets empty, the construction of A is completed and A can be used normally thereafter.
2. Move $d/2$ elements from zone V (from the end neighbouring zone X) to zone Y (to the end neighbouring zone Z). If zone V gets empty, the construction of B is completed and it can be used normally thereafter.

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 the reorganization make A or B smaller, both of them can service at least $\lfloor m/2 \rfloor$ operations. Since the work done in a single reorganization is divided for $\lfloor m/d \rfloor$ pop and push operations, there can never be more than one reorganization process active at a time. To represent D, at most three single 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 the old A or the old B was n_0 just before the reorganization started, $n_0 \leq 2n$ at all times since the reorganization is carried out during the next $\lfloor n_0/d \rfloor$ 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 double resizable arrays.

When we want to locate the element with index k in D, we have to consider two cases. First, if there is no reorganization process 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 there is a reorganization process 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 the 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. Clearly, the location requires only a constant number comparisons and arithmetic operations plus an access to a singly resizable array.

5. Deques

The main difference between a doubly resizable array and a deque is that a deque must also support the general modifying operations. Our implemen-

tation of a 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 the reorganization process if there is one that is active. However, this will only give us a linear-time modifying operations.

More efficient working is possible by implementing the blocks as circular arrays as proposed by Goodrich and Kloss II [1999]. 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 constant time. Only an index of 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 indices can be maintained after which replace, insert, and erase operations are all easy to accomplish.

In a 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 single grow 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 erase operation reduces to a single shrink operation. Clearly, the worst-case complexity is asymptotically the same as that for the insert operation.

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

One technical detail omitted so far is the maintenance of the headers. Here we should make sure that the pointers in a header are not initialized repeatedly. To handle this we maintain a pool of three headers and pointers to them. Two of the pointers point always to A and B respectively, and one to the old A or to the old B. Of course, all the headers are initialized once at the object instantiation time. Hereafter, each time a level is freed in a twin-pile the corresponding pointer in the header is also nullified. Hence, when a reorganization is started, we have two empty headers available. Only an update in the pointers to them is required after which they are ready for

use.

Since the blocks are realized as circular arrays, for each full block one new pointer pointing to the current first element of the circular array must be stored in the twin-piles. This will double their size but the extra space needed is still quadratic. There is a constant number of non-full blocks (at the ends of zones R, U, V, and Z). For each of these blocks one more index is used to indicate the location of its last element, but these indices require only a constant amount of extra space.

To summarize, all sequence operations run in $O(1)$ time. If the deque stores n elements, the total number of blocks in A, B, and the 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 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 SGI's implementation for the core deque operations. For reference, we have included the results for SGI's vector in our comparisons where applicable[1].

All benchmarks were carried out on a dual Pentium III with a 933 Mhz processor. The compiler used was `gcc` (version 2.95.2) and the C++ standard library including the SGI STL (version 3.3). All optimizations were enabled during compilation. The timings have been averaged over multiple runs using integer containers of various sizes.

Various benchmarks 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 `push_back`s) as there were elements to be restructured improved the time pr. `push_back` by approximately 10 ns compared to $d = 2$. Increasing d to values higher than 50 did not provide any significant improvement in running times. This indicates that our data structure's memory access patterns do 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.

To determine the cost we pay for space-efficiency, we performed a number of benchmarks for the front and back modifying operations. The best case for our data structure is when neither doubly resizable array becomes empty, since reorganization will then never be initiated. The worst case for one operation occurs when it also executes part of the reorganization. The following table summarizes the results for back modifying operations. The front modifying operations provided similar results and are omitted.

[1] A vector does not support front-modifying operations.

operation	*container*	*time/operation (ns)*
`push_back()`	`std::deque`	85
	`std::vector`	115
	Our deque (no reorganization)	113
	Our deque (with reorganization)	455
`pop_back()`	`std::deque`	10
	`std::vector`	2
	Our deque (no reorganization)	35

The performance of `push_back`-operations for our deque is on par with that of 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. When reorganization is involved, for our deque `push_back` operations are approximately five times slower, compared to SGI's deque. This fits well with the sum of moving four elements and the overhead of doing modifying operations, which is also observed in the general case.

Looking at the `pop_back` operations the first thing to notice is that they are significantly faster than the `push_back` operations, simply because memory deallocation is faster than allocation. However, in this test SGI's deque and vector are approximately 3 and 15 times faster than our deque, respectively. The reason for this is that these two structures do not deallocate memory before 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 vector's entire data memory block.

From these benchmarks the overhead of being space-efficient can be seen. Even if it was possible to reduce the overall overhead for the front and back modifying operations (for instance, through performance engineering), the reorganization would still incur at least a factor two overhead (for $d = 2$) due to the move of elements being necessary.

Accessing a random element in a vector is done simply by adding an offset to a base pointer. Random access to a deque is more complicated. Even though SGI's deque is simple, experiments reported in [Mortensen 2001] indicated that improving access times compared to SGI's deque is possible if we rely on shift operations instead of division and modulus operations. The following table gives the average access times.

operation	*container*	*time/operation (ns)*
`operator[]()`	`std::deque`	210
	`std::vector`	60
	Our deque	150

SGI's deque is approximately 40 percent slower than our deque even though the work done to locate an element in our data structure comprises more operations. 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 [Bentley 2000, Appendix 3]). In our deque, we must calculate for instance $2^{\lfloor k/2 \rfloor}$ (the number of blocks on level k), which can be expressed using left (`<<`) and right (`>>`) shift operations as `1 << (k >> 1)`. Furthermore, because we use circular arrays as our data blocks 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 random access in circular arrays almost as fast as random access in vectors.

The improved time bounds for insert and erase operations achieved by using circular arrays are clearly evident in the benchmarks. The table below gives the results of a test inserting 1000 elements in the container. Results for the erase operation were similar and are omitted.

operation	*# of elements*	*container*	*total time (s)*
1000 `insert()`s	50000	`std::deque`	0.35
		`std::vector`	0.07
		Our deque	0.01
1000 `insert()`s	500000	`std::deque`	9.34
		`std::vector`	6.44
		Our deque	0.03

With 50000 elements in the container before the 1000 insert operations, SGI's deque is 35 times slower than our deque, and SGI's vector is 7 times slower. The difference between $O(n)$ to $O(\sqrt{n})$ is even more clear when n is 500000. SGI's deque and vector are outperformed approximately by a factor 300 and factor 200, respectively.

References

Bentley, Jon. 2000. *Programming Pearls*, 2nd edition. Addison-Wesley, Reading, Massachusetts.

Bojesen, Jesper. 1998. Heap implementations and variations. Written Project, Department of Computing, University of Copenhagen, Copenhagen, Denmark.

Brodnik, Andrej, Carlsson, Svante, Demaine, Erik D., Munro, J. Ian, and Sedgewick, Robert. 1999a. Resizable Arrays in Optimal Time and Space. In *Proceedings of the 6th International Workshop on Algorithms and Data Structures*, Volume 1663 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin/Heidelberg, Germany, 37–48.

Brodnik, Andrej, Carlsson, Svante, Demaine, Erik D., Munro, J. Ian, and Sedgewick, Robert. 1999b. Resizable Arrays in Optimal Time and Space. Technical Report CS-99-09, Department of Computer Science, University of Waterloo, Waterloo, Canada.

Department of Computing, University of Copenhagen. 2000–2001. The Copenhagen STL.

Goodrich, Michael T. and Kloss II, John G. 1999. Tiered Vectors: Efficient Dynamic Arrays for Rank-Based Sequences. In *Proceedings of the 6th International Workshop on Algorithms and Data Structures*, Volume 1663 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin/Heidelberg, Germany, 205–216.

ISO and IEC. 1998. *International Standard ISO/IEC 14882: Programming Languages — C++*. Genève, Switzerland.

Mortensen, Bjarke Buur. 2001. The deque class in the Copenhagen STL: First attempt. Copenhagen STL Report 2001-4, Department of Computing, University of Copenhagen, Copenhagen, Denmark.

Overmars, M. H. 1983. *The Design of Dynamic Data Structures*. Volume 156 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin/Heidelberg, Germany.

Plauger, P. J., Stepanov, Alexander A., Lee, Meng, and Musser, David R. 2001. *The C++ Standard Template Library*. Prentice Hall PTR, Upper Saddle River, New Jersey.

Silicon Graphics, Inc. 1990–2001. Standard Template Library Programmer's Guide.

Stroustrup, Bjarne. 1997. *The C++ Programming Language*, 3rd edition. Addison-Wesley Publishing Company, Reading, Massachusetts.

Sutter, Herb. 1999. Standard Library News, Part 1: Vectors and Deques. *C++ Report 11*, 7, ?–?

Williams, J. W. J. 1964. Algorithm 232: HEAPSORT. *Communications of the ACM 7*, 347–348.

Appendix A. Declaration of the deque class

```
template <typename element, typename allocator = std::allocator<element> >
class deque {
public:

  // type definitions:

  typedef typename allocator::reference reference;
  typedef typename allocator::const_reference const_reference;
  typedef implementation defined iterator;
  typedef implementation defined const_iterator;
  typedef typename allocator::size_type size_type;
  typedef typename allocator::difference_type difference_type;
  typedef element value_type;
  typedef allocator allocator_type;
  typedef typename allocator::pointer pointer;
  typedef typename allocator::const_pointer const_pointer;
  typedef std::reverse_iterator<iterator> reverse_iterator;
  typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

  // construct, deconstruct, and copy operations:

  explicit deque(const allocator& = allocator());
  explicit deque(size_type, const element& = element(), const allocator& = allocator());
  template <typename input_iterator>
  deque(input_iterator, input_iterator, const allocator& = allocator());
  deque(const deque<element, allocator>&);
  ~deque();
  deque<element, allocator>& operator=(const deque<element, allocator>&);
  template <typename input_iterator>
  void assign(input_iterator, input_iterator);
  template <typename size, typename element>
  void assign(size, const element& = element());
  allocator_type get_allocator() const;

  // iterator operations:
```

```
  iterator begin();
  const_iterator begin() const;
  iterator end();
  const_iterator end() const;
  reverse_iterator rbegin();
  const_reverse_iterator rbegin() const;
  reverse_iterator rend();
  const_reverse_iterator rend() const;

  // capacity operations:

  size_type size() const;
  size_type max_size() const;
  void resize(size_type, element = element());
  bool empty() const;

  // sequence operations:

  reference operator[](size_type);
  const_reference operator[](size_type) const;
  reference at(size_type);
  const_reference at(size_type) const;
  reference front();
  const_reference front() const;
  reference back();
  const_reference back() const;
  void push_front(const element&);
  void pop_front();
  void push_back(const element&);
  void pop_back();

  // modifying operations:

  iterator insert(iterator, const element& = element());
  void insert(iterator, size_type, const element&);
  template <typename input_iterator>
  void insert(iterator, input_iterator, input_iterator);
  iterator erase(iterator);
  iterator erase(iterator, iterator);

  // other special operations:

  void swap(deque<element, allocator>&);
  void clear();
};
```

Appendix B. Source code

The source code is organised as follows: `deque.h` (Appendix B.1) and `deque.cpp` (Appendix B.2) contain the interface and implementation of our space-efficient deque, called `tree_deque` in lack of a better name. The implementation relies on a singly resizable array, called `tree_vector` which is found in `tree_vector.h` (Appendix B.3) and `tree_vector.cpp` (Appendix B.4). The interface to the index class (header and twin-pile) is included as

a nested class in `tree_deque` and it is implemented in `dynamic_tree.cpp` (Appendix B.5). Likewise, the data block class (`circular_array`) is nested in `tree_vector` with implementation in `circular_array.cpp` (Appendix B.6). A couple of utility functions have been implemented in separat files, namely logarithm functions (`log2.h`, Appendix B.7), block arithmetic (`block_arith.h`, Appendix B.8), and debug utilities (`debug.h`, Appendix B.9).

Appendix B.1 deque.h

```
/* -*- C++ -*- $Id: deque.h,v 1.1 2002/04/05 15:10:03 jyrki Exp $ */
/*
  interface for tree_deque-class
*/

#ifndef __deque_h__
#define __deque_h__

#include "tree_vector.h"

template <class T, class A = std::allocator<T> >
class tree_deque {
public:
  //typedefs
  typedef typename A::size_type size_type;
  typedef typename A::reference reference;
  typedef size_type iterator;
private:
  size_type nrestructuring_rounds;
  //three vectors is what we need at most
  tree_vector<T, A> tv1;
  tree_vector<T, A> tv2;
  tree_vector<T, A> tv3;
  //and we need some pointers to them
  tree_vector<T,A>* pA;    //front vector
  tree_vector<T,A>* pB;    //back vector
  tree_vector<T,A>* pOld;  //old vector, used when restructuring

  //restructuring
  enum  { A_is_empty = true, B_is_empty = false };
  bool emptied_indicator;
  void prepare_restructure(bool);
  void do_restructure(bool);
public:
  //construct/destroy
  tree_deque();
  ~tree_deque();
  //capacity
  size_type size();
  //iterators
  iterator begin() { return 0; }
  iterator end() { return size(); }
  //access
  reference back();
  reference front();
  reference operator[](size_type);
```

```
  //modifiers
  void pop_front ();
  void pop_back ();
  void push_front(const T&);
  void push_back (const T&);
  iterator insert(iterator, const T&);
  iterator erase(iterator);
  //debug
  void PRINT_INFO();
};


//include implementation
#include "deque.cpp"

#endif
```

Appendix B.2 deque.cpp

```
/* -*- C++ -*- $Id: deque.cpp,v 1.1 2002/04/05 15:10:03 jyrki Exp $ */
/*
  implementation of tree_deque-class
*/

#ifndef __deque_cpp__
#define __deque_cpp__

template <class T, class A>
tree_deque<T,A>::tree_deque()
  : nrestructuring_rounds(0), pA(&tv1), pB(&tv2), pOld(&tv3)
{
}

template <class T, class A>
tree_deque<T,A>::~tree_deque()
{
}

template <class T, class A>
inline void tree_deque<T,A>::prepare_restructure(bool what)
{
  size_type count_A, count_B, rest;
  assert(pOld->real_size() == 0); //if this assertion fails it means that we didn't empty pOld in last
  if (what == A_is_empty) {
    size_type A_size = pB->size() >> 1; //floor(B.size()/2)
    size_type new_B_size = pB->size() - A_size; //ceil(B.size()/2)
    rest = pB->size() % (RESTRUCTURE_MOVE_COUNT * 2);
    count_A = rest / 2;
    count_B = rest - count_A;

    swap(pB, pOld);
    pA->prepare_restructure(A_size);
    pB->prepare_restructure(new_B_size);
  } else { //B_is_empty
```

```
    size_type B_size = pA->size() >> 1; //floor(A.size()/2)
    size_type new_A_size = pA->size() - B_size; //ceil(A.size()/2)
    rest = pA->size() % (RESTRUCTURE_MOVE_COUNT * 2);
    count_B = rest / 2;
    count_A = rest - count_B;
    swap(pA, pOld);
    pB->prepare_restructure(B_size);
    pA->prepare_restructure(new_A_size);
  }
#ifdef RESTRUCTURE_MOVE_COUNT
  //if we move more than one element at a time lets move elements so that
  //the do_restructure phase moves the same amount of elements each time
  //the modulo and division should be cheap if RESTRUCTURE_MOVE_COUNT is 2^x
  for (size_type i = 0; i < count_A; ++i) {
    T& elem = pOld->back();
    pA->internal_push_front(elem);
    pOld->pop_back();
  }
  for (size_type i = 0; i < count_B; ++i) {
    T& elem = pOld->real_front();
    pB->internal_push_front(elem);
    pOld->internal_pop_front();
  }
  nrestructuring_rounds = pB->nphantom_elements / RESTRUCTURE_MOVE_COUNT;
#else
  if (what == A_is_empty) {
    if (pA->nphantom_elements != pB->nphantom_elements) {
      assert(pA->nphantom_elements == pB->nphantom_elements - 1);
      T& elem = pOld->back();
      pB->internal_push_front(elem);
      pOld->pop_back();
    }
  } else {
    if (pB->nphantom_elements != pA->nphantom_elements) {
      assert(pB->nphantom_elements == pA->nphantom_elements - 1);
      T& elem = pOld->back();
      pA->internal_push_front(elem);
      pOld->pop_back();
    }
  }
  nrestructuring_rounds = pA->nphantom_elements;

#endif
  assert(pB->nphantom_elements == pA->nphantom_elements);
  emptied_indicator = what;
  if (nrestructuring_rounds != 0)
    do_restructure(emptied_indicator);
}
template <class T, class A>
inline void tree_deque<T,A>::do_restructure(bool indicator)
{
#ifndef RESTRUCTURE_MOVE_COUNT
  //the simple version, move one at a time
  //figure out which to move to from etc.
  if (indicator == A_is_empty) {
    T& elem = pOld->back();
```

```
    pB->internal_push_front(elem);
    pOld->pop_back();
    elem = pOld->real_front();
    pA->internal_push_front(elem);
    pOld->internal_pop_front();
  } else {
    //B_is_empty
    T& elem = pOld->back();
    pA->internal_push_front(elem);
    pOld->pop_back();
    elem = pOld->real_front();
    pB->internal_push_front(elem);
    pOld->internal_pop_front();
  }
#else
  //The not so simple version. Move RESTRUCTURE_MOVE_COUNT elements each time
  if (indicator == A_is_empty) {
    //A_is_empty
    for (size_type i = 0; i < RESTRUCTURE_MOVE_COUNT; ++i) {
      T& elem = pOld->back();
      pB->internal_push_front(elem);
      pOld->pop_back();
    }
    for (size_type i = 0; i < RESTRUCTURE_MOVE_COUNT; ++i) {
      T& elem = pOld->real_front();
      pA->internal_push_front(elem);
      pOld->internal_pop_front();
    }
  } else {
    //B_is_empty
    for (size_type i = 0; i < RESTRUCTURE_MOVE_COUNT; ++i) {
      T& elem = pOld->back();
      pA->internal_push_front(elem);
      pOld->pop_back();
    }
    for (size_type i = 0; i < RESTRUCTURE_MOVE_COUNT; ++i) {
      T& elem = pOld->real_front();
      pB->internal_push_front(elem);
      pOld->internal_pop_front();
    }
  }
#endif
  //finally, if we're not restructuring any more we should release the last block of pOld. At the momen
  --nrestructuring_rounds;
  if (nrestructuring_rounds == 0) {
    pOld->last_bi.block->release_empty(pOld->last_bi.capacity, pOld->T_alloc);
  }
}


template <class T, class A>
inline void tree_deque<T,A>::pop_front()
{
  if (nrestructuring_rounds != 0) {
    do_restructure(emptied_indicator);
  } else if (pA->is_empty()) {
```

```
    if (pB->size() == 1) {
      //hack to prevent initiating restructure phase with only one element
      //we simply remove the last element
      pB->pop_back();
      return;
    } else {
      prepare_restructure(A_is_empty);
    }
  }
  //pop from A
  pA->pop_back();
}

template <class T, class A>
inline void tree_deque<T,A>::pop_back()
{
  if (nrestructuring_rounds != 0) {
    do_restructure(emptied_indicator);
  } else if (pB->is_empty()) {
    if (pA->size() == 1) {
      //hack to prevent initiating restructure phase with only one element
      pA->pop_back();
      return;
    } else {
      prepare_restructure(B_is_empty);
    }
  }
  //pop from B
  pB->pop_back();
}

template <class T, class A>
inline void tree_deque<T,A>::push_front(const T& elem)
{
  //  if (nrestructuring_rounds != 0) {
  //    do_restructure(emptied_indicator);
  //  }
  //push onto A
  pA->push_back(elem);
}

template <class T, class A>
inline void tree_deque<T,A>::push_back(const T& elem)
{
  //  if (nrestructuring_rounds != 0) {
  //    do_restructure(emptied_indicator);
  //  }
  //push onto B
  pB->push_back(elem);
}

template <class T, class A>
inline tree_deque<T,A>::reference tree_deque<T,A>::back()
{
  if (pB->is_empty()) {
    if (nrestructuring_rounds != 0) {
```

```
      if (emptied_indicator == A_is_empty) {
        return pOld->back();
      } else {
        //B_is_empty
        return pOld->real_front();
      }
    } else {
      return pA->front();
    }
  } else {
    return pB->back();
  }
}

template <class T, class A>
inline tree_deque<T,A>::reference tree_deque<T,A>::front()
{
  if (pA->is_empty()) {
    if (nrestructuring_rounds != 0) {
      if (emptied_indicator == A_is_empty) {
        return pOld->real_front();
      } else {
        //B_is_empty
        return pOld->back();
      }
    } else {
      return pB->front();
    }
  } else {
    return pA->back();
  }
}

template <class T, class A>
inline tree_deque<T,A>::size_type tree_deque<T,A>::size()
{
  return pA->real_size() + pB->real_size() + pOld->real_size();
}

template <class T, class A>
inline tree_deque<T,A>::reference tree_deque<T,A>::operator[](size_type rank)
{
  size_type real_rank;
  if (rank < pA->real_size()) {
    //element is in A
    real_rank = pA->size() - rank - 1;
    return (*pA)[real_rank];
  } else if (rank >= size() - pB->real_size()) {
    //element is in B
    real_rank = rank - (size() - pB->real_size());
    return (*pB)[real_rank];
  } else {
    //element is in Old (only if we are restructuring)
    assert(nrestructuring_rounds != 0);
    if (emptied_indicator == A_is_empty) {
      real_rank = rank - pA->real_size();
```

```
    } else {
      real_rank = pOld->size() - (rank - pA->real_size()) - 1;
    }
    return (*pOld)[real_rank];
  }
}

template <class T, class A>
inline tree_deque<T,A>::iterator
tree_deque<T,A>::insert(iterator rank, const T& elem)
{
  //  if (nrestructuring_rounds != 0) {
  //    do_restructure(emptied_indicator);
  //  }
  size_type real_rank;
  if (rank < pA->real_size()) {
    //element is in A
    real_rank = pA->size() - rank - 1;
    pA->insert(real_rank, elem);
  } else if (rank >= size() - pB->real_size()) {
    //element is in B
    real_rank = rank - (size() - pB->real_size());
    pB->insert(real_rank, elem);
  } else {
    //element is in Old (only if we are restructuring)
    assert(nrestructuring_rounds != 0);
    if (emptied_indicator == A_is_empty) {
      real_rank = rank - pA->real_size();
      pOld->insert(real_rank, elem);
      pA->insert(0, pOld->back());
      pOld->pop_back();
    } else {
      real_rank = pOld->size() - (rank - pA->real_size()) - 1;
      pOld->insert(real_rank, elem);
      pB->insert(0, pOld->back());
      pOld->pop_back();
    }
  }
  return rank;
}

template <class T, class A>
inline tree_deque<T,A>::iterator
tree_deque<T,A>::erase(iterator rank)
{
  if (nrestructuring_rounds != 0) {
    do_restructure(emptied_indicator);
  }
  size_type real_rank;
  if (rank < pA->real_size()) {
    //element is in A
    real_rank = pA->size() - rank - 1;
    pA->erase(real_rank);
  } else if (rank >= size() - pB->real_size()) {
    //element is in B
    real_rank = rank - (size() - pB->real_size());
```

```
    pB->erase(real_rank);
  } else {
    //element is in Old (only if we are restructuring)
    assert(nrestructuring_rounds != 0);
    if (emptied_indicator == A_is_empty) {
      real_rank = rank - pA->real_size();
      pOld->erase(real_rank);
      pOld->push_back(pA->real_front());
      pA->internal_pop_front();
    } else {
      real_rank = pOld->size() - (rank - pA->real_size()) - 1;
      pOld->erase(real_rank);
      pOld->push_back(pB->real_front());
      pB->internal_pop_front();
    }
  }
  return rank;
}

//debug
template <class T, class A>
inline void tree_deque<T,A>::PRINT_INFO()
{
  DEBUG_OUT("tree_deque::PRINT_INFO():\n");
  DEBUG_OUT("pA:\n");
  DEBUG_DO(pA->PRINT_INFO());
  DEBUG_OUT("pB:\n");
  DEBUG_DO(pB->PRINT_INFO());
  DEBUG_OUT("pOld:\n");
  DEBUG_DO(pOld->PRINT_INFO());
  DEBUG_OUT("nrestructuring_rounds = " << nrestructuring_rounds << endl);
}

#endif
```

Appendix B.3 tree_vector.h

```
/* -*- C++ -*- $Id: tree_vector.h,v 1.1 2002/04/05 15:10:03 jyrki Exp $ */

#ifndef __tree_vector_h__
#define __tree_vector_h__

#include <memory>
#include "log2.h"
#include "block_arith.h"

#include "debug.h"

//defines
//MASK: calculates index % bufsize, provided that bufsize is 2^x, for integer x
#define MASK(index, capacity) (index & (capacity-1))

//helper functions and variables
namespace {
```

```
  //apparently (gnu?) c++ does not allow me to have these as class members
  static const unsigned int static_levels = 8;
  static const unsigned int index_min_size = number_of_blocks<static_levels-1>();
}

//forward decl.
template <typename T, typename A>
class tree_deque;

template <typename T, typename A = std::allocator<T> >
class tree_vector {
  friend class tree_deque<T,A>;
private:
  //forward decl.
  class circular_array;
  //typedefs
  typedef typename A::size_type size_type;
  typedef typename A::reference reference;
  typedef A T_allocator;
  typedef circular_array block_type;
  typedef typename T_allocator::rebind<block_type>::other block_allocator;
  typedef typename T_allocator::rebind<block_type*>::other block_pointer_allocator;

  //class circular_array - used as data blocks, each accessed through index
  class circular_array {
  public:
    //construct
    void initialize(size_type, T_allocator&);
    circular_array();
    circular_array(const circular_array&);
    //destroy
    void release_empty(size_type, T_allocator&); //empty blocks
    void release_full(size_type, T_allocator&); //full blocks
    void release(size_type, size_type, T_allocator&); //non-full blocks
    ~circular_array();
    //access
    reference access(size_type, size_type);
    reference front ();
    reference back (size_type, size_type);
    //modifiers
    void insert(size_type, const T&, size_type, size_type, T_allocator&);
    void erase(size_type, size_type, size_type, T_allocator&);
    void push_front(const T&, size_type, size_type, T_allocator&);
    void push_back(const T&, size_type, size_type, T_allocator&);
    void pop_front(size_type, T_allocator&);
    void pop_back(size_type, size_type, T_allocator&);
    //--DEBUG -- should be removed
    reference RAW(size_type);
    void PRINT (size_type, size_type);
    friend ostream& operator<<(ostream& os, const circular_array& ca) {
      return os << "buf: " << ca.buf << ", head: " << ca.head;
    };
  private:
    T* buf;
    size_type head;
```

```
  size_type pos_from_rank(size_type, size_type);
  void move_forward(size_type, size_type, size_type, size_type, T_allocator&);
  void move_back(size_type, size_type, size_type, size_type, T_allocator&);
  //declare assignment private for now
  circular_array* operator=(circular_array);
};

//class dynamic_tree - used as index (consists of header and index tree)
class dynamic_tree {
  friend tree_vector;
private:
  //variables
  static const size_type max_depth = 8 * sizeof(size_type);
  size_type depth;
  size_type phantom_depth;
  size_type nblocks;
  size_type nblocks_below_depth;
  size_type nphantom_blocks_at_phantom_depth;
  bool has_empty_level;
  block_allocator B_alloc;
  block_pointer_allocator Bp_alloc;
  T_allocator T_alloc;

  block_type* static_data;
  block_type** header;
public:
  dynamic_tree();
  ~dynamic_tree();
  //modifiers
  block_type& push_back();
  void pop_back();
  block_type& internal_push_front();
  void internal_pop_front();
  //access
  block_type& front();
  block_type& back();
  block_type& one_before_back();
  block_type& one_after_real_front();
  block_type& real_front(); //when in restructuring phase we use this

  //functions used by tree_vector
  size_type level_of_one_before_back();
  size_type level_of_one_after_real_front();
  size_type level_of_next_block(size_type, size_type);
  block_type& access(size_type level, size_type index) {
    return *(header[level] + index);
  }
  size_type get_depth() {
    return depth;
  }
  size_type size() {
    return nblocks;
  }
  size_type get_phantom_depth() {
    return phantom_depth;
  }
```

```
    size_type real_size() {
      return nblocks - nphantom_blocks;
    }
    //DEBUG
    void PRINT_ARRAYS();
    void PRINT_ALL_ARRAYS();
    void PRINT_HEADER();
    void PRINT_INFO();
    void PRINT_LEVEL(size_type level);

  };

  // block_info - used for storing info on non-full blocks (last, and possibly first block)
  struct block_info {
    size_type size; //how many elements is in it
    size_type capacity;  //how many elements can it store
    block_type* block;   //pointer to the block
    void set(size_type s, size_type c, block_type* b) {
      size = s; capacity = c; block = b;
    }
    bool is_full() {
      return size == capacity;
    }
    bool is_empty() {
      return size == 0;
    }
  };

  //variables
  T_allocator T_alloc;
  size_type nelements;  //total number of elements in tree_vector
  size_type nphantom_elements;  //number of elements that are phantom
  dynamic_tree index;   //our index class
  bool has_empty_block; //do we have an allocated empty extra block
  block_info last_bi;  //info about last block
  block_info first_bi;  //info about first block
  size_type nphantom_elements_first_block; //number of phantom elements in first block
  //restructuring
  void prepare_restructure(size_type);
public:
  //very simple iterator impl.
  typedef size_type iterator;
  iterator begin() { return 0; }
  iterator end() { return nelements; }

  //Construct/destroy/copy
  tree_vector();
  ~tree_vector();
  //capacity
  size_type size() { return nelements; }
  size_type real_size() { return nelements - nphantom_elements; }
  bool is_empty() { return nelements == 0; }
  //Element Access
  reference operator[] (size_type);
  reference front ();
  reference real_front ();
```

```
  reference back ();
  //Modifiers
  void pop_back ();
  void push_back (const T&);
  void internal_push_front(const T&);
  void internal_pop_front();
  iterator insert(iterator, const T&);
  iterator erase(size_type);
  //DEBUG
  void PRINT_INFO();
};

//include implementation
#include "circular_array.cpp"
#include "dynamic_tree.cpp"
#include "tree_vector.cpp"


#endif
```

Appendix B.4 tree_vector.cpp

```
/* -*- C++ -*- $Id: tree_vector.cpp,v 1.1 2002/04/05 15:10:03 jyrki Exp $ */
/* implementation of tree_vector class */

#ifndef __tree_vector_cpp__
#define __tree_vector_cpp__

template <typename T, typename A>
tree_vector<T, A>::tree_vector()
  : T_alloc(), nelements(0), nphantom_elements(0),
    index(), has_empty_block(false),
    nphantom_elements_first_block(0)
{
  last_bi.set(0,0,0);
  first_bi.set(0,0,0);
}

template <typename T, typename A>
tree_vector<T, A>::~tree_vector()
{
  //simple impl of destructor
  //leaving nothing to dynamic_tree's destructor.
  if (has_empty_block) {
    size_type bs = block_size(index.get_depth());
    index.back().release_empty(bs, T_alloc);
    index.pop_back();
  }
  if (!last_bi.is_empty()) {
    last_bi.block->release(last_bi.size, last_bi.capacity, T_alloc);
    index.pop_back();
    if (last_bi.block != first_bi.block) {
      block_type* blockp = &index.back();
      while (blockp != first_bi.block) {
```

```
        size_type bs = block_size(index.get_depth());
        blockp->release_full(bs, T_alloc);
        index.pop_back();
        blockp = &index.back();
      }
      if (first_bi.is_empty()) {
        first_bi.block->release_empty(first_bi.capacity, T_alloc);
      } else if (first_bi.is_full()) {
        first_bi.block->release_full(first_bi.capacity, T_alloc);
      } else {
        first_bi.block->release(first_bi.size, first_bi.capacity, T_alloc);
      }
    }
  }
}

//prepares the tree_vector to grow to a certain size through a number of internal_push_front()'s
template <typename T, typename A>
inline void
tree_vector<T, A>::prepare_restructure(size_type final_size)
{
  //calculate helper values
  size_type last_elem_rank = final_size - 1;
  size_type level = floor_log2(final_size);
  size_type elements_on_level = (1 << level); //2^level
  size_type elements_below_level = elements_on_level - 1;
  size_type level_index = last_elem_rank - elements_below_level;
  size_type bs_exponent = block_size_exponent(level);
  size_type bs = block_size(level);
  size_type block = level_index >> bs_exponent; //=level_index / bs
  size_type block_index = level_index & (bs-1); //=level_index % bs

  //if we have an empty block then release it
  //this block is always the first block,
  //since we are empty when this function is called
  if (has_empty_block) {
    index.header[0]->release_empty(1, T_alloc);
    index.B_alloc.destroy(index.header[0]);
    has_empty_block = false;
  }
  //setup index
  index.depth = level;
  for (size_type i = 0; i < level; ++i)
    index.nblocks_below_depth += block_count(i); //hmm, like to avoid loop
  index.nblocks = index.nblocks_below_depth + block + 1;
  index.phantom_depth = level;
  index.nphantom_blocks_at_phantom_depth = block + 1;

  //allocate level in index
  if (level >= static_levels)
    index.header[level] = index.B_alloc.allocate(block_count(level));
  //create new block
  block_type* b = index.header[level] + block;
  index.B_alloc.construct(b, block_type());
  b->initialize(bs, T_alloc);
```

```
  //setup variables
  nphantom_elements_first_block = block_index+1;
  nelements = final_size;
  nphantom_elements = final_size;
  //setup block info
  first_bi.set(0, bs, b);
  last_bi = first_bi;
}

template <typename T, typename A>
inline void
tree_vector<T, A>::push_back(const T& elem)
{
  bool new_block = ((last_bi.block == first_bi.block) && (first_bi.capacity - nphantom_elements_first_
  if (new_block) {
    if (has_empty_block) {
      has_empty_block = false;
      //update block info
      last_bi.set(0, block_size(index.get_depth()), &index.back());
    } else {
      //create new block
      block_type& b = index.push_back();
      size_type block_size_on_level = block_size(index.get_depth());
      //update block info
      last_bi.set(0, block_size_on_level, &b);
      if (nelements == 0)
        first_bi = last_bi;
    }
  }
  last_bi.block->push_back(elem, last_bi.size, last_bi.capacity, T_alloc);
  //update element counts
  ++last_bi.size;
  if (last_bi.block == first_bi.block)
    ++first_bi.size;
  ++nelements;
}

template <typename T, typename A>
inline void
tree_vector<T, A>::internal_push_front(const T& elem)
{
  //DEBUG_OUT("internal_push_front(" << elem << "):" << endl);
  block_type* b;
  if (nphantom_elements_first_block == 0) {
    //create new block
    b = &index.internal_push_front();
    //update block info
    size_type block_size_on_level = block_size(index.get_phantom_depth());
    first_bi.set(0, block_size_on_level, b);
    nphantom_elements_first_block = block_size_on_level;
  } else {
    b = &index.real_front();
  }
  b->push_front(elem, first_bi.size, first_bi.capacity, T_alloc);
  //update element counts
  ++first_bi.size;
```

```
  if (last_bi.block == first_bi.block)
    ++last_bi.size;
  --nphantom_elements;
  --nphantom_elements_first_block;
  //DEBUG_OUT("\tfirst_bi.size = " << first_bi.size << endl);
  //DEBUG_OUT("end internal_push_front" << endl);
}

template <typename T, typename A>
inline tree_vector<T, A>::reference
tree_vector<T, A>::back()
{
  return last_bi.block->back(last_bi.size, last_bi.capacity);
}

template <typename T, typename A>
inline tree_vector<T, A>::reference
tree_vector<T, A>::real_front()
{
  return first_bi.block->front();
}

template <typename T, typename A>
inline tree_vector<T, A>::reference
tree_vector<T, A>::front()
{
  return index.access(0, 0).front();
}

template <typename T, typename A>
inline void
tree_vector<T, A>::pop_back()
{
  last_bi.block->pop_back(last_bi.size, last_bi.capacity, T_alloc);
  --nelements;
  --last_bi.size;
  if (last_bi.block == first_bi.block)
    --first_bi.size;

  if (last_bi.is_empty() && (last_bi.block != first_bi.block)) {
    if (has_empty_block) {
      //remove empty block
      size_type bs = block_size(index.get_depth());
      index.back().release_empty(bs, T_alloc);
      index.pop_back();
    } else {
      //mark last block as empty
      has_empty_block = true;
    }
    //update block info
    if (nelements == 0) {
      last_bi.set(0,0,0);
      first_bi.set(0,0,0);
    } else {
      block_type& new_lb = index.one_before_back();
      if (&new_lb == first_bi.block) {
```

```
        last_bi = first_bi;
      } else {
        size_type bs = block_size(index.level_of_one_before_back());
        if (last_bi.block == first_bi.block) {
          last_bi.set(bs, bs, &new_lb);
          first_bi = last_bi;
        } else {
          last_bi.set(bs, bs, &new_lb);
        }
      }
    }
  }
}

template <typename T, typename A>
inline void
tree_vector<T, A>::internal_pop_front()
{
  first_bi.block->pop_front(first_bi.capacity, T_alloc);
  ++nphantom_elements;
  ++nphantom_elements_first_block;
  --first_bi.size;
  if (last_bi.block == first_bi.block)
    --last_bi.size;

  if (first_bi.is_empty() && (last_bi.block != first_bi.block)) {
    block_type& b = *first_bi.block;
    b.release_empty(first_bi.capacity, T_alloc);
    index.internal_pop_front();
    //update block info
    if (nelements == 0) {
      last_bi.set(0,0,0);
      first_bi.set(0,0,0);
    } else {
      block_type& new_fb = index.real_front();
      if (&new_fb == last_bi.block) {
        first_bi = last_bi;
        nphantom_elements_first_block = first_bi.capacity - first_bi.size;
      } else {
        size_type bs = block_size(index.get_phantom_depth());
        first_bi.set(bs, bs, &new_fb);
        nphantom_elements_first_block = 0;
      }
    }
  }
}

template <typename T, typename A>
inline tree_vector<T, A>::reference
tree_vector<T, A>::operator[](size_type rank)
{
  size_type level = floor_log2(rank+1);
  size_type elements_on_level = (1 << level); //2^level
  size_type elements_below_level = elements_on_level - 1;
  size_type level_index = rank - elements_below_level;
  size_type bs_exponent = block_size_exponent(level);
```

```
  //  size_type bs = block_size(level);
  size_type bs = 1 << bs_exponent;
  size_type block = level_index >> bs_exponent; //=level_index / bs
  size_type block_index = level_index & (bs-1); //=level_index % bs
  //  assert(block == (level_index / bs));
  //  assert(block_index == (level_index % bs));

  return index.access(level, block).access(block_index, bs);
}

//insert: inserts elem before rank
template <typename T, typename A>
inline tree_vector<T, A>::iterator
tree_vector<T, A>::insert(iterator rank, const T& elem)
{
  size_type level = floor_log2(rank+1);
  size_type elements_on_level = (1 << level); //2^level
  size_type elements_below_level = elements_on_level - 1;
  size_type level_index = rank - elements_below_level;
  size_type bs_exponent = block_size_exponent(level);
  size_type bs = block_size(level);
  size_type block = level_index >> bs_exponent; //=level_index / bs
  size_type block_index = level_index & (bs-1); //=level_index % bs

  //all blocks but last and possibly first are full
  block_type* blockp = &index.access(level, block);
  size_type elements_in_block;
  if (blockp == last_bi.block) {
    elements_in_block = last_bi.size;
  } else if (blockp == first_bi.block) {
    elements_in_block = first_bi.size;
  } else {
    elements_in_block = bs;
  }

  // remove element from block and insert elem in it
  T move_elem = blockp->back(elements_in_block, bs);
  blockp->pop_back(elements_in_block, bs, T_alloc);
  if (block_index == bs - 1) {
    //specal case when inserting as last element in block
    index.access(level, block).push_back(elem, bs-1, bs, T_alloc);
  } else {
    index.access(level, block).insert(block_index, elem, bs-1, bs, T_alloc);
  }

  //move elements towards end
  T next_elem;
  size_type next_bs = bs;
  block_type* next_blockp;

  while (blockp != last_bi.block) {
    size_type next_level = index.level_of_next_block(level, block);
    if (next_level == level) {
      ++block;
    } else {
      block = 0;
```

```
      level = next_level;
      next_bs = block_size(next_level);
    }
    next_blockp = &index.access(level, block);
    size_type next_sz;
    if (next_blockp == last_bi.block) {
      next_sz = last_bi.size;
      if (next_sz == 0) break; //special case: we just need to put move_elem into last block. The final
    } else {
      next_sz = next_bs;
    }
    next_elem = next_blockp->back(next_sz, next_bs);
    next_blockp->pop_back(next_sz, next_bs, T_alloc);
    next_blockp->push_front(move_elem, next_sz-1, next_bs, T_alloc);
    blockp = next_blockp;
    move_elem = next_elem; //this might be expensive and/or non-standard
  }
  push_back(move_elem); //calling 'global' push_back makes sure new data block is created if needed
  return rank;
}

//erase: erases element at rank
template <typename T, typename A>
inline tree_vector<T, A>::iterator
tree_vector<T, A>::erase(size_type rank)
{
  size_type level = floor_log2(rank+1);
  size_type elements_on_level = (1 << level); //2^level
  size_type elements_below_level = elements_on_level - 1;
  size_type level_index = rank - elements_below_level;
  size_type bs_exponent = block_size_exponent(level);
  size_type bs = block_size(level);
  size_type block = level_index >> bs_exponent; //=level_index / bs
  size_type block_index = level_index & (bs-1); //=level_index % bs

  //all blocks but last and possibly first are full
  block_type* blockp = &index.access(level, block);
  size_type elements_in_block;
  if (blockp == last_bi.block) {
    elements_in_block = last_bi.size;
  } else if (blockp == first_bi.block) {
    elements_in_block = first_bi.size;
  } else {
    elements_in_block = bs;
  }

  // remove elem from block
  blockp->erase(block_index, elements_in_block, bs, T_alloc);

  //move elements towards the hole we've just created
  T* next_elem;
  size_type next_bs = bs;
  block_type* next_blockp;
  while (blockp != last_bi.block) {
    size_type next_level = index.level_of_next_block(level, block);
    if (next_level == level) {
```

```
      ++block;
    } else {
      block = 0;
      level = next_level;
      next_bs = block_size(next_level);
    }
    next_blockp = &index.access(level, block);
    next_elem = &next_blockp->front();
    if (blockp == first_bi.block) {
      //special case since first block might not be full
      blockp->push_back(*next_elem, first_bi.size-1, bs, T_alloc);
    } else {
      blockp->push_back(*next_elem, bs-1, bs, T_alloc);
    }
    if ((next_blockp == last_bi.block) && last_bi.size == 1) {
      //special case when we are about to empty last block
      //call pop_back to make sure block is destroyed if needed
      //OK, since pop_back is equvivalent to pop_front when we have 1 element.
      pop_back();
      return rank;
    } else {
      next_blockp->pop_front(next_bs, T_alloc);
    }
    blockp = next_blockp;
    bs = next_bs;
  }
  --last_bi.size;
  if (last_bi.block == first_bi.block)
    --first_bi.size;
  --nelements;
  return rank;
}

//DEBUG
template <typename T, typename A>
inline void
tree_vector<T, A>::PRINT_INFO()
{
  cout << "tree_vector (" << this << "):"
       << endl << "\tnelements: " << nelements
       << endl << "\thas_empty_block: " << has_empty_block
       << endl << "\tfirst_bi: " << first_bi.size << ", " << first_bi.capacity << ", " << first_bi.blo
       << endl << "\tlast_bi: " << last_bi.size << ", " << last_bi.capacity << ", " << last_bi.block
       << endl << "\tnphantom_elements: " << nphantom_elements
       << endl << "\tnphantom_elements_first_block: " << nphantom_elements_first_block
       << endl;
}
#endif
```

Appendix B.5 dynamic_tree.cpp

```
/* -*- C++ -*- $Id: dynamic_tree.cpp,v 1.1 2002/04/05 15:10:03 jyrki Exp $ */
/* implementation of dynamic tree class */
```

```
#ifndef __dynamic_tree_cpp__
#define __dynamic_tree_cpp__

template <typename T, typename A>
tree_vector<T, A>::dynamic_tree::dynamic_tree() :
  depth(0), phantom_depth(0),
  nblocks(0),
  nblocks_below_depth(0), nphantom_blocks_at_phantom_depth(1), //!
  has_empty_level(false),
  B_alloc(), Bp_alloc(), T_alloc(),
  static_data(B_alloc.allocate(index_min_size)),
  header(Bp_alloc.allocate(max_depth))
{
  //initialize first cells to point to the right places in static_data
  header[0] = static_data;
  size_type blocks_below_level_i = 1;
  for (size_type i = 1; i < static_levels; ++i) {
    header[i] = static_data + blocks_below_level_i;
    blocks_below_level_i += block_count(i);
  }
}

template <typename T, typename A>
tree_vector<T, A>::dynamic_tree::~dynamic_tree()
{
  //the pragmatic approach here is to simply deallocate, and not do destroys
  //because we know that tree_vector will call release on all blocks
  if (phantom_depth == depth) {
    //special case when only one level exists
    size_type offset = nphantom_blocks_at_phantom_depth-1;
    size_type end = nblocks - nblocks_below_depth;
    for (size_type i = offset; i < end; ++i)
      B_alloc.destroy(header[depth] + i);
    if (depth >= static_levels) {
      size_type blocks_on_level = block_count(depth);
      B_alloc.deallocate(header[depth], blocks_on_level);
    }
  } else {
    //destroy first level
    size_type blocks_on_low_level = block_count(phantom_depth);
    size_type offset = nphantom_blocks_at_phantom_depth-1;
    for (size_type i = offset; i < blocks_on_low_level; ++i)
      B_alloc.destroy(header[phantom_depth] + i);
    if (phantom_depth >= static_levels)
      B_alloc.deallocate(header[phantom_depth], blocks_on_low_level);

    //destroy all intermediate blocks
    for (size_type i = phantom_depth+1; i < depth; ++i) {
      size_type blocks_on_level = block_count(i);
      for (size_type j = 0; j < blocks_on_level; ++j)
        B_alloc.destroy(header[i] + j);
      if (i >= static_levels)
        B_alloc.deallocate(header[i], blocks_on_level);
    }
    //destroy last level
    if (depth != phantom_depth) {
```

```
      size_type blocks_on_level = nblocks - nblocks_below_depth;
      for (size_type i = 0; i < blocks_on_level; ++i)
        B_alloc.destroy(header[depth] + i);
      if (depth >= static_levels)
        B_alloc.deallocate(header[depth], blocks_on_level);
    }
  }
  B_alloc.deallocate(static_data, index_min_size);
  Bp_alloc.deallocate(header, max_depth);
}

template <typename T, typename A>
inline tree_vector<T, A>::block_type&
tree_vector<T, A>::dynamic_tree::push_back()
{
  size_type blocks_available_this_level = block_count(depth);
  if (blocks_available_this_level + nblocks_below_depth == nblocks) {
    //level is full
    assert(depth < max_depth-1);
    ++depth;
    nblocks_below_depth += blocks_available_this_level;
    if (nblocks >= index_min_size) {
      if (has_empty_level) {
        has_empty_level = false;
      } else {
        header[depth] = B_alloc.allocate(block_count(depth));
      }
    }
  }
  block_type* pos = header[depth] + (nblocks - nblocks_below_depth);
  B_alloc.construct(pos, block_type());
  pos->initialize(block_size(depth), T_alloc);
  ++nblocks;
  return *pos;
}

template <typename T, typename A>
inline tree_vector<T, A>::block_type&
tree_vector<T, A>::dynamic_tree::internal_push_front()
{
  if (nphantom_blocks_at_phantom_depth == 1) {
    //must create new level
    assert(phantom_depth > 0);
    --phantom_depth;
    size_type bc = block_count(phantom_depth);
    nphantom_blocks_at_phantom_depth = bc;
    if (phantom_depth >= static_levels)
      header[phantom_depth] = B_alloc.allocate(bc);
  } else {
    --nphantom_blocks_at_phantom_depth;
  }
  block_type* pos = header[phantom_depth] + nphantom_blocks_at_phantom_depth-1;
  B_alloc.construct(pos, block_type());
  pos->initialize(block_size(phantom_depth), T_alloc);
  return *pos;
}
```

```
template <typename T, typename A>
inline void
tree_vector<T, A>::dynamic_tree::internal_pop_front()
{
  block_type* pos = header[phantom_depth] + nphantom_blocks_at_phantom_depth - 1;
  B_alloc.destroy(pos);
  size_type blocks_on_level = block_count(phantom_depth);
  if (nphantom_blocks_at_phantom_depth == blocks_on_level) {
    //level has been emptied
    if (phantom_depth >= static_levels) {
      B_alloc.deallocate(header[phantom_depth], blocks_on_level);
      header[phantom_depth] = 0;
    }
    ++phantom_depth;
    nphantom_blocks_at_phantom_depth = 1; //!
  } else {
    ++nphantom_blocks_at_phantom_depth;
  }
}


template <typename T, typename A>
inline void
tree_vector<T, A>::dynamic_tree::pop_back()
{
  //  size_type blocks_available_this_level = 1<<(depth>>1);
  block_type* pos = header[depth] + (nblocks - nblocks_below_depth - 1);
  B_alloc.destroy(pos);
  --nblocks;
  if (nblocks_below_depth == nblocks) {
    //level has been emptied
    if (depth >= static_levels) {
      if (has_empty_level) {
        //deallocate empty level
        B_alloc.deallocate(header[depth+1], block_count(depth+1));
        header[depth+1] = 0;
      } else {
        //don't destroy, but mark as empty
        has_empty_level = true;
      }
    }
    if (depth != 0) {
      --depth;
      nblocks_below_depth -= block_count(depth);
    }
  }
}

template <typename T, typename A>
inline tree_vector<T, A>::block_type&
tree_vector<T, A>::dynamic_tree::front()
{
  return *(header[0]);
}
```

```
template <typename T, typename A>
inline tree_vector<T, A>::block_type&
tree_vector<T, A>::dynamic_tree::back()
{
  block_type* pos = header[depth] + (nblocks - nblocks_below_depth - 1);
  return *pos;
}

//returns the level that one_before_back-block exists on
template <typename T, typename A>
inline tree_vector<T, A>::size_type
tree_vector<T, A>::dynamic_tree::level_of_one_before_back()
{
  if (nblocks - nblocks_below_depth == 1) {
    //block is on level below depth
    return depth-1;
  } else {
    return depth;
  }
}

//returns the level that one_after_front-block exists on
template <typename T, typename A>
inline tree_vector<T, A>::size_type
tree_vector<T, A>::dynamic_tree::level_of_one_after_real_front()
{
  if (nphantom_blocks_at_phantom_depth == block_count(phantom_depth)) {
    //block is on level after depth
    return phantom_depth + 1;
  } else {
    return phantom_depth;
  }
}

//returns the actual first block in the structure
//this depends on the number of phantom blocks
template <typename T, typename A>
inline tree_vector<T, A>::block_type&
tree_vector<T, A>::dynamic_tree::real_front()
{
  block_type* pos = header[phantom_depth]  + nphantom_blocks_at_phantom_depth - 1;
  return *pos;
}


//returns the level of the block that follows the block identified by (level, index)
template <typename T, typename A>
inline tree_vector<T, A>::size_type
tree_vector<T, A>::dynamic_tree::level_of_next_block(size_type level, size_type index)
{
  if (index == block_count(level) - 1) {
    return level + 1;
  } else {
    return level;
  }
}
```

```
template <typename T, typename A>
inline tree_vector<T, A>::block_type&
tree_vector<T, A>::dynamic_tree::one_before_back()
{
  if (nblocks - nblocks_below_depth == 1) {
    //block is on level below depth
    block_type* pos = header[depth-1] + (block_count(depth-1)) - 1;
    return *pos;
  } else {
    block_type* pos = header[depth] + (nblocks - nblocks_below_depth - 2);
    return *pos;
  }
}

template <typename T, typename A>
inline tree_vector<T, A>::block_type&
tree_vector<T, A>::dynamic_tree::one_after_real_front()
{
  if (nphantom_blocks_at_phantom_depth == block_count(phantom_depth)) {
    //block is on level after phantom_depth
    block_type* pos = header[phantom_depth+1];
    return *pos;
  } else {
    block_type* pos = header[phantom_depth] + nphantom_blocks_at_phantom_depth - 1;
    return *pos;
  }
}

//----------------- DEBUG ------------------------
template <typename T, typename A>
void
tree_vector<T, A>::dynamic_tree::PRINT_ARRAYS()
{
  for (size_type i=phantom_depth; i < depth+1; ++i)
    PRINT_LEVEL(i);
  DEBUG_OUT(std::endl);
}

template <typename T, typename A>
void
tree_vector<T, A>::dynamic_tree::PRINT_ALL_ARRAYS()
{
  for (size_type i=0; (i < max_depth) && (header[i] != 0); ++i)
    PRINT_LEVEL(i);
  DEBUG_OUT(std::endl);
}

template <typename T, typename A>
void
tree_vector<T, A>::dynamic_tree::PRINT_HEADER()
{
  DEBUG_OUT("header: pointer, distance to last header pointer\n");
  DEBUG_OUT("header[0]: " << header[0] << std::endl);
  for (size_type i = 1; i< max_depth; ++i) {
    DEBUG_OUT("header[" << i <<"]: " << header[i]
```

```
              << ", " << header[i]-header[i-1] << std::endl);
  }
}

template <typename T, typename A>
void
tree_vector<T, A>::dynamic_tree::PRINT_INFO()
{
  DEBUG_OUT("header: " << max_depth << " levels = "
            << sizeof(header) * max_depth << "b\n");
  DEBUG_OUT("static: " << static_levels
            << " levels -> index_min_size = " << index_min_size << " = "
            << sizeof(static_data)*index_min_size << "b"  << std::endl);
  DEBUG_OUT("depth = " << depth << std::endl);
  DEBUG_OUT("nblocks = " << nblocks << std::endl);
  DEBUG_OUT("nblocks_below_depth = " << nblocks_below_depth << std::endl);
  DEBUG_OUT("phantom_depth = " << phantom_depth << std::endl);
  DEBUG_OUT("has_empty_level = " << has_empty_level << std::endl);
}

template <typename T, typename A>
void
tree_vector<T, A>::dynamic_tree::PRINT_LEVEL(size_type level)
{
  DEBUG_OUT("level " << level << ": ");
  DEBUG_OUT((level < static_levels? "in static_data " : ""));
  DEBUG_OUT((level > depth? "above depth" : ""));
  DEBUG_OUT(std::endl);
  for (size_type i = 0; i < block_count(level); ++i) {
    DEBUG_OUT(*(header[level] + i) << "  ");
  }
  DEBUG_OUT(std::endl);
}

#endif
```

Appendix B.6 circular_array.cpp

```
/* -*- C++ -*- $Id: circular_array.cpp,v 1.1 2002/04/05 15:10:03 jyrki Exp $ */
/*
   implementation of circular_array class
*/

#ifndef __circular_array_cpp__
#define __circular_array_cpp__

//----------- construct/destroy/copy
template <typename T, typename A>
inline
void tree_vector<T, A>::circular_array::initialize(size_type capacity,
   T_allocator& alloc)
{
  buf = alloc.allocate(capacity);
  //buf = (T*)malloc(capacity * sizeof(T));
}
```

```
template <typename T, typename A>
inline
tree_vector<T, A>::circular_array::circular_array()
{
  //dummy constructor.
  //Initialize must be called manually after constructor
}

template <typename T, typename A>
inline
tree_vector<T, A>::circular_array::circular_array(const circular_array& other)
{
  //dummy copy constructor (needed because of the way allocators work)
  //the assumption is here that the copy constructor is always called with an empty other
  //after "copy construction" initialize must be called
}

//release functions:
//one for empty blocks, one for full blocks and one for non-full
template <typename T, typename A>
inline
void tree_vector<T, A>::circular_array::release_empty(size_type capacity,
      T_allocator& alloc)
{
  //assumption: block is empty
  alloc.deallocate(buf, capacity);
}

template <typename T, typename A>
inline
void tree_vector<T, A>::circular_array::release_full(size_type capacity,
     T_allocator& alloc)
{
  //assumption: block is full
  for (T* p = buf; p != buf + capacity; ++p)
    alloc.destroy(p);

  alloc.deallocate(buf, capacity);
}

template <typename T, typename A>
inline
void tree_vector<T, A>::circular_array::release(size_type size, size_type capacity,
T_allocator& alloc)
{
  //assumption: block is non-full and non-empty
  //  assert(size != 0);
  size_type tail = MASK(head + size - 1, capacity);
  if (head > tail) {
    for (T* p = buf + head; p !=  buf + capacity; ++p)
        alloc.destroy(p);
      for (T* p = buf; p !=  buf + tail+1; ++p)
        alloc.destroy(p);
    } else {
      for (T* p = buf + head; p != buf + tail+1; ++p)
```

```
        alloc.destroy(p);
    }
  alloc.deallocate(buf, capacity);
}

template <typename T, typename A>
inline
tree_vector<T, A>::circular_array::~circular_array()
{
  //dummy destructor, since we don't have info on size and buffer space
  //release should be called manually before destructor is called
}

//------------------ helper functions
template <typename T, typename A>
inline tree_vector<T, A>::size_type
tree_vector<T, A>::circular_array::pos_from_rank(size_type rank, size_type capacity)
{
  return MASK(head + rank, capacity);
}

//moves all elements from start to end (both incl.) dist cells forward
template <typename T, typename A>
inline void
tree_vector<T, A>::circular_array::move_forward(size_type start, size_type end,
                                                size_type dist,
                                                size_type capacity,
                                                T_allocator& alloc)
{
  for (size_type i = end; i != (start-1); --i) {
    alloc.construct(buf + pos_from_rank(i+dist, capacity),
                    buf[pos_from_rank(i, capacity)]);
    alloc.destroy(buf + pos_from_rank(i, capacity));
  }
}

//moves all elements from start to end (both incl.) dist cells back
template <typename T, typename A>
inline void
tree_vector<T, A>::circular_array::move_back(size_type start, size_type end,
                                             size_type dist, size_type capacity,
                                             T_allocator& alloc)
{
  for (size_type i = start; i <= end; ++i) {
    alloc.construct(buf + pos_from_rank(i + capacity - dist, capacity),
                    buf[pos_from_rank(i, capacity)]);
    alloc.destroy(buf + pos_from_rank(i, capacity));
  }
}


//---------- element access
template <typename T, typename A>
inline tree_vector<T, A>::reference
tree_vector<T, A>::circular_array::access(size_type rank, size_type capacity)
{
```

```
  return buf[pos_from_rank(rank, capacity)];
}

template <typename T, typename A>
inline tree_vector<T, A>::reference
tree_vector<T, A>::circular_array::front ()
{
  return buf[head];
}

template <typename T, typename A>
inline tree_vector<T, A>::reference
tree_vector<T, A>::circular_array::back (size_type size, size_type capacity)
{
  size_type tail = MASK(head + size - 1, capacity);
  return buf[tail];
}


//---------- modifiers
template <typename T, typename A>
inline void
tree_vector<T, A>::circular_array::insert(size_type rank, const T& elem,
  size_type size, size_type capacity,
  T_allocator& alloc)
{
  assert(size != capacity); //make sure there is room

  if (size > 0) {
    //    DEBUG_OUT("insert " << elem << " at rank " << rank << endl);
    assert((0 <= rank) && (rank < size));
    if (rank == 0) { //insertion in front
      head = MASK(head + capacity - 1, capacity);
      alloc.construct(buf + head, elem);
    } else {
      if (rank < size - rank) {
        //fewer elements before rank
        move_back(0, rank - 1, 1, capacity, alloc);
        head = MASK(head - 1, capacity);
      } else {
        //fewer elements after rank
        move_forward(rank, size-1, 1, capacity, alloc);
      }
      alloc.construct(buf + pos_from_rank(rank, capacity), elem);
    }
  } else {
    assert(0 == rank);
    alloc.construct(buf, elem);
    head = 0;
  }
  //  DEBUG_DO(PRINT(size+1, capacity));
}

template <typename T, typename A>
inline void
tree_vector<T, A>::circular_array::erase(size_type rank, size_type size,
```

```
 size_type capacity, T_allocator& alloc)
{
  assert((0 <= rank) && (rank < size));
  alloc.destroy(buf + pos_from_rank(rank, capacity));
  if (rank == 0) { //erase from front
    head = MASK(head + 1, capacity);
  } else {
    if (rank < size - rank) {
      //fewer elements before rank
      move_forward(0, rank - 1, 1, capacity, alloc);
      head = MASK(head + 1, capacity);
    } else {
      //fewer elements after rank
      move_back(rank+1, size-1, 1, capacity, alloc);
    }
  }
}

template <typename T, typename A>
inline void
tree_vector<T, A>::circular_array::push_front(const T& elem, size_type size,
      size_type capacity,
      T_allocator& alloc)
{
  insert(0, elem, size, capacity, alloc);
}

template <typename T, typename A>
inline void
tree_vector<T, A>::circular_array::push_back(const T& elem, size_type size,
     size_type capacity, T_allocator& alloc)
{
  assert(size != capacity); //make sure there is room
  if (size == 0) {
    alloc.construct(buf, elem);
    head = 0;
  } else {
    size_type tail = MASK(head + size, capacity);
    alloc.construct(buf + tail, elem);
  }
}

template <typename T, typename A>
inline void
tree_vector<T, A>::circular_array::pop_front(size_type capacity, T_allocator& alloc)
{
  alloc.destroy(buf + head);
  head = MASK(head + 1, capacity);
}

template <typename T, typename A>
inline void
tree_vector<T, A>::circular_array::pop_back(size_type size, size_type capacity,
    T_allocator& alloc)
{
  size_type tail = MASK(head + size - 1, capacity);
```

```
  alloc.destroy(buf + tail);
}

// DEBUG
template <typename T, typename A>
tree_vector<T, A>::reference
tree_vector<T, A>::circular_array::RAW(size_type i)
{
  return buf[i];
}

template <typename T, typename A>
void
tree_vector<T, A>::circular_array::PRINT(size_type size, size_type capacity)
{
  cout << this << ": buf=" << buf << ", c=" << capacity << ", s=" << size << ", h="<< head << endl;
  cout << "  sequential: ";
  for (size_type i = 0; i < size; i++)
    cout << buf[pos_from_rank(i, capacity)] << " ";
  cout << endl;
}

#endif
```

Appendix B.7 log2.h

```
/* $Id: log2.h,v 1.1 2002/04/05 15:10:03 jyrki Exp $ */
/*
 * floor_log2: function for calculating floor(lg(x))
 *
 */
#ifndef __log2_h__
#define __log2_h__


#ifdef __GNUC__
//if we are using gnu cc, then we have a library function.
//I believe it works on all types
#include <cmath>
template <typename T, int size>
inline T internal_floor_log2(T x)
{
  return ilogb(x);
}

#else
//otherwise use Sofus' binary search-like impl.
//works for 32-bit integers
//NOTE: does NOT specialize as wanted for 32 bits
template <typename T, int size>
T internal_floor_log2(T x) {
  static const unsigned char log_table[256] = {
    0xff, // <--- rogue value
    0,
    1, 1,
```

```
    2, 2, 2, 2,
    3, 3, 3, 3, 3, 3, 3, 3,
    4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
    5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
    5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
    6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
    6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
    6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
    6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
    7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
    7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
    7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
    7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
    7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
    7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
    7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
    7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7
  };
  long rv = 0;
  if (x & 0xffff0000) { rv += 16; x >>= 16; }
  if (x & 0xff00) { rv += 8; x >>= 8; }
  if (x & 0xf0) { rv += 4; x >>= 4; }
  if (x & 0xc) { rv += 2; x >>= 2; }
  if (x & 0x2) { rv += 1; x >>= 1; }
  return rv + log_table[x];
}
#endif

template <typename T>
inline T floor_log2(T x)
{
  return internal_floor_log2<T, sizeof(T)>(x);
}


#endif
```

Appendix B.8 block_arith.h

```
/* -*- C++ -*- $Id: block_arith.h,v 1.1 2002/04/05 15:10:03 jyrki Exp $ */

#ifndef __block_arith_h__
#define __block_arith_h__

namespace {
  //block size arithmetic
  inline unsigned int block_size_exponent(unsigned int level) {
    return (level + 1) >> 1; //ceil(level/2) == floor((level+1)/2)
  }

  inline unsigned int block_size(unsigned int level) {
    return 1 << ((level + 1) >> 1);
  }

  inline unsigned int block_count_exponent(unsigned int level) {
```

```
    return level >> 1; //floor(level/2)
  }

  inline unsigned int block_count(unsigned int level) {
    return 1 << (level >> 1);
  }

  // number of data blocks for a level
  template <unsigned int level>
  inline unsigned int number_of_blocks() {
    return number_of_blocks<level-1>() + block_count(level);
  }
  template <>
  inline unsigned int number_of_blocks<0>() {
    return 1;
  }

}
#endif
```

Appendix B.9 debug.h

```
/* -*- C++ -*- $Id: debug.h,v 1.1 2002/04/05 15:10:03 jyrki Exp $ */

#ifndef __debug_h__
#define __debug_h__

// debug macros
#if defined(__DEBUG__)
#undef DEBUG_OUT
#undef DEBUG_DO
#define DEBUG_OUT(x) std::cout << x
#define DEBUG_DO(x) x
#else
#undef DEBUG_OUT
#undef DEBUG_DO
#define DEBUG_OUT(x) //void
#define DEBUG_DO(x) //void
#endif

#endif
```