The deque class in the Copenhagen STL: First attempt

Bjarke Buur Mortensen
Ålborggade 22, 1. th.
2100 Copenhagen East
Denmark
`rodaz@diku.dk`

Abstract. This report presents work done to implement the `deque` class for the Copenhagen STL. The ideas of two articles are explored and combined in an attempt to implement a data structure that improves the running times of the core `deque` operations compared to an existing implementation. Even though the theoretical time requirements are achieved, benchmarks show that the data structure performs poorly compared to the existing `deque` implementation in the operations concerned with growing and shrinking the data structure at the ends. Considering that these operations are key operations of a `deque`, the measured performance makes the implemented `deque` an unappealing choice for the Copenhagen STL `deque` class. However, the implemented `deque` offers substantial improvements of the random element access operations, and greatly outperforms the `deque` to which it is compared in operations that inserts and erase elements. It is recommended that an implementation combining the work of this project and the existing deque implementation is pursued, so as to put the benefits of both implementations into the Copenhagen STL.

Keywords. deque, resizable arrays

1. Introduction

The original working title for this project was "Efficient random access data structures in C++". In the C++-standard there are two such sequences, namely `vector` and `deque` [4, 23.2.4 23.2.1] . The `vector` class is intended to be a replacement for the classic C-arrays, and in addition to be dynamically resizeable. Since the array is probably the most widely used data structure in most programs it was intriguing to try to improve this data structure. However, because of the aim of being a replacement for static arrays, a `vector` implementation is forced to contain its elements ordered by rank in a contiguous block of memory[1]. This allows the vector to be used with old C and C++-functions that expect to operate on an array. Unfortunately, this constraint also implies its implementation.

The choice for exploration thus became the `deque` data structure.

[1] The first edition of the C++-standard did not specify this, but the issue has been identified as a C++ Standard Library Defect, meaning that the next edition will include the requirement. For details, see [5, #69].

1.1 The Deque data structure

The name *deque*[2] stands for "double-ended queue". This means that a `deque` is a sequence that can grow and shrink in both ends efficiently, i.e. in constant time. In addition a `deque` also supports random access to any element. Random access implies that any element can be retrieved in constant time.

1.1.1 C++-standard requirements

The C++-standard requires that the front and back modifying operations (also referred to as grow and shrink operations), i.e. `push_front`(), `push_back`(), `pop_front`() and `pop_front`() always take constant time. This means that the time-complexity must be worst-case constant time, or $O(1)$, and not, as in the case of `vector`, amortized constant time. Likewise random access, i.e. the `operator`[] and `at`()-functions, must run in $O(1)$-time. Inserting and erasing in the middle of the data structure, i.e. `insert`() and `erase`()-operations, can take $O(n)$ time when n is the number of elements in the `deque`[3].

The front/back-operations, the insert/erase-operations and the random access-operation can be considered the core operations of the `deque` class, as specified by the C++-standard interface. I have focused on implementing these so that they conform to the interface.

In addition to these operations the standard specifies other interface-requirements that an implementation must conform to. I have left out these other parts of the interface. Furthermore, the standard imposes certain requirements on the validity of iterators and element references, depending on which modifier is applied to the `deque`. I have not addresses these issues in this project. They are left as future work (see section 6.1).

It is worth noting that the C++-standard does not impose any bounds on the space consumption of the data structure. However, this project has focused on minimizing the space overhead.

1.2 Project context: The Copenhagen STL

This report is written in the context of the Copenhagen STL project. The Copenhagen STL Project aims at improving existing implementations of STL-components or replacing components with alternative versions that provide better time and/or space performance. The goal is to provide a full implementation of the STL-part of the C++-standard.

Given the context of a project that aims at implementing a C++ library, this report should be considered as a documentation of the efforts made in implementing a `deque`-data structure that fits this library, rather than an exploration of the theoretic aspects of implementing a `deque`. Thus, the project has focused on combining the ideas of two articles into an actual `deque`-implementation ([3], [1]).

The resulting source code is available through the Copenhagen STL homepage [2], and as appendix Appendix B

[2] pronounced "deck"

[3] More precisely, the bound is $O(\min(r, n-r))$, where r is the rank at which to insert or delete an element.

1.3 Outline

The rest of the report is organized as follows. Section 2 gives the background for the work done in this project by describing the implementation of the `deque` class in SGI STL and outlining the ideas of the two articles studied. In section 3 the ideas of the articles are studied in detail and their use in the implemented `deque` is described section . Section 4 makes comments on the details of the `deque` implementation, and section 5 reports on the findings in a series of benchmark runs. Section 6 concludes the report.

2. Inspiration

The `deque` class is a versatile data structure, providing efficient resizability from both ends and random access to any element. Therefore the implementation becomes more complex than that of vector. To my knowledge any standards-compliant version of vector is implemented, so that whenever the current memory block's capacity is reached, a new block of size proportional to the current size is allocated[4], and the elements are relocated to this new block. This gives the `vector` an amortized constant time bound for grow and shrink-operations.

Clearly an implementation like this cannot be used for the `deque` class, since the grow and shrink operations must be worst case constant time. Instead, the implementations considered here add a level of indirection to element access by using an index block which points to a number of data blocks, which in turn contain a fraction of the elements. This is true for the SGI implementation (section 2.1), and the ideas presented in the two articles studied (section 2.2).

2.1 Existing deque-implementations

The STL implementation shipped with the widely used gcc-compiler is the STL of SGI [6]. The implementation of `deque` in the SGI STL uses a number of **fixed** size data blocks, called *nodes*, and an index block, called a *node map*. Whenever a new element is appended at either end, the element is appended to the relevant outermost data block. Clearly this takes constant time on the worst case. If a data block is full a new one is allocated and a pointer to it is appended to the index block. The element is then inserted in this new block. This also takes worst case constant time.

However, to the best of my knowledge, the SGI `deque` is not fully standards-compliant. Whenever the index block becomes full, and no more data blocks can be appended, the index block is reallocated and the node pointers are moved to the middle of the new block. This operation takes time $O(n/b)$, where b is the fixed size of the data blocks, since there are n/b data blocks. This cost is amortized over the number of elements already inserted in the data structure. Thus

> "Inserting an element at the beginning or end of a `deque` takes amortized constant time." [6, Deque description]

, and not, as stipulated by the standard, worst case constant time.

The space requirements for the data blocks is $O(n)$. All data blocks except the outermost ones are always full, meaning that only a constant amount of space is wasted in the data blocks. The size of the index block, however, is linear in n, since

[4] Typically, this block has double the capacity of the existing block

it must store n/b pointers. Thus the amount of extra space required at any time to store n elements is $O(n)$.

The work done in this project addresses both the worst case constant time grow/shrink-operations and the space issue.

2.2 Earlier work

This section briefly summarizes the results of Goodrich and Kloss [3] and Brodnik et al. [1]. Their ideas are explored in detail in section 3, since they constitute the building blocks of the work done in this project.

Goodrich's and Kloss' article concerns the vector data type and in particular the Java vector class. Thus they are not concerned with the contiguous memory requirements of the C++ standard. Their aim is to improve the time bound for insert and erase operations while at the same time providing constant time random access. To this end they devise a general k-level data structure. Their basic building block, the 1-level Tiered Vector, is a circular sequence (one block of memory), that allows inserts and erases from the ends in constant time (provided that it does not need to grow) while at the same time allowing random access to the elements in constant time.

When $k = 2$ their data structure resembles the SGI implementation with an index block and a number of data blocks. However, they show that for $k = 2$ the extra space requirements for their data structure is only $O(\sqrt{n})$, because they make sure that all data blocks are of size $O(\sqrt{n})$. This way there are at any time only $O(\sqrt{n})$ data blocks and the size of the index block is thus bounded by this.

This property and the fact that all data blocks are circular arrays allow them to implement insert and erase-operations in $O(\sqrt{n})$-time. The time bounds are, however, only amortized, since they address resizing of the index block and of the data blocks only when the data structure is full or when n drops below a given level.

The article by Brodnik et al. can be seen as improving the results of Goodrich and Kloss by deamortizing their time bounds. By deamortization we mean the process of doing a constant amount of the work that needs to be done later, each time an operation to which the time bound is amortized is used. For instance, if growing a data structure at some time will effect that its memory must be reallocated and the elements moved to the new location, a constant amount of work of this relocation can be done each time the grow operation is invoked. Clearly, since the relocation can be amortized it suffices to do a constant amount of work in each grow-operation. The space requirements, however, become bigger since the memory block to copy to must be allocated when deamortization begins.

They use this technique to achieve worst case $O(1)$ time bounds for back and front operations, while at the same time assuring that all data blocks are of size $O(\sqrt{n})$, and that the index block can be relocated in constant time.

3. 2-level deque

Since the `deque` described by Brodnik et al. conforms to the requirements of the standard, the data structure in this project will build primarily on their idea. However, since Goodrich and Kloss have shown how to achieve faster insert times, something not considered by Brodnik et al., we will use the circular sequence of Goodrich and Kloss as a basic building block for our `deque` data structure. This will give us a data structure that fulfills the `deque` standard requirements while at the same time improving the space bounds (Brodnik et al) and allowing faster insert and erase operations (Goodrich and Kloss).

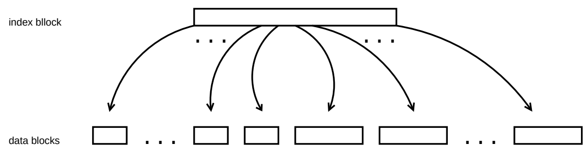
3.1 Overview

The basic idea of Brodnik et al. is to keep all blocks roughly at size $O(\sqrt{n})$, as is the case for Goodrich and Kloss. However, in their data structure they operate with two block sizes, small blocks of size s_1 and large blocks of size s_2. The values for these are given thus

$$s_1 \quad = \quad 2^{\left\lfloor \frac{\log_2(1+n)}{2} \right\rfloor} \quad and \quad s2 \quad = \quad 2^{1+\left\lfloor \frac{\log_2(1+n)}{2} \right\rfloor}$$

These numbers have the properties that whenever n becomes $2^x - 1$, for some even integer x, they double in size. That is, for instance, for $n = 63$, s_1 equals 8, which is $O(\sqrt{n})$ and s_2 equals 16, which is $O(2\sqrt{n})$. When n reaches 255, s_1 becomes 16 and s_2 becomes 32. More generally, when $1 + n$ grows by a factor of 4, s_1 and s_2 grows by a factor of 2.

The data structure must at all times uphold the invariants that only two block sizes exist, and that all small blocks are consecutive followed by all the large blocks which are consecutive. The following figure illustrates these properties.



The invariants are used to obtain the required time bounds for the basic operations. Work done in the grow and shrink-operations make sure that these invariants are maintained, as described in section 3.4.

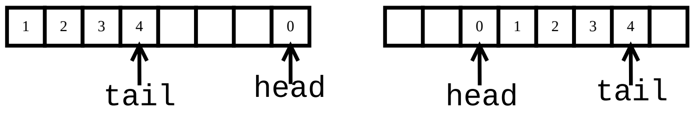
As in the SGI `deque`, only the outermost blocks are allowed to be non-full. Since all blocks are of size $O(\sqrt{n})$, only $O(\sqrt{n})$ space is wasted in the data blocks. Furthermore, since there are only $O(\sqrt{n})$ data blocks the index block will at no time take up more than $O(\sqrt{n})$ space. The amount of extra storage needed at any time is thus always $O(\sqrt{n})$.

To achieve fast insert and erase times all the data blocks will be circular sequences, as suggested by Goodrich and Kloss. Likewise, the index block will be maintained as a circular sequence, so I will describe this data structure first (section 3.2). Then I will describe how to implement an index block that can grow and shrink in worst case constant time, which we require for constant time operations (section 3.3). Sections 3.4 through 3.6 will address how to achieve the required time bounds for the basic operations, random access and grow and shrink, and how to achieve better than standard requirements time bounds on insert and erase operations.

3.2 Circular sequence

The basic building block of the `deque` data structure is the circular sequence described by Goodrich and Kloss. Because of its properties it is used for implementing both the index block and the data blocks. The sequence is basically a block of memory, and two pointers, a head and a tail, that point into the memory block. The use of the term circular in this context means that the sequence uses its memory in a circular fashion. It does not mean that element access is circular in a way that asking for the element after the last gives the first element.

The basic layout of the sequence is illustrated in the below figure. The elements of the block are ordered sequentially in the block modulo the block's capacity. This means that the two layouts below are possible.



The operations on the sequence are the following.

Element access Mapping from a rank r to the actual position p of the element in the memory block is done simply by a modulus-operation. Thus $p = (h+r) \bmod c$, where h is the head pointer and c is the capacity of the memory block. Clearly, this takes worst case $O(1)$-time.

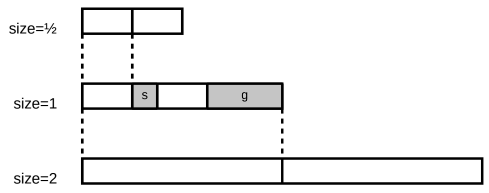
Front and back-operations Inserting elements in the front and the back of the block is also simple. In the front simply place the new element just before the head pointer and update the head-pointer to point to the new element. In the back, place the element just after the tail pointer and update the tail pointer to point to the new element. Likewise, deleting in the front and back is a matter of removing the element and updating either the head or tail pointer. These operations all take constant time.

Insert and erase To insert an element at rank r we first make room for the element by moving either the r elements before r one place back $(\bmod c)$ or moving the $n - r$ elements after r one place forward $(\bmod c)$, depending on which side of r the fewest elements are stored. This way we have opened up a slot for the element we want to insert. Likewise, when erasing an element we close up the slot belonging to the erased element by moving all elements on one of the sides towards the slot. Inserts and erases can thus be done in $O(\min(r, n - r))$-time.

Inserting into the structure can only be done as long as there are empty slots in the memory block.

3.3 Index block

The requirements of the index block is that it must be able to be expanded and shrunk in worst case constant time. In other words, we need to avoid a time consuming relocation of elements when the index block runs full. The idea suggested by Brodnik et al. is that whenever we add a new element to our block, we copy a constant number of elements to an index block of double size. When we remove an element, we copy a constant number of elements to an index block of half size. When the index block becomes full or $\frac{1}{4}$ full we switch to larger or smaller blocks, respectively. For our purposes, the structure is a bit more complicated than that of Brodnik et al. (they discuss it only in the context of singly resizable arrays). First, since we must be able to add data block pointer to both ends, we need to maintain the index block as a circular sequence. Secondly, we need the concept of a *copy range*, to keep control of what elements we have copied to either the smaller or the larger block. The following figure illustrates the index block used in this project.



The grey areas indicate when we do part of the relocation work. In this particular version we use a resolution of $\frac{1}{8}$, but this is arbitrarily chosen and could be something else. When the size of the index block falls below $\frac{3}{8}$ full (the area marked by s for *shrink*), we create the smaller index block and copies 2 elements from the leftmost white zone to the smaller block. This is done as long as we do shrink operations in the s-zone. When the index block becomes $\frac{2}{8}$ full all elements left will have been copied to the smaller block (since the size of s is $\frac{1}{8}$, and $\frac{1}{8} * 2 = \frac{2}{8}$), and we can switch to using this block. Observe that the leftmost zone does not represent the first elements of the block, but rather the $\frac{2}{8}$ elements that are left when we shrink the data structure, since we can remove elements from both ends.

When we enter the zone marked g (for *grow*), we create the larger block and copy three elements to the larger block and this is repeated every time we add an element while in the g-zone. The size of g is $\frac{3}{8}$, so $\frac{3}{8} * 3 \geq \frac{8}{8}$ means that all elements have been copied to the larger block when the block becomes full. Thus, we switch to using the new larger block.

The copy range is maintained as follows: when we first enter one of the grey zones, we copy the two or three middle elements of the block to the relevant copy block. Whenever we do a shrink or grow we grow the copy range from the middle, since elements can be inserted and deleted from both ends. The rationale behind this heuristic is that we want to minimize the number of times we need to erase an element from the copy blocks.

Observe that the index block as described here fullfills all the requirements of a C++-`deque`, which makes it possible to implement a `deque` using this data structure alone. However, the space requirements for this structure can be as high as $4.8n$ (when block is $\frac{5}{8}$-full we use $\frac{8}{8}$for the primary block and $\frac{16}{8}$for the larger copy). Furthermore we must make sure that the possibly two copies of an element are both updated, when an element is updated. This becomes complex.

For our purposes, i.e. using it only to store pointers to data blocks, it suffices to make sure that whenever we remove a block from either end, we also remove it from the relevant copy block, in case the element has been copied there.

3.4 Grow and shrink-operations

As mentioned in section 3.1 work is done in the grow and shrink operations to make sure that the blocks are always kept at size $O(\sqrt{n})$.

The idea is that just before we create a new data block, we join the two small blocks adjacent to the first large block. When we delete an empty data block we split the first large into to small blocks. When this is done we check whether n has fallen below its shrink limit or risen above its grow limit. If this is the case we update the block sizes s_1 and s_2.

3.4.1 Maintaining the invariant

We need to show that whenever we update the values of s_1 and s_2 all the data blocks have the same size. This way we ensure that when we insert a new data block using the new value of either s_1 and s_2, we will not violate the invariant of always having only two block sizes.

At the time we grow the block sizes from $s_1 \to s\prime_1 = s_2$ and $s_2 \to s\prime_2 = 2s_2$ we must ensure that all blocks are of size $s\prime_1$. Thus, if we do a `push_back`-operation it will create a block of size $s\prime_2$ which is allowed since all our other blocks are of size $s\prime_1$. Consider the case when n has just reached a limit n_1, and all our data blocks are of size $s_1 = \sqrt{n_1}$. We then must have $\sqrt{n_1}$ data blocks. We must show that when n reaches $n_2 = 4n_1$ we have joined all our $\sqrt{n_1}$ data blocks into larger blocks. If we only create large blocks, i.e. only do `push_back`-operations, we must create as least $\frac{1}{2}\sqrt{n_1}$, since each creation will join two of the small blocks. The number of blocks b we create before $n = n_2$ is:

$$b = \frac{n_2 - n_1}{s_2} = \frac{3n_1}{2\sqrt{n_1}} = \frac{3\sqrt{n_1}^2}{2\sqrt{n_1}} = \frac{3}{2}\sqrt{n_1}$$

, which is greater than $\frac{1}{2}\sqrt{n_1}$, as requested. The case when we only create small blocks using `push_front` is similar:

$$b = \frac{n_2 - n_1}{s_1} = \frac{3n_1}{\sqrt{n_1}} = \frac{3\sqrt{n_1}^2}{\sqrt{n_1}} = 3\sqrt{n_1}$$

This is greater than $\sqrt{n_1}$, which is the number of merges we need to do to get all the existing blocks joined.

The argument for the case when we shrink the block sizes is similar to the case in which we grow them.

3.4.2 Deamortizing splits and joins

Joining or splitting data blocks take time $O(\sqrt{n})$. To ensure constant time pop and push operations we need to deamortize this. This can be achieved in the same way as the index block is deamortized. That is, whenever we do a push-operation we copy a number of elements from the two last small blocks to a block that is twice as big. When we need to insert a new data block, we join the two blocks simply by replacing them by the one large block. The case for pop-operations is similar. In addition, because we keep two copies of each element we must also make sure that both copies are updated if we do an element update operation. Due to its complexity I have not implemented this feature.

3.4.3 Relation to the index block

Joining two small blocks into a single large block causes holes to occur in the index block, since a pointer is freed. Splitting a large block into two smaller blocks creates an extra pointer, which must be inserted into the index block in the middle. This cannot be done in constant time. The solution to this problem is to use two index blocks, one for small blocks and one for large blocks, instead of just one. Joining two small blocks simply involves removing them from the back of the small index block and inserting one large block in the front of the large index block. Because of the `deque`-properties of the index blocks this can be done in constant time.

This means that the index block in the figure in section 3.1 is really implemented using two index blocks.

Because we use two index blocks, we must swap the index pointers when we update s_1 and s_2, since in case of a grow all large blocks are now small blocks, and in case of a shrink all small blocks are now large.

3.4.4 Preventing poor performance in border cases

A couple of things complicate the design of the data structure. Whenever a data block becomes empty we erase it and whenever a data block becomes full we create a new one. This can result in bad performance if the `deque` is used as a stack that continuously pops and pushes elements onto one end. If this happens on a border, each operation will have the extra overhead of creating or deleting a data block. To prevent this, Brodnik et al. suggests only marking a block as empty instead of deleting it. This block is then deleted when another block becomes empty.

Similarly when we have just merged two small blocks into one larger block because we have created a new data block we might need to split this larger block immediately after because the block just created is deleted again. In essence we undo the join just done, so by keeping the two small blocks instead of just deleting them we can revert to the these if the large block needs to be split.

Because of the added complexity I have not implemented these features, but they should be implemented to prevent that certain patterns of use will show poor performance.

3.5 Element access

A random element access provides a rank. From this rank we must find the relevant block, and the elements rank within that block.

Since all blocks are full except perhaps the two outermost and since we only have two block sizes, all of the same size being following each other, finding the right block and position simply becomes a matter of arithmetics. The locate function becomes somewhat complicated by the presence of two index blocks as can be confirmed by looking at its implementation. However it can still be performed in constant time.

3.6 Insert and erase-operations

By keeping all data blocks as circular sequences, we can implement the $O(\sqrt{n})$ insert and erase operations suggested by Goodrich and Kloss. If we do an erase we simply close up the hole made by the erase by moving elements from either side towards the hole. In case of an insert, we move elements away to either side to make room for the element we want to insert. Now, since all data blocks are circular we don't have to move the entire contents of all blocks, but only need to do a series of pop/push-operations. This means that we do $O(\sqrt{n})$ work to insert or erase the element from the relevant block, and $O(\sqrt{n})$ pop/push-operations.

If we decide that we should move elements right we do a series of `pop_back` - `push_front` operations. If we move elements left we do a series of `pop_front` - `push_back` operations. Since all blocks are full (except the outermost, from which we do not pop elements) the pop-operation will provide room for exactly the one element we want to push onto the data block. Thus, all blocks will still be full when we are done, so that invariant is kept.

Ideally, we decide whether to move right or left by considering to which side of the relevant block there are fewest blocks. We move towards that end in case of inserts and from that end in case of erases. This way we obtain the shortest sequence of pop/push-operations possible. Because of the added complexity of using two index blocks I have chosen not to implement this strategy. Section 4.2.3 explains how it is actually done.

4. Implementation

This section gives details about the implementation. The source code for the deque implementation is divided into three files, which are available as appendices. Appendix Appendix B.1 contains the code for the 2-level deque, here called `level2_deque`. Appendix Appendix B.2 contains the index block, `index_block`, and the circular sequence, here called `level1_tieredvector`, is contained in appendix Appendix B.3. Finally, appendix Appendix B.4 contains the source code for a simple non-standards-compliant deque, `simpledeque`. The source code still contains debug code, but this does not affect the benchmarks performed, since it is only included if a `__DEBUG__` define is present.

4.1 A simple deque data structure

Having implemented the circular sequence first, I quickly realized that I could implement a `deque` using only this data structure. Even though it only provides amortized constant time push and pop operations, I chose to implement it, just to have something to which I could compare the SGI STL `deque`. The `simpledeque::deque` class is basically a wrapper around a circular sequence block, and makes sure that the block grows whenever it becomes full (it does not shrink).

4.2 2-level deque

My general observation about the implementation of the 2 level deque, is that the code is rather complex. A lot of special cases need to be considered in almost every operation. The index block must consider the fullness of its block in almost every operation. Likewise, the level2_deque must at certain places consider whether elements are in one of the large blocks or in one of the small. The SGI implementation, which at first I found confusing and huge, is more easily understood compared to my implementation. The complexity of the code is reflected in the discovery of bugs rather late in the process.

However I have found the modular design of the code pleasing to work with, and this has enabled me to write quite specific unit-testing programs to root out bugs. First of all, the circular sequence is used for both the index block and the data blocks. Secondly, when I realized that I needed two index blocks, I simply added it, with no extra concern.

4.2.1 Memory allocation

The standard allows the user of a standard container to specify how the container should allocate memory by giving the container an allocator-template parameter. The implication of this is that all memory allocation by the container should be done through the allocator (this way the user can implement garbage collection, for instance).

My implementation does not adhere to this demand The `T`-elements of the container are correctly allocated by the allocator, but new data blocks and new index blocks are created simply by using `new` and `delete`. At first, I did not implement it because I did not know how to get a data block allocator from a `T`-allocator. When I discovered how to do it (it requires using the allocator's `typedef rebind<>::other`), I realized that I would have to write a copy constructor for the circular sequence class, something which I did not find the time to do.

4.2.2 Efficiency concerns

To achieve high efficiency in locating elements the implementation takes advantage of the fact that data blocks are always a power of two.

When we wish to find the block number from the element rank we must make division by the block size, which is either s_1 or s_2. In hardware, division is much more expensive than other arithmetic operations such as multiplication and shift. However if $s_1 = 2^x$, division by s_1, r/s_1 can be expressed as $r \gg x$ ($\gg$ meaning right shift). Division by s_2 is thus $r \gg x + 1$. For simplicity the exponent x, called `s1_exponent`, is stored in the class and updated whenever s_1 is updated.

The circular sequence uses a modulus-operation to calculate an element's position in memory. Modulus is just as expensive as division, which means that it should be avoided. Again using the power of two-property, it can be realized that when the blocks capacity is $c = 2^y$ calculating $r \bmod c$ is the same as calculating $r\&(c-1)$, where & means binary and.

4.2.3 Insert and erase

The optimal way to implement insert and erase is to do the push/pop sequence in the direction which has the smallest number of blocks. This is not how I have implemented it. Basically, I found it to complex to possibly have to switch index block in the middle of an operation that was already complex. The solution is to do push/pop-operations only in the small blocks, when we insert into or erase from a small block, and conversely only to push and pop from the large blocks if we do inserts into or erases from a large block. The time bound is still $O(\sqrt{n})$.

5. Benchmark results

Since this has primarily been an implementation project the results of running benchmarks are particularly interesting. I have tested the three core types of operations, that is grow and shrink-operations (section 5.1), element access operations (section 5.2), and insert and erase operations (section 5.3). Lastly, I make some remarks about space requirements (section 5.4).

All benchmarks have been performed on a 350 MHz Intel Pentium II machine running Redhat Linux 6.1. The compiler used was gcc-2.95.2. Four data structures are included in the benchmarks, namely `std::deque`, `std::vector`, `simpledeque::deque` and `level2_deque::deque`. Since the `vector` does not support front operations it has been left out of some benchmarks.

5.1 Grow and shrink

As the only standard container `deque` supports grow- and shrink-operations from both ends while providing random access. Therefore, when users choose the `deque` class they expect that *1)* the time bounds are worst case, meaning that no operation suddenly takes longer than the average, and *2)* that the grow and shrink operations are very fast. The useability of a deque implementation must be measured by these criteria. The first criteria seems to be fulfilled by the `level2_deque`, which can be seen from the straight curve that is plotted in appendices Appendix A.1, Appendix A.2 and Appendix A.3, eventhough sometimes the operations take $O(\sqrt{n})$-time. In comparison, the curves of `simpledeque` and `std::vector` are jagged, because their operations sometimes take linear time.

The second criteria, on the other hand, is not met. Looking at the graph in appendix Appendix A.1, the performance of level2_deque is approximately the same as that of simpledeque, which is rather disappointing. The result is even more disappointing when one considers that this is actually the best case situation for growing the data structure, since we never create blocks of small size, and thus only have to merge the small blocks created when block resizing occurs (cf. the arguments in section 3.4). That the data structure will perform even worse when growing it from the front can be seen from appendix Appendix A.2. Here we always create small blocks and the number of blocks we must join is thus maximal. Consequently, even `simpledeque` outperforms `level2_deque`.

The similar worst case for shrinking the data structure is when all shrinking is done from the front. When this approach is taken we never remove elements from the large blocks, and thus have the maximum possible blocks to be split. Appendix Appendix A.3clearly demonstrates this. The competition in this test is somewhat unfair, since `simpledeque` never shrinks its buffer and `std::deque` never shrinks its node map.

5.2 Element Access

For the deque data structure two types of element access are possible, random access and iterator access.

5.2.1 Random access

The random access test is performed simply by making n random access into a data structure of size n. The results are given in appendix Appendix A.4. The first thing to notice is that `simpledeque` is just as fast as `vector`, which means that the masking strategy for modulo calculation is very efficient. Secondly, the performance of `level2_deque` is significantly faster than that of `std::deque`. This validates the decision of using shift-operations instead of plain division for discovering in which block a given element is located.

5.2.2 Iterator access

The generic way to access STL containers is by using iterators. Hence, a benchmark of accessing the data structure through iterators is interesting. Appendix A.5 shows the performance of the data structures for a sequential access to all the elements, using the iterator's `operator*` and `operator++`. The results show that `vector` is the fastest, which is expected since the iterator type for `vector` is simply a `T*`. Likewise, `deque` also simply dereferences a pointer. The increment operation takes advantage of the internals of the data structure, so that the iterator operations are almost as fast as that of `vector`. `Simpledeque` uses the `level1_tieredvector`'s `operator`[], but since this function simply does an addition and a masking the performance comes close to that of `vector`.

The `level2_deque::deque` stands out compared to the other data structures by its poor performance. This is due to the very simple implementation of its iterator; the iterator is simply a rank. Thus, when the iterator is dereferenced, a call to `operator[]` is done to retrieve the element. This call has significant overhead compared to just dereferencing a raw memory pointer, or in the case of `simpledeque` to do a mask operation. An efficient iterator for the `level2_deque` can be implemented using the internals of the data structure, but my main motivation for implementing the iterator was to be able to write the `insert`- and `erase`-operations, which both take iterators as arguments. I expect the performance will be only a little slower than `std::deque`.

5.3 Insert and erase

The implementation of insert and erase-operations provides a theoretically improved time bound compared to existing implementations. This gain can clearly be seen from the insert and erase benchmarks. In fact, I found it difficult to devise a test in which the measured performance of `level2_deque` could be seen easily alongside the other data structures, since the gap to the other results was so large. In appendix Appendix A.6 and Appendix A.7 1000 elements are inserted or erased. Look for the dashed line at the bottom of the chart for the performance of `level2_deque`. The numbers range from 0.01 to 0.05 seconds. With 500000 elements `std::deque` is outperformed by a factor of approximately 280.

To demonstrate the performance of `level2_deque` more clearly I made another benchmark (Appendix A.8). In this benchmark 10000 elements are inserted, and the size of the data structure is as much as 10 million elements. None of the other data structures could finish his benchmark within a reasonably amount of time, and are thus not included in the chart. The curve shows that the insertion time of `level2_deque` actually grows (something that could not be discerned from the first two benchmarks), but not in a linear fashion. I repeated the test a number of times and the results were similar each time.This means that it is not system load or other external influence that make the running times fluctuate.

The results have to do with the state of the deque at the time of measurement. Since I chose to move elements towards the front of the deque whenever an element is to be inserted in one of the small blocks, and towards the end for insertion in large blocks the running time is determind by the number of blocks of the relevant size that is present at the time of insert. Had I chosen always to move in the direction that would give the fewest number of push/pop-pairs the curves would show a more linear growth.

5.4 Space measurement

I have been unable to come up with a good way to benchmark the space requirements. As a final solution I stopped execution of one of the benchmarks at the time when the data structure was at it largest, for all four data structures. Using the `Unix ps` command I was able to get some memory information. `ps` reported that all four data structures used approximately the same amount of memory. As expected `std::vector` took up the least memory, followed by `simpledeque`, `level2_deque` and `std::deque` but the differences were not big.

I will refrain from concluding anything based on this observation, but the space requirements should be explored further and in some other way than just using the `ps`-command.

6. Conclusion

A little about the result of the report. What have I done, what am I happy about.

Two directions for the future work on a deque class for the Copenhagen STL

6.1 Extending the 2-level deque

I have focused on implementing the core operations of the `deque` class. However, on the algorithmical level, some of the more unpleasant details such as deamortizing the splits and joins and keeping empty blocks available at the ends, have been left out and should be implemented (cf. section 3.4). Furthermore, work needs to be done to make the `deque` conform the the C++-standard. Naturally, this includes implementing all member and non-member-functions required, but two things in particular are non-trivial, namely iterators and exception safety.

First of all, the implemented iterator is too slow for practical use. Secondly, in a couple of special cases, the standard requires that iterators retain their validity after a grow or shrink operation. This is also the case for references to elements in the container. For this deque implementation, these guarantees are hard to achieve, since a grow or shrink possibly rearranges a large number of the elements. Another important case is that of exception safety. This requirement says that if an operation throws an operation (perhaps because some memory allocation failed), it must be as if the operation was not performed at all. For some operations, for instance when we insert elements in the middle, this can require the implementation of roll back semantics.

Finally, the implementation could quite possibly be tuned. This means that the functions should be profiled, in order to realize in what functions time is spend. However, I do not believe that even the most fine grained performance engineering will make the push and pop operations move anywhere near the performance of `std::deque`.

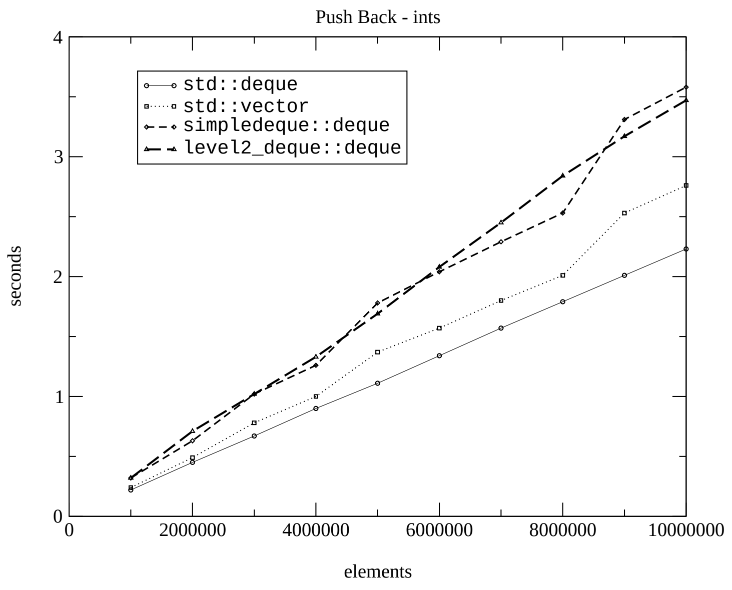
6.2 Future work

My suggestion for future work at implementing the `deque` class for the Copenhagen STL is thus to combine the ideas of the SGI deque and the results of this project. First of all, one should use fixed size data blocks. At the cost of space efficiency this will alleviate the need for joining and splitting blocks. Secondly, the use of an index block like the one used here, should be used, to prevent the relocation of the data block pointers.

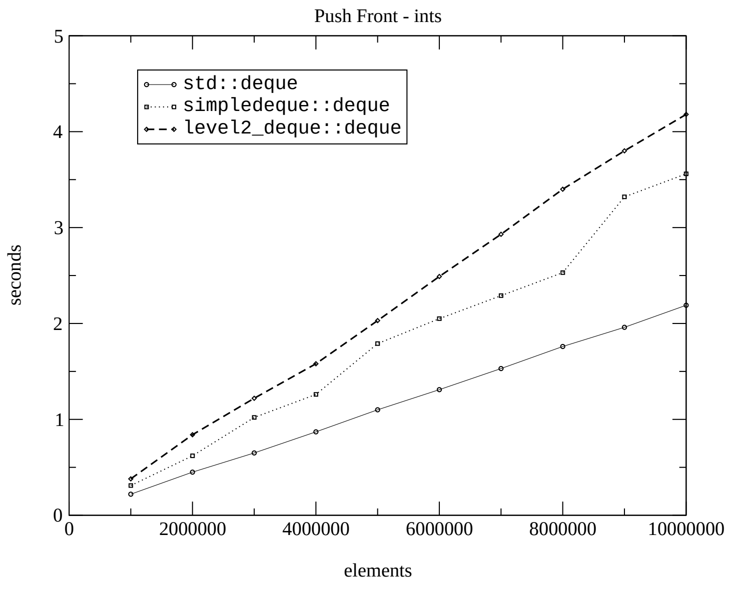
The benchmarks show that the deque implementation of this project offers performance gains in both random access and insert and erase operations. The random access performance is achieved simply by taking advantage of the fact that the data blocks are powers of two. There is no problem in extending this to a fixed block size solution to avoid the costly division-operations. The performance gain for inserts and erases can also be achieved for a fixed block size implementation by using circular sequences. The time bound will not be $O(\sqrt{n})$, but $O(\frac{n}{b})$. For a reasonably large block size, say 512, this is substantially better than the $O(n)$ time that `std::deque` offers.

Appendix A. Benchmark results

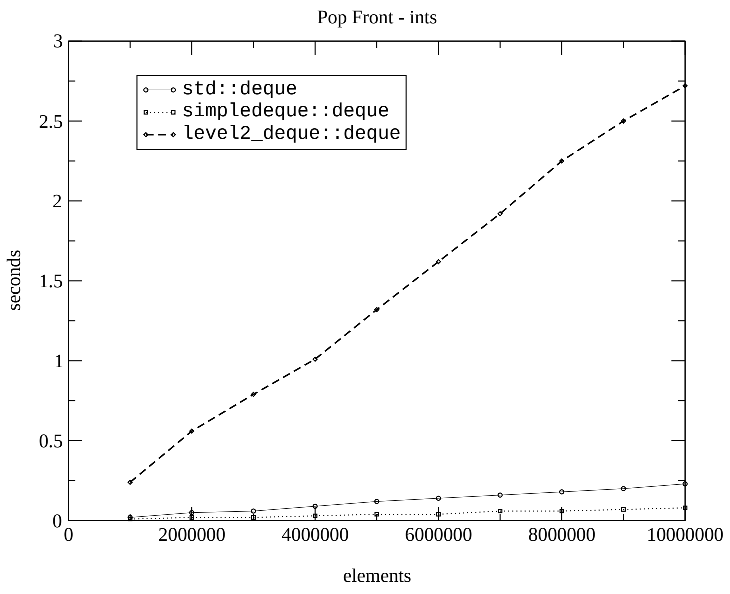
Appendix A.1 Push back



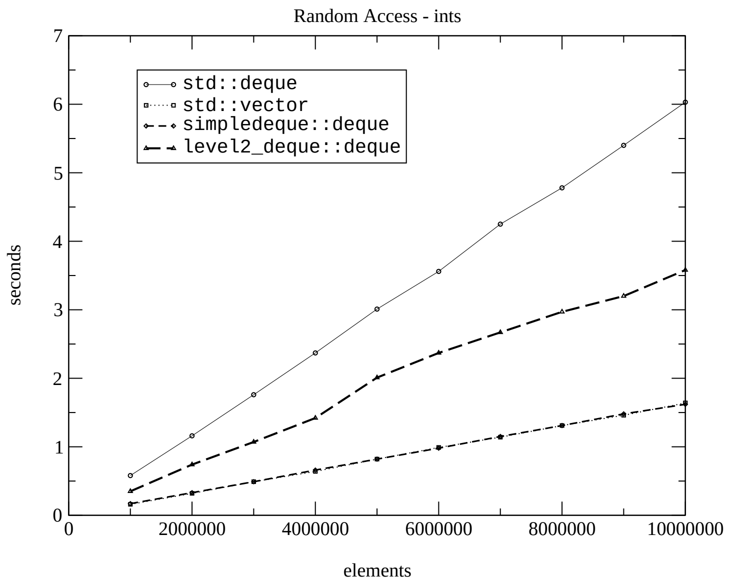
Appendix A.2 Push front



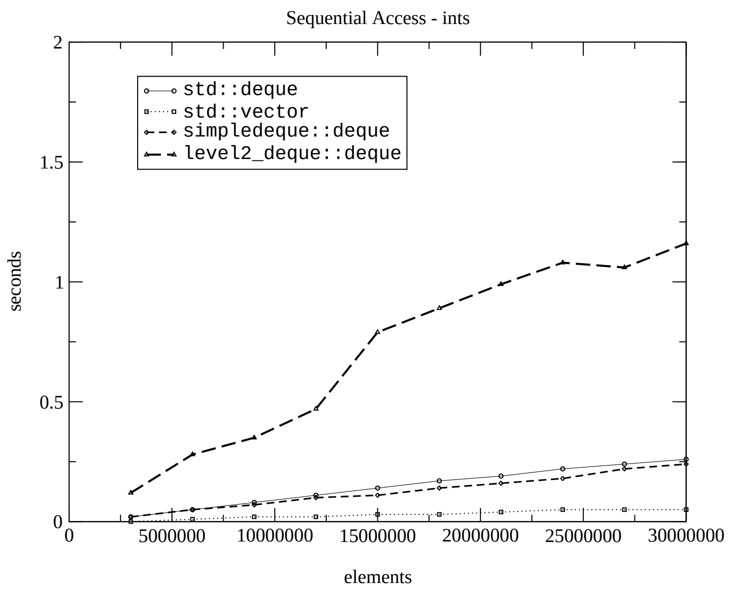
Appendix A.3 Pop front



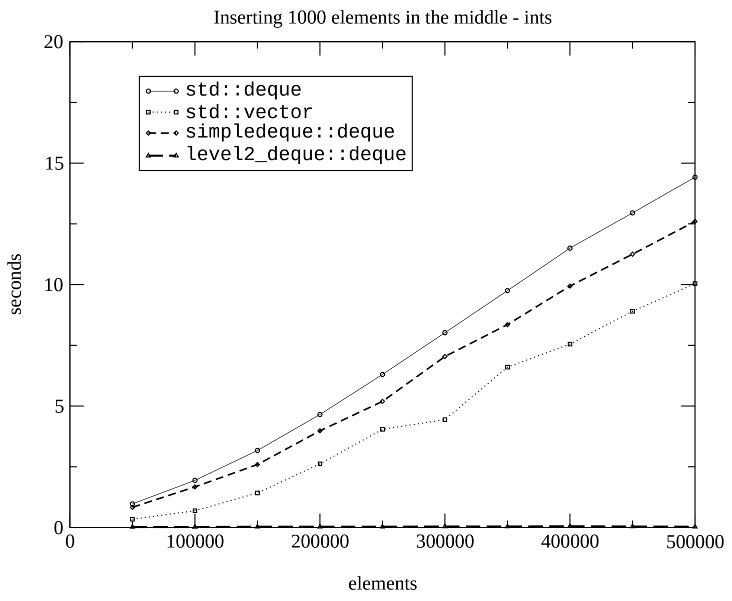
Appendix A.4 Random access



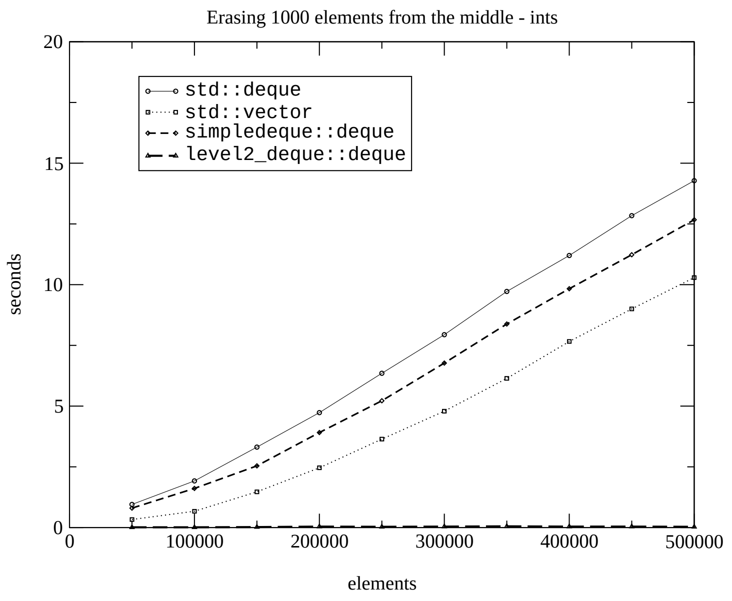
Appendix A.5 Sequential access



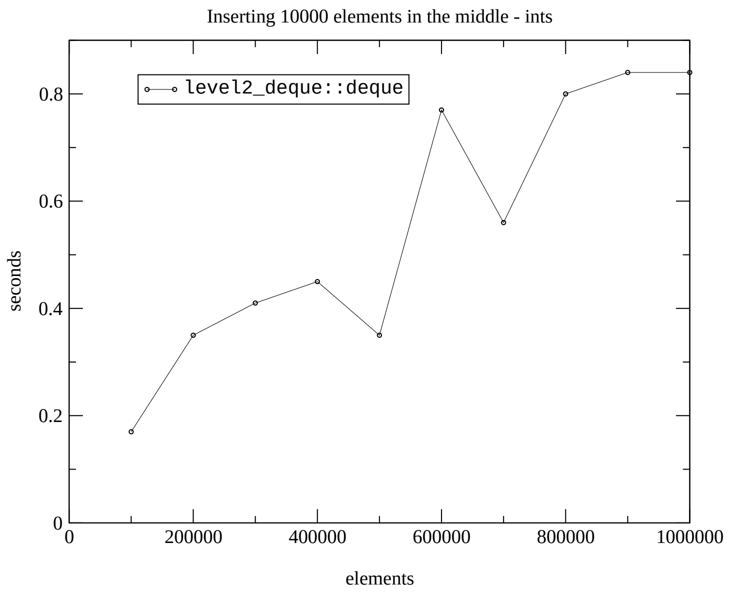
Appendix A.6 Insert



Appendix A.7 Erase



Appendix A.8 Insert 2



Appendix B. Source Code

Appendix B.1 level2_deque.cpp

```
/* -*- C++ -*- $Id: level2_deque.cpp,v 1.1 2002/03/07 14:56:47 jyrki Exp $ */
#ifndef __LEVEL2_DEQUE__
#define __LEVEL2_DEQUE__

#include <utility>
#include "index_block.cpp"

#if defined(__DEBUG__) && defined(L2D_LOCALMAIN)
#undef DEBUG_OUT(x)
#undef DEBUG_DO(x)
#define DEBUG_OUT(x) cout << x
#define DEBUG_DO(x) x
#else
#undef DEBUG_OUT(x)
#undef DEBUG_DO(x)
#define DEBUG_OUT(x) //void
#define DEBUG_DO(x) //void
#endif

namespace cphstl {
  namespace level2_deque {
    template <typename T, typename A, bool isconst> class deque_iterator; //forward declare

    template <class T, class A = std::allocator<T> >
    class deque {
      friend deque_iterator<T, A, true>;
      friend deque_iterator<T, A, false>;
      //types
    private:
      typedef level1_tieredvector<T, A> data_block_type;
      typedef data_block_type* data_block_pointer;
    public:
      typedef deque_iterator<T,A, false> iterator;
      typedef deque_iterator<T,A, true> const_iterator;
      typedef T value_type;
      typedef A allocator_type;
      typedef typename A::reference reference;
      typedef typename A::const_reference const_reference;
      typedef typename A::pointer pointer;
      typedef typename A::const_pointer const_pointer;
      typedef typename A::size_type size_type;
      typedef typename A::difference_type difference_type;
      typedef typename std::reverse_iterator<iterator> reverse_iterator;
      typedef typename std::reverse_iterator<const_iterator> const_reverse_iterator;
    private:
      //initial sizes
      static const size_type index_block_start_size = 8;
      static const size_type s1_start_size = 2; //512, 8
      static const size_type s1_exponent_start_size = 1; //9, 3

      typedef index_block<data_block_pointer,
        std::allocator<data_block_pointer>, index_block_start_size > index_block_type;
      typedef index_block_type* index_block_pointer;

      typedef std::pair<index_block_pointer,
        std::pair<size_type, size_type> > index_block_place_type; //rather ugly

      index_block_type index1; //initially small_index
      index_block_type index2; //initially large_index
      index_block_pointer small_index;
      index_block_pointer large_index;

      size_type s1; //size of small blocks
      size_type s2; //size of large blocks
      size_type shrink_limit; //when to halfe s1 and s2
      size_type grow_limit;   //when to double s1 and s2
      size_type s1_exponent; //x, where s1 = 2^x

      size_type total_size; //total number of elements

      index_block_place_type rank_to_index_block_place(size_type);
      reference locate(size_type); //find an element from rank
      void shrink_block_sizes();   //decrease s1 and s2 and related variables
      void grow_block_sizes();     //increase -- " --
      bool merge_blocks();         //merge last two small blocks
      bool split_block();          //split first large block
    public:
      //Construct/destroy/copy
      deque();
      ~deque();
      //Capacity
      size_type size() { return total_size; }
      // Iterators
      iterator begin () { return iterator(this, 0); }
      const_iterator begin () const { return iterator(this, 0); }
      iterator end () { return iterator(this, total_size); }
      const_iterator end () const { return iterator(this, total_size); };
      //Element Access
      reference operator[] (size_type rank) { return locate(rank); }
      reference front () { return locate(0); }
      reference back () { return locate(total_size - 1); }
      //Modifiers
      void pop_front ();
      void pop_back ();
      void push_front (const T& elem);
      void push_back (const T& elem);
      iterator insert (iterator it, const T& elem);
      iterator erase (iterator it);
      //debug
#if defined(__DEBUG__)
      void PRINT() {
        for (size_type i = 0; i < small_index->size(); ++i) {
          for (size_type j = 0; j < (*small_index)[i]->size(); ++j) {
            cout << (*(*small_index)[i])[j] << " ";
          }
        }
        for (size_type i = 0; i < large_index->size(); ++i) {
          for (size_type j = 0; j < (*large_index)[i]->size(); ++j) {
            cout << (*(*large_index)[i])[j] << " ";
          }
        }
        cout << endl;
      }

      void PRINTBLOCKS() {
        for (size_type i = 0; i < small_index->size(); ++i) {
          cout << "b" << i << ": ";
          for (size_type j = 0; j < (*small_index)[i]->size(); ++j) {
            cout << (*(*small_index)[i])[j] << " ";
          }
          cout << endl;
        }
        for (size_type i = 0; i < large_index->size(); ++i) {
          cout << "b" << i+small_index->size() << ": ";
          for (size_type j = 0; j < (*large_index)[i]->size(); ++j) {
            cout << (*(*large_index)[i])[j] << " ";
          }
          cout << endl;
        }
      }

      void PRINTSIZES() {
        for (size_type i = 0; i < small_index->size(); ++i) {
          cout << (*small_index)[i]->size() << "/"
               << (*small_index)[i]->capacity() << " ";
        }
        for (size_type i = 0; i < large_index->size(); ++i) {
          cout << (*large_index)[i]->size() << "/"
               << (*large_index)[i]->capacity() << " ";
        }
        cout << endl;
      }
#endif
    };

    //---- helper functions
    template <class T, class A>
    inline bool deque<T, A>::merge_blocks()
    {
      DEBUG_OUT("----- merge_blocks -----" << endl);
      bool result = false; //returns true if block sizes was updated
      //above limit
      if ((total_size >= grow_limit) && (small_index->size() == 1)) {
        data_block_pointer block = small_index->back();
        small_index->back()->grow_to_double(); //O(sqrt(n))
        large_index->new_block_front(block);
        small_index->delete_block_back();
        grow_block_sizes();
        result = true;
      } else if (small_index->size() > 1) {
        //relocate one block and copy elements from the other block to this
        data_block_pointer last_b = small_index->back();
        small_index->delete_block_back();
        data_block_pointer next_to_last_b = small_index->back();
        small_index->delete_block_back();

        next_to_last_b->grow_to_double(); //O(sqrt(n))
        for (size_type i = 0; i < s1; ++i) { //O(sqrt(n))
          next_to_last_b->push_back((*last_b)[i]);
        }
        delete last_b;
        large_index->new_block_front(next_to_last_b);
        if (total_size >= grow_limit) {
          grow_block_sizes();
          result = true;
        }
      } else { //last case: no small blocks exist: just grow if necessary
        if (total_size >= grow_limit) {
          grow_block_sizes();
          result = true;
        }
      }
      return result;
    }

    template <class T, class A>
    inline bool deque<T, A>::split_block()
    {
      DEBUG_OUT("----- split_block -----" << endl);
      bool result = false; //returns true if block sizes was updated
      if (large_index->size() > 0) {
        data_block_pointer first_lb = large_index->first_block();

        data_block_pointer sb_1 = new data_block_type(s1);
        size_type run_to = (first_lb->size() > s1) ? s1 : first_lb->size();
        for (size_type i = 0; i < run_to; ++i) {
          //O(sqrt(n))
          sb_1->push_back((*first_lb)[i]);
        }
        small_index->new_block_back(sb_1);

        if (first_lb->size() > s1) {
          data_block_pointer sb_2 = new data_block_type(s1);
          run_to = first_lb->size();
          for (size_type i = s1; i < run_to; ++i) {
            //O(sqrt(n))
            sb_2->push_back((*first_lb)[i]);
          }
          small_index->new_block_back(sb_2);
        }
        delete first_lb;
        large_index->delete_block_front();
      }
      if ((total_size <= shrink_limit) && (shrink_limit > 0)) {
        shrink_block_sizes();
        result = true;
      }
      return result;
    }
```

```
template <class T, class A>
inline void deque<T, A>::shrink_block_sizes()
{
  DEBUG_OUT("-- shrink_block_sizes()" << endl);
  //swap index pointers
  assert(large_index->size() == 0);
  index_block_pointer temp = small_index;
  small_index = large_index;
  large_index = temp;
  //setup new sizes and limits
  s1 /= 2;
  s2 /= 2;
  grow_limit /= 4;
  shrink_limit = (s1 == s1_start_size)? 0 : shrink_limit /= 4;
  --s1_exponent;

  DEBUG_OUT("s1: " << s1 << ", s2: " << s2
            << ", grow_l: " << grow_limit << ", shrink_l: "
            << shrink_limit << ", s1_exponent: " << s1_exponent << endl);
}

template <class T, class A>
inline void deque<T, A>::grow_block_sizes()
{
  DEBUG_OUT("-- grow_block_sizes()" << endl);
  //swap index pointers
  assert(small_index->size() == 0);
  index_block_pointer temp = small_index;
  small_index = large_index;
  large_index = temp;
  //setup new sizes and limits
  s1 *= 2;
  s2 *= 2;
  grow_limit *= 4;
  shrink_limit = grow_limit/4; //to handle initial 0 shrink_limit
  ++s1_exponent;

  DEBUG_OUT("s1: " << s1 << ", s2: " << s2
            << ", grow_l: " << grow_limit << ", shrink_l: "
            << shrink_limit << ", s1_exponent: " << s1_exponent << endl);
}

template <class T, class A>
inline deque<T, A>::index_block_place_type deque<T, A>::rank_to_index_block_place(size_type rank)
{
  //result = (index-pointer, (blockno, index in block))
  index_block_place_type result;
  size_type available_first_block;
  size_type elements_in_small_blocks;

  if (small_index->size() > 0) {
    available_first_block = s1 - small_index->first_block()->size();
    elements_in_small_blocks = s1 * small_index->size() - available_first_block;
  } else {
    available_first_block = s2 - large_index->first_block()->size();
    elements_in_small_blocks = 0;
  }
  if (rank < elements_in_small_blocks) {
    //element is in small blocks
    result.first = small_index;
    if (rank < small_index->first_block()->size()) {
      //element in very first small block
      result.second.first = 0;
      result.second.second = rank;
    } else {
      //element in another small block
      result.second.first = 1 +
        ((rank - small_index->first_block()->size()) >> s1_exponent);
      result.second.second = rank - (result.second.first*s1 - available_first_block);
    }
  } else {
    //element is in a large block
    result.first = large_index;
    size_type large_rank = rank - elements_in_small_blocks;
    size_type available_first_large_block = (s2 - large_index->first_block()->size());
    if (large_rank < large_index->first_block()->size()) {
      //element in first large block
      result.second.first = 0;
      result.second.second = large_rank;
    } else {
      //element in another large block
      result.second.first = 1 +
        ((large_rank - large_index->first_block()->size()) >> (s1_exponent+1));
      result.second.second = large_rank - (result.second.first*s2 - available_first_large_block);
    }
  }
  return result;
}

template <class T, class A>
inline deque<T, A>::reference deque<T, A>::locate(size_type rank)
{
  index_block_place_type p = rank_to_index_block_place(rank);
  return (*(*p.first)[p.second.first])[p.second.second];
}

//---- construct/destroy/copy
template <class T, class A>
inline deque<T, A>::deque()
  : small_index(&index1), large_index(&index2),
  s1(s1_start_size), s2(s1*2),
  shrink_limit(0), grow_limit(s1*s1),
  s1_exponent(s1_exponent_start_size),
  total_size(0)
{
  DEBUG_OUT("s1: " << s1 << ", s2: " << s2
            << ", grow_l: " << grow_limit << ", shrink_l: "
            << shrink_limit << ", s1_exponent: " << s1_exponent << endl);
}

template <class T, class A>
inline deque<T, A>::~deque()
```

```
{
  //destroy all data blocks
  while (!(small_index->size() == 0)) {
    delete small_index->back();
    small_index->delete_block_back();
  }
  while (!(large_index->size() == 0)) {
    delete large_index->back();
    large_index->delete_block_back();
  }
}

//---- modifiers
template <class T, class A>
inline void deque<T, A>::pop_front ()
{
  DEBUG_OUT("------- pop_front() -------" << endl);
  data_block_pointer first_block;
  if (small_index->size() == 0) {
    //no small blocks
    first_block = large_index->first_block();
    first_block->pop_front();
    if (first_block->is_empty()) {
      delete first_block;
      large_index->delete_block_front();
      split_block();
    }
  } else {
    first_block = small_index->first_block();
    first_block->pop_front();
    if (first_block->is_empty()) {
      delete first_block;
      small_index->delete_block_front();
      split_block();
    }
  }
  --total_size;
}

template <class T, class A>
inline void deque<T, A>::pop_back ()
{
  DEBUG_OUT("------- pop_back() -------" << endl);
  data_block_pointer last_block;
  if (large_index->size() == 0) {
    //no large blocks
    last_block = small_index->last_block();
    last_block->pop_back();
    if (last_block->is_empty()) {
      delete last_block;
      small_index->delete_block_back();
      split_block();
    }
  } else {
    last_block = large_index->last_block();
    last_block->pop_back();
    if (last_block->is_empty()) {
      delete last_block;
      large_index->delete_block_back();
      split_block();
    }
  }
  --total_size;
}

template <class T, class A>
inline void deque<T, A>::push_front (const T& elem)
{
  DEBUG_OUT("------- push_front() -------" << endl);
  data_block_pointer first_block;
  if (small_index->size() == 0) {
    //first block should be created
    first_block = 0;
  } else {
    //first block is first small block
    first_block = small_index->first_block();
  }

  if ((first_block == 0) || first_block->is_full()) {
    merge_blocks();
    first_block = new data_block_type(s1);
    small_index->new_block_front(first_block);
  }
  first_block->push_front(elem);
  ++total_size;
}

template <class T, class A>
inline void deque<T, A>::push_back (const T& elem)
{
  DEBUG_OUT("------- push_back() -------" << endl);
  data_block_pointer last_block;
  if (large_index->size() == 0) {
    //last block should be created
    last_block = 0;
  } else {
    //last block is last large block
    last_block = large_index->last_block();
  }

  if ((last_block == 0) || last_block->is_full()) {
    merge_blocks();
    last_block = new data_block_type(s2);
    large_index->new_block_back(last_block);
  }
  last_block->push_back(elem);
  ++total_size;
}

template <class T, class A>
inline deque<T, A>::iterator deque<T, A>::insert(iterator it, const T& elem)
{
  DEBUG_OUT("------- insert() -------" << endl);
  size_type rank = it.rank;
  index_block_place_type p = rank_to_index_block_place(rank);
  DEBUG_OUT(p.first << ": " << p.second.first << ", "
            << p.second.second << endl);
  //move the elements
  if (p.first == small_index) {
    //move left
    T elem1;
    if (p.second.second == 0) {
      elem1 = elem;
    } else {
      elem1 = ((*small_index)[p.second.first])->front();
      ((*small_index)[p.second.first])->pop_front();
      ((*small_index)[p.second.first])->insert(p.second.second-1, elem);//pop -> -1
    }
    if (p.second.first > 0) {
      size_type i = p.second.first - 1;
      while (i >= 0) {
        T elem2 = (*small_index)[i]->front();
        (*small_index)[i]->pop_front();
        (*small_index)[i]->push_back(elem1);
        elem1 = elem2;
        if (i == 0) {
          break;
        } else {
          --i;
        }
      }
    }
    push_front(elem1);
  } else {
    //move right
    size_type i = p.second.first;
    T elem1 = elem;
    while (i < large_index->size()) {
      T elem2 = (*large_index)[i]->back();
      (*large_index)[i]->pop_back();
      if (i == p.second.first) {
        if (p.second.second < (*large_index)[i]->size()) {
          (*large_index)[i]->insert(p.second.second, elem1);
        } else {
          (*large_index)[i]->push_back(elem1);
        }
      } else {
        (*large_index)[i]->push_front(elem1);
      }
      elem1 = elem2;
      ++i;
    }
    push_back(elem1); //is it really that simple? YES!
    //if we pop from a non-full block then just put it back.
    //If it is full then calling push makes sure structure grows accordingly
  }
  return it; //because of our simple iterator
}
```

```
//-- erase
template <class T, class A>
inline deque<T, A>::iterator deque<T, A>::erase(iterator it)
{
  DEBUG_OUT("------- erase() -------" << endl);
  size_type rank = it.rank;
  index_block_place_type p = rank_to_index_block_place(rank);
  DEBUG_OUT(p.first << ": " << p.second.first << ", "
            << p.second.second << endl);
  //move the elements
  if (p.first == small_index) {
    //we are in the small index: move right
    size_type i = p.second.first;
    T elem;
    while (i >= 0) {
      if (i == p.second.first) {
        ((*small_index)[i])->erase(p.second.second);
      } else {
        elem = ((*small_index)[i])->back();
        ((*small_index)[i+1])->push_front(elem);
        ((*small_index)[i])->pop_back();
      }
      if (i == 0) {
        break; //to prevent infinite loop when size_type is unsigned
      } else {
        --i;
      }
    }
    if (small_index->first_block()->is_empty()) {
      delete (small_index->first_block());
      small_index->delete_block_front();
      split_block();
    }
  } else {
    //we are in the large index: move left
    size_type i = p.second.first;
    T elem;
    while (i < large_index->size()) {
      if (i == p.second.first) {
        ((*large_index)[i])->erase(p.second.second);
      } else {
        elem = ((*large_index)[i])->front();
        ((*large_index)[i-1])->push_back(elem);
        ((*large_index)[i])->pop_front();
      }
      ++i;
    }
    if (large_index->last_block()->is_empty()) {
      delete (large_index->last_block());
      large_index->delete_block_back();
      split_block();
    }
  }
  --total_size;
  //return iterator pointing to next element
  //which is still it. If we erase end-element
  //it will be the end-iterator. I suppose this is ok
  return it;
}

//---- iterator class
template <bool flag, class IsTrue, class IsFalse>
struct choose;

template <class IsTrue, class IsFalse>
struct choose<true, IsTrue, IsFalse> {
  typedef IsTrue type;
};

template <class IsTrue, class IsFalse>
struct choose<false, IsTrue, IsFalse> {
  typedef IsFalse type;
};

template <class T, class A, bool isconst>
class deque_iterator {
  friend deque<T, A>;
private:
  deque<T, A>* d;
  deque<T, A>::size_type rank;
public:
  //traits
  typedef random_access_iterator_tag iterator_category;
  typedef deque<T, A>::value_type value_type;
  typedef deque<T, A>::difference_type difference_type;
  typedef typename choose<isconst, deque<T, A>::const_pointer,
    deque<T, A>::pointer>::type pointer;
  typedef typename choose<isconst, deque<T, A>::const_reference,
    deque<T, A>::reference>::type reference;
  //construct/destruct
  deque_iterator() : d(0), rank(0) {}
  deque_iterator(deque<T, A>* dq, deque<T, A>::size_type nrank) : d(dq), rank(nrank){ }
  deque_iterator(const deque_iterator& x) : d(x.d), rank(x.rank) {}
  ~deque_iterator() {}

  bool operator==(const deque_iterator& x) const {
    return  ((d == x.d) && (rank == x.rank));
  }

  bool operator!=(const deque_iterator& x) const {
    return ((rank != x.rank) || (d != x.d));
  }

  reference operator*() const {
    return (*d)[rank];
  }

  pointer operator->() const {
    return &(operator*());
  }
```

```
      deque_iterator& operator++() {
        if (!(rank == d->total_size))
          rank++;
        return *this;
      }

      deque_iterator operator++(int) {
        deque_iterator tmp = *this;
        ++*this;
        return tmp;
      }

      deque_iterator& operator--() {
        rank--; //also works for the end-iterator, since it never grows beyond total_size
        return *this;
      }

      deque_iterator operator--(int) {
        deque_iterator tmp = *this;
        --*this;
        return tmp;
      }
    };

  } //namespave level2_deque
} //namespace cphstl

#endif
```

Appendix B.2 index_block.cpp

```
/* -*- C++ -*- $Id: index_block.cpp,v 1.1 2002/03/07 14:56:47 jyrki Exp $ */

#ifndef __INDEX_BLOCK__
#define __INDEX_BLOCK__

#include "tieredvector.cpp"

#if defined(__DEBUG__) && defined(IB_LOCALMAIN)
#define DEBUG_OUT(x) cout << x
#define DEBUG_DO(x) x
#else
#define DEBUG_OUT(x) //void
#define DEBUG_DO(x) //void
#endif

#ifdef INDEX_BLOCK_TEST
template <typename T> class index_block_test; //forward declare
#endif

namespace cphstl {
  template <typename T, typename A, unsigned int min_size = 8>
  class index_block {
#if defined(INDEX_BLOCK_TEST)
    friend index_block_test<T>;
#endif
```

```
private:
  //types
  typedef unsigned int size_type;
  typedef T& reference;
  typedef T* pointer;
  typedef level1_tieredvector<T, A> index_block_type;
  typedef index_block_type* index_block_pointer;

  //index block allocator
  //use of this is deferred
  //typedef typename A::rebind<index_block_type>::other index_block_alloc_type;
  //    index_block_alloc_type block_alloc;
  //copy range
  size_type copied_from;//smallest ranking elem copied to small/large_block
  size_type copied_to;  //largest ranking elem copied to small/large_block
  //ranges
  size_type _2_8_full; //when to shrink
  size_type _3_8_full; //when to start deamortizing shrinks
  size_type _5_8_full; //when to start deamortizing grows
  //index blocks
  index_block_pointer small_block;
  index_block_pointer block;
  index_block_pointer large_block;
  //helper functions
  void setup_ranges();
  void deamortize_grow();
  void deamortize_shrink(bool); //true=delete_front, false=delete_back
  void grow();   //grows index block
  void shrink(); //shrinks, but only if current capacity > min_size
public:
  //construct/destruct/copy
  index_block();
  ~index_block();
  //block operations
  void new_block_front(const T&);
  void new_block_back(const T&);
  void delete_block_front();
  void delete_block_back();
  //access
  reference front () { return block->front(); }
  reference back () { return block->back(); }

  //info (used by level2_deque)
  size_type size() { return block->size(); } //current number of data blocks
  size_type capacity() { return block->capacity(); } //max number of data blocks
  //if these are called when size() == 0 then argh!
  reference last_block() { return (*block)[block->size()-1]; }
  reference first_block() { return (*block)[0]; }
  reference operator[](size_type i) { return (*block)[i]; }
};

//----------- helper functions
template <typename T, typename A, unsigned int min_size>
inline void index_block<T, A, min_size>::setup_ranges()
{
  DEBUG_OUT("----------- setup_ranges -----------\n");
  size_type _1_8_full = capacity()/8;
  _2_8_full = 2 * _1_8_full;
  _3_8_full = 3 * _1_8_full;
  _5_8_full = 5 * _1_8_full;
  copied_from = copied_to = -1;
}

template <typename T, typename A, unsigned int min_size>
inline void index_block<T, A, min_size>::deamortize_grow() {
  DEBUG_OUT("---- deamortize_grow ----\n");
  if (block->size() == _5_8_full + 1) {
    //create large_block and copy middle 3 elements
    large_block = new level1_tieredvector<T, A>(capacity()*2);
    size_type middle = block->size()/2;
    large_block->push_back((*block)[middle-1]);
    large_block->push_back((*block)[middle]);
    large_block->push_back((*block)[middle+1]);
    copied_from = middle-1; copied_to = middle+1;
  } else if (block->size() > _5_8_full + 1) {
    //always copy most middle elements
    size_type copied = 0;
    while ((copied < 3) &&
           ((copied_from != 0) || (copied_to != block->size() - 1))) {
      if (copied_from != 0) {
        copied_from--;
        large_block->push_front((*block)[copied_from]);
        copied++;
      }
      if ((copied < 3) && (copied_to != block->size() - 1)) {
        copied_to++;
        large_block->push_back((*block)[copied_to]);
        copied++;
      }
    }
  }
  DEBUG_OUT("copied_from " << copied_from
            << ", copied_to: " << copied_to << endl);
  DEBUG_OUT(endl << "block (" << block << ", "
            << block->capacity() << "):" << endl);
  DEBUG_DO(block->PRINT());
  DEBUG_OUT("large_block (" << large_block << ", " << large_block->capacity() << "):" << endl);
  DEBUG_DO(large_block->PRINT());
}

template <typename T, typename A, unsigned int min_size>
inline void index_block<T, A, min_size>::deamortize_shrink(bool is_delete_front) {
  if (capacity() == min_size) return; //we don't shrink if we are already smallest size
  DEBUG_OUT("---- deamortize_shrink ----" << endl);
  if (block->size() == _3_8_full - 1) {
    //create and copy
    small_block = new level1_tieredvector<T, A>(capacity()/2);
    size_type middle = block->size()/2;
    small_block->push_back((*block)[middle]);

    if (is_delete_front) {
      small_block->push_back((*block)[middle+1]);
      copied_from = middle; copied_to = middle+1;
    } else {
      small_block->push_front((*block)[middle-1]);
      copied_from = middle-1; copied_to = middle;
    }
  } else if (block->size() < _3_8_full - 1) {
    //copy acording to is_delete_front parameter
    size_type copied = 0;
    if (is_delete_front) {
      //first back, then front
      while ((copied < 2) && (copied_to != block->size() - 1)) {
        copied_to++;
        small_block->push_back((*block)[copied_to]);
        copied++;
      }
      while ((copied < 2) && (copied_from != 0)) {
        copied_from--;
        small_block->push_front((*block)[copied_from]);
        copied++;
      }
    } else {
      //first front, then back
      while ((copied < 2) && (copied_from != 0)) {
        copied_from--;
        small_block->push_front((*block)[copied_from]);
        copied++;
      }
      while ((copied < 2) && (copied_to != block->size() - 1)) {
        copied_to++;
        small_block->push_back((*block)[copied_to]);
        copied++;
      }
    }
  }

  DEBUG_OUT("copied_from " << copied_from
            << ", copied_to: " << copied_to << endl);
  DEBUG_OUT(endl << "block (" << block << ", " << block->capacity()
            << "):" << endl);
  DEBUG_DO(block->PRINT());
  DEBUG_OUT("small_block (" << small_block << ", "
            << small_block->capacity() << "):" << endl);
  DEBUG_DO(small_block->PRINT());
}

//----------- construct/destroy/copy
template <typename T, typename A, unsigned int min_size>
inline index_block<T, A, min_size>::index_block()
  : small_block(0), large_block(0)
{
  //block
  block = new level1_tieredvector<T, A>(min_size);
  DEBUG_OUT("block->capacity(): " << block->capacity() << endl);
  //initial setup
  setup_ranges();
}
```

```
template <typename T, typename A, unsigned int min_size>
inline index_block<T, A, min_size>::~index_block()
{
  delete block;
  if (small_block != 0)
    delete small_block;
  if (large_block != 0)
    delete large_block;
}

//----------- block operations
//new_block_front/back: deamortizes switch to large_block
template <typename T, typename A, unsigned int min_size>
inline void index_block<T, A, min_size>::new_block_front(const T& elem)
{
  DEBUG_OUT("----------- new_block_front ----------" << endl);
  if (block->is_full()) {
    grow();
    block->push_front(elem);
  } else {
    block->push_front(elem);
    if (block->size() > _5_8_full) {
      copied_from++;
      copied_to++;
      deamortize_grow();
    } else if (block->size() == _3_8_full) {
      delete small_block;
      small_block = 0;
      copied_from = copied_to = -1;
    } else if (block->size() < _3_8_full) {
      copied_from++;
      copied_to++;
    }
  }
}

template <typename T, typename A, unsigned int min_size>
inline void index_block<T, A, min_size>::new_block_back(const T& elem)
{
  DEBUG_OUT("----------- new_block_back ----------" << endl);
  if (block->is_full()) {
    grow();
    block->push_back(elem);
  } else {
    block->push_back(elem); //!
    if (block->size() > _5_8_full) {
      deamortize_grow();
    } else if (block->size() == _3_8_full) {
      delete small_block;
      small_block = 0;
      copied_from = copied_to = -1;
    } else if (block->size() < _3_8_full) {
      //do nothing
    }
  }
```

```
}

//delete_block_front/back: deamortizes switch to small_block
template <typename T, typename A, unsigned int min_size>
inline void index_block<T, A, min_size>::delete_block_front()
{
  DEBUG_OUT("----------- delete_block_front ----------" << endl);
  block->pop_front();
  if ((block->size() < _3_8_full) && (capacity() != min_size)) {
    if (copied_from == 0) {
      small_block->pop_front();
      copied_to--;
    } else {
      copied_from--;
      copied_to--;
    }
    deamortize_shrink(true); //true indicates delete_block_front
    if (block->size() == _2_8_full) {
      shrink();
    }
  }
  else if (block->size() == _5_8_full) {
    delete large_block;
    large_block = 0;
    copied_from = copied_to = -1;
  }
  else if (block->size() > _5_8_full) {
    if (copied_from == 0) {
      large_block->pop_front();
      copied_to--;
    } else {
      copied_from--;
      copied_to--;
    }
  }
}


template <typename T, typename A, unsigned int min_size>
inline void index_block<T, A, min_size>::delete_block_back()
{
  DEBUG_OUT("----------- delete_block_back ----------" << endl);
  block->pop_back();
  if ((block->size() < _3_8_full) && (capacity() != min_size)) {
    if (copied_to == block->size()) {
      copied_to--;
      small_block->pop_back();
    }
    deamortize_shrink(false); //false indicates delete_block_back
    if (block->size() == _2_8_full) {
      shrink();
    }
  } else if (block->size() == _5_8_full) {
    delete large_block;
    large_block = 0;
    copied_from = copied_to = -1;
    } else if (block->size() > _5_8_full) {
      if (copied_to == block->size()) {
        copied_to--;
        large_block->pop_back();
      }
    }
  }

  //----------- grow/shrink
  template <typename T, typename A, unsigned int min_size>
  inline void index_block<T, A, min_size>::grow()
  {
    DEBUG_OUT("----------- grow ----------" << endl);
    delete block;
    block = large_block;
    large_block = 0;
    setup_ranges();
  }

  template <typename T, typename A, unsigned int min_size>
  inline void index_block<T, A, min_size>::shrink()
  {
    if (capacity() != min_size) { //only shrink if we are bigger than min_size
      DEBUG_OUT("----------- shrink ---------" << endl);
      delete block;
      block = small_block;
      small_block = 0;
      setup_ranges();
    }
  }

} //end namespace cphstl

#endif
```

Appendix B.3 tieredvector.cpp

```
/* -*- C++ -*- $Id: tieredvector.cpp,v 1.1 2002/03/07 14:56:47 jyrki Exp $ */
/*! \ingroup deque
 * \file tieredvector.cpp
 * \brief This file provides the basic tiered vector component, the class level1_tieredvector.
 */
#ifndef __LEVEL1_TIEREDVECTOR__
#define __LEVEL1_TIEREDVECTOR__

#include <memory>
//temporary includes
#include <cassert>
#include <iostream>
using std::cout;
using std::endl;

namespace cphstl {
  //forward declarations
  namespace simpledeque {
    template <typename T, typename A> class deque;
    template <typename T, typename A, bool isconst> class deque_iterator;
  };

  /*! \ingroup deque
   *  \class level1_tieredvector
   *  \brief \c level1_tieredvector<T, \c Allocator> manages a contigous memory block of elements of t
   *
   * It consists of \c head and \c tail pointers that point to different places in the same contigous m
   * <dl>
   * <dt><code>head > tail</code></dt>
   * <dd>\c buf: <img align="top" src="lv1tv_1.gif"></dd>
   * <dt><code>tail > head </code></dt>
   * <dd>\c buf: <img align="top" src="lv1tv_2.gif"></dd>
   * </dl>
   * \par Implementation details
   * <dl>
   * <dt> Modulus-calculation </dt>
   * <dd> A modulus-operation is used for mapping between ranks and actual positions in memory. This i
   * </dl>
   *
   * \remark
   * Relies on \c std::allocator<T>
   * \remark
   * At the moment \c assert 's are used at various places. These should be removed when code reaches a
   * \remark
   * Debug-functions are presently available: \c RAW(i) gives value of element at position <code>buf +
   */
  template < typename T, typename A >
  class level1_tieredvector {
    friend simpledeque::deque<T, A>;
    friend simpledeque::deque_iterator<T, A, true>;
    friend simpledeque::deque_iterator<T, A, false>;
  public:
    //typedefs
    typedef T value_type;
    typedef A allocator_type;
    typedef typename A::size_type size_type;
    typedef typename A::difference_type  difference_type;
    typedef typename A::pointer pointer;
    typedef typename A::const_pointer const_pointer;
    typedef typename A::reference reference;
    typedef typename A::const_reference const_reference;
    //construct/destruct/copy
    level1_tieredvector(size_type = 128);
    ~level1_tieredvector();
    //capacity
    size_type size() const { return actsize; }
    size_type capacity() const { return bufsize; }
    bool is_full() const { return actsize == bufsize; }
    bool is_empty() const { return actsize == 0; }
    //element access
    reference operator[](size_type);
    reference front ();
    reference back ();
    //modifiers
    void insert(size_type, const T&);
    void erase(size_type);
    void push_front(const T&);
    void push_back(const T&);
    void pop_front();
    void pop_back();

    //--DEBUG -- should be removed
    reference RAW(size_type i) {return buf[i];}
    void PRINT () {
      cout << "c:" << bufsize << ",as:" << actsize << ",h:"<<head << ",t:" << tail << endl;
      cout << "  sequential: ";
      for (size_type i = 0; i < actsize; i++)
        cout << buf[pos_from_rank(i)] << " ";
      cout << endl;
    }
    /// Allocates new buffer of double size and moves elements to front of this buffer
    void grow_to_double();
  private:
    /// Pointer to memory block
    T *buf;
    /// Index to head of array
    size_type head;
    /// Index to tail of array
    size_type tail;
    /// Number of actual elements
    size_type actsize;
    /// Capacity
    size_type bufsize;
    /// Mask used for fast \%-operations (assumes \c bufsize = \f$2^x\f$)
    size_type mask;
    /// Allocator used for allocating/deallocating memory and for constructing and destroying element
    A alloc;

    /// Calculates an elements position in memory from its rank in the sequence using modulus-operati
    size_type pos_from_rank(size_type pos);
    /// Moves a range of elements forward in the memory block, modulo \c bufsize
    void move_forward(size_type, size_type, size_type);
    /// Moves a range of elements backwards in the memory block, modulo \c bufsize
    void move_back(size_type, size_type, size_type);

  private: //declare copy constructor and assignment private for now
    level1_tieredvector(level1_tieredvector&);
    level1_tieredvector* operator=(level1_tieredvector);
  };

  //----------- construct/destroy/copy
  template <typename T, typename A>
  inline level1_tieredvector<T, A>::level1_tieredvector(size_type bz)
    : head(-1), tail(-1), actsize(0), bufsize(bz), mask(bufsize-1), alloc(A())
  {
    assert((bufsize & mask) == 0);
    buf = alloc.allocate(bufsize); //make space for bufsize Ts, but don't new them
  }

  template <typename T, typename A>
  inline level1_tieredvector<T, A>::~level1_tieredvector()
```

```
{
  //destroy all elements, but don't deallocate
  if (head > tail) {
    for (size_type i = head; i !=  bufsize-1; i++)
      alloc.destroy(buf + i);
    for (size_type i = 0; i !=  tail+1; i++)
      alloc.destroy(buf + i);
  } else {
    for (size_type i = head; i !=  tail; i++)
      alloc.destroy(buf + i);
  }
  //deallocate
  alloc.deallocate(buf, bufsize);
}

//------------------ helper functions
template <typename T, typename A>
inline level1_tieredvector<T, A>::size_type level1_tieredvector<T, A>::pos_from_rank(size_type rank)
{
  return (head + rank) & mask;
}

//moves all elements from start to end (both incl.) dist cells forward
template <typename T, typename A>
inline void level1_tieredvector<T, A>::move_forward(size_type start,
                                                    size_type end, size_type dist)
{
  for (size_type i = end; i != (start-1); i--) {
    alloc.construct(buf + pos_from_rank(i+dist), buf[pos_from_rank(i)]);
    alloc.destroy(buf + pos_from_rank(i));
  }
}

//moves all elements from start to end (both incl.) dist cells back
template <typename T, typename A>
inline void level1_tieredvector<T, A>::move_back(size_type start,
                                                 size_type end, size_type dist)
{
  for (size_type i = start; i <= end; i++) {
    alloc.construct(buf + pos_from_rank(i+bufsize-dist), buf[pos_from_rank(i)]);
    alloc.destroy(buf + pos_from_rank(i));
  }
}


//doubles the allocated size
template <typename T, typename A>
inline void level1_tieredvector<T, A>::grow_to_double()
{
  size_type newbufsize = bufsize*2;
  T *newbuf = alloc.allocate(newbufsize); //exception here?
  for (size_type i = 0; i < actsize; i++) {
    alloc.construct(newbuf + i, buf[pos_from_rank(i)]);
    alloc.destroy(buf + pos_from_rank(i));
  }
  alloc.deallocate(buf, bufsize);
  //if all is well update variables
  head = 0;
  tail = actsize - 1;
  bufsize = newbufsize;
  buf = newbuf;
  mask = bufsize - 1;
}
//---------- element access
template <typename T, typename A>
inline level1_tieredvector<T, A>::reference level1_tieredvector<T, A>::operator[](size_type i)
{
  return buf[pos_from_rank(i)];
}

template <typename T, typename A>
inline level1_tieredvector<T, A>::reference level1_tieredvector<T, A>::front ()
{
  return buf[head];
}

template <typename T, typename A>
inline level1_tieredvector<T, A>::reference level1_tieredvector<T, A>::back ()
{
  return buf[tail];
}


//---------- modifiers
template <typename T, typename A>
inline void level1_tieredvector<T, A>::insert(size_type rank, const T& elem)
{
  assert(actsize != bufsize); //make sure there is room

  if (actsize > 0) {
    assert((0 <= rank) && (rank < actsize));
    if (rank == 0) { //insertion in front
      head = (head + bufsize - 1) & mask;
      alloc.construct(buf + head, elem);
    } else {
      if (rank < actsize - rank) {
        //fewer elements before rank
        move_back(0, rank - 1, 1);
        head = (head - 1) & mask;
      } else {
        //fewer elements after rank
        move_forward(rank, actsize-1, 1);
        tail = (tail + 1) & mask;
      }
      alloc.construct(buf + pos_from_rank(rank), elem);
    }
  } else {
    assert(0 == rank);
    alloc.construct(buf, elem);
    head = tail = 0;
  }
  actsize++;
}

template <typename T, typename A>
inline void level1_tieredvector<T, A>::erase(size_type rank)
{
  assert((0 <= rank) && (rank < actsize));
  alloc.destroy(buf + pos_from_rank(rank));
  if (rank == 0) { //erase from front
    head = (head + 1) & mask;
  } else if (rank == actsize-1){ //erase from back
    tail = (tail + bufsize - 1) & mask;
  } else {
    if (rank < actsize - rank) {
      //fewer elements before rank
      move_forward(0, rank - 1, 1);
      head = (head + 1) & mask;
    } else {
      //fewer elements after rank
      move_back(rank+1, actsize-1, 1);
      tail = (tail - 1) & mask;
    }
  }
  actsize--;
}

template <typename T, typename A>
inline void level1_tieredvector<T, A>::push_front(const T& elem)
{
  insert(0, elem);
}

template <typename T, typename A>
inline void level1_tieredvector<T, A>::push_back(const T& elem)
{
  assert(actsize != bufsize); //make sure there is room
  if (actsize > 0) {
    tail = (tail + 1) & mask;
    alloc.construct(buf + tail, elem);
  } else {
    alloc.construct(buf, elem);
    head = tail = 0;
  }
  actsize++;
}

template <typename T, typename A>
inline void level1_tieredvector<T, A>::pop_front()
{
  alloc.destroy(buf + head);
  head = (head + 1) & mask;
  actsize--;
}

template <typename T, typename A>
inline void level1_tieredvector<T, A>::pop_back()
```

```
  {
    alloc.destroy(buf + tail);
    tail = (tail + bufsize - 1) & mask;
    actsize--;
  }

}

#endif
```

Appendix B.4 simpledeque.cpp

```
/* -*- C++ -*- $Id: simpledeque.cpp,v 1.1 2002/03/07 14:56:47 jyrki Exp $ */
/*! \ingroup deque
 * \file simpledeque.cpp
 * \brief This file contains the simpledeque namespace,
 * including the class cphstl::simpledeque::deque
 */

#include <memory>
#include <iterator>

#include "tieredvector.cpp"

namespace cphstl {

  /*! \ingroup deque
   *  \namespace cphstl::simpledeque
   *  \brief Holds the simpledeque::deque class and the
   * simpledeque::deque_iterator
   */
  namespace simpledeque {
    /*! \ingroup deque
     *  \class deque
     *  \brief A simple deque data structure, that provides amortized constant
     * time front and back operations.
     *
     * This class is basically a wrapper for a level1_tieredvector object.
     * It makes sure that the contained object grows (and shrinks) when needed.
     * \remark At the moment removing elements from the sequence does not
     * cause the sequence to shrink.
     *
     */

    /*!
     * \example container_test1.cpp
     * Tests basic functions like push/pop and random access for deque.
     */
    /*!
     * \example insert_erase_test.cpp
     * Tests erases and inserts for cphstl::simpledeque::deque using iterators
     */
    template <class T, class A = std::allocator<T> >
    class deque {
    private:
      level1_tieredvector<T, A> tv;
      void grow_if_full() {
        if (tv.is_full())
          tv.grow_to_double();
      }
    public:
      // Types
      /// \c iterator typedef, \c false passed as \c isconst template parameter to deque_iterator.
      typedef deque_iterator<T,A, false> iterator;
      /// \c const_iterator typedef, \c true passed as \c isconst template parameter to deque_iterator
      typedef deque_iterator<T,A, true> const_iterator;
      typedef T value_type;
      typedef A allocator_type;
      typedef typename A::reference reference;
      typedef typename A::const_reference const_reference;
      typedef typename A::size_type size_type;
      typedef typename A::difference_type difference_type;
      typedef typename std::reverse_iterator<iterator> reverse_iterator;
      typedef typename std::reverse_iterator<const_iterator> const_reverse_iterator;
      // Construct/Copy/Destroy
      explicit deque (const A& = A()) { }
      explicit deque (size_type);
      deque (size_type, const T& value,
             const A& = A ());
      deque (const deque<T,A>&);

      template <class InputIterator>
      deque (InputIterator, InputIterator, const A& = A ());
      ~deque () { }

      deque<T,A>& operator=(const deque<T,A>&);

      template <class InputIterator>
      void assign (InputIterator, InputIterator);

      void assign (size_type, const T&);
      allocator_type getallocator () const;
      // Iterators
      iterator begin () { return iterator(tv, 0); }
      const_iterator begin () const { return iterator(tv, 0); }
      iterator end () { return iterator(tv, tv.actsize); }
      const_iterator end () const { return iterator(tv, tv.actsize); };

      reverse_iterator rbegin ();
      const_reverse_iterator rbegin () const;
      reverse_iterator rend ();
      const_reverse_iterator rend () const;
      // Capacity
      size_type size () const { return tv.size(); }
      size_type max_size () const;

      void resize (size_type);
      void resize (size_type, T);

      bool empty () const { return tv.actsize == 0; }
      // Element access
      reference operator[] (size_type rank) { return tv[rank]; }
      const_reference operator[] (size_type rank) const { return tv[rank]; }
      reference at (size_type);
      const_reference at (size_type) const;

      reference front () { return tv.front(); }
      const_reference front () const { return tv.front(); }
      reference back () { return tv.back(); }
      const_reference back () const { return tv.back(); }
      // Modifiers
      void push_front (const T& elem) { grow_if_full(); tv.push_front(elem); }
      void push_back (const T& elem) { grow_if_full(); tv.push_back(elem); }

      iterator insert (iterator it, const T& elem) {
        grow_if_full();
        tv.insert(it.rank, elem);
        return it;
      }
      void insert (iterator, size_type, const T&);
      template <class InputIterator>
      void insert (iterator, InputIterator, InputIterator);

      void pop_front () { tv.pop_front(); }
      void pop_back () { tv.pop_back(); }

      iterator erase (iterator it) { tv.erase(it.rank); return it;}
      iterator erase (iterator, iterator);

      void swap (deque<T, A>&);
      void clear();
    };

    // Non-member Operators
    template <class T, class A>
    bool operator== (const deque<T, A>&,
                     const deque<T, A>&);
    template <class T, class A>
    bool operator!= (const deque<T, A>&,
                     const deque<T, A>&);

    template <class T, class A>
    bool operator< (const deque<T, A>&,
                    const deque<T, A>&);

    template <class T, class A>
    bool operator> (const deque<T, A>&,
                    const deque<T, A>&);
    template <class T, class A>
    bool operator<= (const deque<T, A>&,
                     const deque<T, A>&);
    template <class T, class A>
    bool operator>= (const deque<T, A>&,
                     const deque<T, A>&);

    // Specialized Algorithms
    template <class T, class A>
    void swap (deque<T, A>&, deque<T, A>&);
```

```
/*! \ingroup deque
 *  \class deque_iterator
 *  \brief Iterator class for simpledeque::deque, that implements
 *  both \c iterator and \c const_iterator.
 *
 * \c isconst template parameter determines the type
 * of \c reference using an auxillary struct \c choose.
 * \see
 * Declaration of simpledeque::deque::iterator and
 * simpledeque::deque::const_iterator. For details on the use of the
 * \c choose -struct se Matt Austerns C++ Users Journal article
 * (http://cuj.com/experts/1901/austern.html)
 */
template <bool flag, class IsTrue, class IsFalse>
struct choose;

template <class IsTrue, class IsFalse>
struct choose<true, IsTrue, IsFalse> {
  typedef IsTrue type;
};

template <class IsTrue, class IsFalse>
struct choose<false, IsTrue, IsFalse> {
  typedef IsFalse type;
};

template <class T, class A, bool isconst>
class deque_iterator {
  friend deque<T, A>;
private:
  level1_tieredvector<T, A>* tv;
  level1_tieredvector<T, A>::size_type rank;
public:
  //traits
  typedef random_access_iterator_tag iterator_category;
  typedef level1_tieredvector<T, A>::value_type value_type;
  typedef level1_tieredvector<T, A>::difference_type difference_type;
  typedef typename choose<isconst, level1_tieredvector<T, A>::const_pointer, level1_tieredvector<T
  typedef typename choose<isconst, level1_tieredvector<T, A>::const_reference, level1_tieredvector
  //construct/destruct
  deque_iterator() : tv(0), rank(0) {}
  deque_iterator(level1_tieredvector<T, A>& ntv, deque<T, A>::size_type nrank) : tv(&ntv), rank(nr
  deque_iterator(const deque_iterator& x) : tv(x.tv), rank(x.rank) {}
  ~deque_iterator() {}

  bool operator==(const deque_iterator& x) const {
    return  (tv == x.tv) && (rank == x.rank) ;
  }

  bool operator!=(const deque_iterator& x) const {
    return (rank != x.rank) || (tv != x.tv);  //check rank first because other check will usually
  }

  reference operator*() const {
        return tv->buf[tv->pos_from_rank(rank)];
      }

      pointer operator->() const {
        return &(operator*());
      }

      deque_iterator& operator++() {
        if (!(rank == tv->actsize))
          rank++;
        return *this;
      }

      deque_iterator operator++(int) {
        deque_iterator tmp = *this;
        ++*this;
        return tmp;
      }

      deque_iterator& operator--() {
        rank--; //also works for the end-iterator
        return *this;
      }

      deque_iterator operator--(int) {
        deque_iterator tmp = *this;
        --*this;
        return tmp;
      }
    };
  }
}
```

References

[1] 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 (1999).

[2] Department of Computing, University of Copenhagen, The copenhagen stl, Website accessible at `http://www.cphstl.dk/` (2001).

[3] M. T. Goodrich and J. G. K. II, Tiered vectors: Efficient dunamic arrays for rank-based sequences, *1999 Workshop on Algorithms and Data Structures (WADS)* (1999).

[4] International Organization for Standardization (ISO), *ISO/IEC 14882: Standard for the C++ Programming Language*, Genevé (1998).

[5] ISO JTC1/SC22/WG21 - C++, C++ standard library defect report list (revision 16), Worldwide Web Document (2000). Available at `http://anubis.dkuug.dk/JTC1/SC22/WG21/docs/lwg-defects.html`.

[6] Silicon Graphics Computer Systems, Inc., Standard template library programmer's guide, Worldwide Web Document (1999). Available at `http://www.sgi.com/Technology/STL/`.