

# Circular Buffer Implementation of SGI'S Deque Container

Assad Jarrhian  
Gary Moore  
Jim Percent  
Frank Vardaro  
Marcin Zalewski

December 19, 2000

# Contents

<b>1</b>	<b>User's Guide</b>	<b>1</b>
1.1	Introduction . . . . .	1
1.1.1	Overview . . . . .	1
1.1.2	Intended Audience . . . . .	2
1.2	Terms and Definitions . . . . .	2
1.2.1	User Level . . . . .	2
1.2.2	Implementation Level . . . . .	3
<b>2</b>	<b>Reference Manual</b>	<b>4</b>
2.1	Description . . . . .	4
2.1.1	Implementation Notes . . . . .	4
2.2	Example . . . . .	4
2.3	Template Parameters . . . . .	4
2.4	Model of . . . . .	4
2.5	Type Requirements . . . . .	5
2.6	Public base classes . . . . .	5
2.7	Members . . . . .	6
2.8	Iterators in APdeque . . . . .	8
<b>3</b>	<b>Design Issues</b>	<b>10</b>
3.1	Abstraction Level Design Issues . . . . .	10
3.2	Implementation Level Design Issues . . . . .	10
<b>4</b>	<b>Source Code</b>	<b>11</b>
4.1	deque's compiler directives . . . . .	11
4.2	deque_iterator . . . . .	11
4.2.1	typedefs, public data members, and constructors . . . . .	12
4.2.2	Dereference Operator * . . . . .	12
4.2.3	Post/Pre Increment and Decrement Operators (++ and --) . . . . .	13
4.2.4	Plus and Minus Operators (+ and -) . . . . .	13
4.2.5	Element Access Operator ([ ]) . . . . .	14
4.2.6	Comparison Operators (==, !=, >, <=, >=, and <) . . . . .	14
4.3	deque . . . . .	15
4.3.1	typedefs and friends . . . . .	15
4.3.2	Constructors . . . . .	15
4.3.3	Destructor . . . . .	17
4.3.4	Push Back and Push Front . . . . .	18
4.3.5	Pop Back and Pop Front . . . . .	19
4.3.6	Insert . . . . .	20
4.3.7	Resize . . . . .	22
4.3.8	Swap . . . . .	23
4.3.9	Overloaded Operator(s) . . . . .	23
4.3.10	Erase and Clear . . . . .	24

4.3.11	Size Functions	26
4.3.12	begin() and end()	26
4.3.13	front() and back()	26
4.3.14	empty()	27
4.3.15	rbegin() and rend()	27
4.3.16	Private data members and member functions	27
4.4	deque's global members (== and <)	30
<b>5</b>	<b>Test Plans and Results</b>	<b>30</b>
5.1	Overview	30
5.2	Functionality Testing	30
5.3	Possible States	31
5.3.1	StateFull	31
5.3.2	StateBegin	31
5.3.3	StateMiddle	33
5.3.4	StateWrapped	34
5.3.5	StateEmpty	35
5.4	Constructor Tests	37
5.5	Iterator Tests	38
5.6	Mutating Operation Tests	39
5.7	Insert and Delete Operations Tests	40
5.8	Performance Testing	43
5.8.1	Introduction	43
5.8.2	How Performance was Tested	43
5.8.3	Adaptor Performance	44
5.8.4	Stack Performance	48
5.8.5	Priority Queue Performance	48
5.8.6	Algorithm Performance	49
5.8.7	Sorting Performance	52
5.8.8	Random Shuffle Performance	52
5.8.9	Inserting Element Performance	53
5.8.10	Pushing and Popping Performance	58
5.8.11	Insertion Performance	66
<b>6</b>	<b>Future Work</b>	<b>69</b>
6.1	Removing Iterator Validity Guarantees Approach	69
6.2	Providing Iterator Validity Guarantees Approach	69
6.3	Summary	70
<b>A</b>	<b>Appendix A</b>	<b>72</b>
A.1	ap.h	72
A.2	hirestimer.h	72
<b>B</b>	<b>Appendix B</b>	<b>77</b>
B.1	TestPackage.h	77



# 1 User's Guide

## 1.1 Introduction

STL (Standard Template Library) is a set of concepts/type requirements that provides a wide range of generic components, algorithms, and iterators. A logical hierarchy of concepts enables developers to specify a complete set of types modeling these concepts. Concepts specify abstract requirements that a modeling component must fulfill. This allows for easy addition of new components, or concepts that fit into the existing hierarchy. Addition or re-implementation of a component only requires that the new or re-implemented component will fulfill the abstract requirements imposed by STL's concept hierarchy.

This paper will describe and show an alternative implementation of one of SGI's types, namely deque. The new implementation will be compared with the existing and well-established one in SGI's STL. From the user's point of view there is no difference in functionality between the two implementations, except for the new implementation allows for better preservation of iterator validity. Also, there is a difference in the performance, and memory usage. This is the main objective of this paper — to compare in detail the performance of the original versus the proposed implementation for several well-defined classes of input. As a result of the new implementation, there are some performance tradeoffs due to iterator validity. This modification is an improvement for end users so they do not have to worry about invalidating iterators. SGI's implementation is somewhat faster in any of the operations that involve dereferencing or pushing. However, popping and inserting operations out perform SGI's implementation.

### 1.1.1 Overview

A deque is very similar to a vector; for example, it models a sequence that supports random access to elements, constant time insertion and removal of elements at the end of the sequence, and linear time insertion and removal of elements in the middle. The difference between vector and deque is that deque guarantees amortized constant time insertion and removal of elements at the beginning of the sequence. SGI's deque as well as vector do not provide any guarantees on iterator validity when using member functions. In contrast the new implementation of deque will guarantee iterator validity for most member functions. The SGI deque is implemented as an array of pointers to fixed size blocks. Since the blocks are fixed in size, index calculations can be done at random positions. However, a level of indirection is introduced by the array of pointers. The new implementation shows that this level of indirection can be removed by implementing the deque as a circular buffer. Since the two containers, deque and vector, are very similar the following strategy is proposed: use a single vector, treated as a circular buffer, as the underlying implementation of the deque. The vector container is in many cases the most efficient of all STL containers, and since the proposed implementation of deque uses the vector container it is also efficient. In all aspects the new implementation is at least as

efficient as the existing SGI's implementation, except for dereferencing elements. The new implementation will be denoted APdeque, and the SGI implementation will be denoted SGId deque, to avoid any ambiguity throughout the paper.

### 1.1.2 Intended Audience

This paper assumes the reader should have a general knowledge of the C++ language. The following is intended for end users, as well as developers with interest in extending and/or re-implementing parts of STL. At least, a basic knowledge of templates is required to understand both usage and implementation. General knowledge of STL may be beneficial to understanding some of the points made, but not necessary. The usage of deque will be explained in the Reference Manual section.

## 1.2 Terms and Definitions

### 1.2.1 User Level

Several terms will be introduced for user level understanding of the paper. Most of those terms are explained in STL documentation however key definitions required for this paper will be repeated here.

**Random Access Container:** Reversible Container whose iterator type is a Random Access Iterator. It provides amortized constant time access to arbitrary elements.

**Sequence:** variable-sized Container whose elements are arranged in a strict linear order. It supports insertion and removal of elements.

**Front Insertion Sequence:** a Sequence where it is possible to insert an element at the beginning, or to access the first element, in amortized constant time.

**Back Insertion Sequence:** a Sequence where it is possible to add an element to the end, or to access the last element, in amortized constant time.

**Random Access Iterator:** an iterator that provides both increment and decrement, and that also provides constant-time methods for moving forward and backward in arbitrary-sized steps. Random Access Iterators provide essentially all of the operations of ordinary C pointer arithmetic.

**Vector:** a Sequence that supports random access to elements, constant time insertion and removal of elements at the end, and linear time insertion and removal of elements at the beginning or in the middle. The number of elements in a vector may vary dynamically; memory management is automatic. Vector is the simplest of the STL container classes, and in many cases the most efficient.

### 1.2.2 Implementation Level

Several terms will be introduced at the implementation level to provide a better understanding of the paper.

**Circular Buffer:** a Sequence that always has a next element, and contains a fixed number of elements. Given a circular buffer of size  $n$ , one iteration past the  $n$ th element (finish) will point to the first element (start).

**deque\_iterator:** a friend of the deque (APdeque) class which is a model of a random access iterator.

**internal\_rep:** a private data member of type `vector<T>` of APdeque, which is the underlying vector container.

**size\_block:** a private data member of type `int` of the APdeque, which is the size of the the `internal_rep`. If `size_block` is equal to the vector's `size() - 1`, then the vector is reallocated.

**start:** a private data member of type `vector<T>::iterator` of APdeque, which points to the first element of the deque.

**finish:** a private data member of type `vector<T>::iterator` of APdeque, which points to one past the last element of the deque. In an empty deque, start and finish point to the same location.

**offset:** a public data member of type `difference_type` of the `deque_iterator` class, which is the iterators distance from start.

**d\_num\_elements:** a private data member of type `int` of the APdeque, which is the number of elements the deque contains. This data member is incremented/decremented every time an element is inserted/deleted, respectively.

## 2 Reference Manual

### 2.1 Description

A deque is very similar to a vector. It is a sequence that supports random access to elements, constant time insertion and removal of elements at the end of the sequence, and linear time insertion and removal of elements in the middle. The major difference is that deque also supports constant time insertion and removal of elements at the beginning of the sequence.

#### 2.1.1 Implementation Notes

The proposed `APdeque` implementation was developed on Solaris 2.8, SUN Ultra10 system. Tests were conveyed using g++ 2.95.2 compiler. No other compilers or platforms were tested. There is no guarantee that `APdeque` will compile and run correctly if different hardware or compiler is used.

### 2.2 Example

```
#include "ap.h"
...
deque<int> Q;
Q.push_back(3);
Q.push_front(1);
Q.insert(Q.begin() + 1, 2);
Q[2] = 0;
...
//STL style of output to the screen - alternative is a loop
copy(Q.begin(), Q.end(), ostream_iterator<int>(cout, " "));
// The values that are printed are 1 2 0
```

### 2.3 Template Parameters

Parameter	Description	Default
T	The deque's value type: the type of object that is stored in the deque	
Alloc	The deque's allocator, used for all internal memory management	alloc

alloc is a default STL allocator.

### 2.4 Model of

Random access container, Front insertion sequence, Back insertion sequence.

## **2.5 Type Requirements**

None, except for those imposed by the requirements of Random access container, Front insertion sequence, and Back insertion sequence as defined in STL documentation.

## **2.6 Public base classes**

None.

## 2.7 Members

Members	Where Defined	Description
<code>value_type</code>	Container	The type of object, T, stored in the deque
<code>pointer</code>	Container	Pointer to T
<code>reference</code>	Container	Reference to T
<code>const_reference</code>	Container	Const reference to T
<code>size_type</code>	Container	An unsigned integral type
<code>difference_type</code>	Container	A signed integral type
<code>iterator</code>	Container	Iterator used to iterate through deque
<code>const_iterator</code>	Container	Const iterator used to iterate through deque
<code>reverse_iterator</code>	Reversible Container	iterator used to iterate backwards through a deque
<code>const_reverse_iterator</code>	Reversible Container	Const iterator used to iterate backwards through a deque
<code>iterator begin()</code>	Container	Returns an iterator pointing to the beginning of the deque
<code>iterator end()</code>	Container	Returns an iterator pointing to the end of the deque
<code>const_iterator begin() const</code>	Container	Returns a const_iterator pointing to the beginning of the deque
<code>const_iterator end() const</code>	Container	Returns a const_iterator pointing to the end of the deque
<code>reverse_iterator rbegin()</code>	Reversible Container	Returns a reverse_iterator pointing to the beginning of the reversed deque
<code>reverse_iterator rend()</code>	Reversible Container	Returns a reverse_iterator pointing to the end of the reversed deque

<code>const_reverse_iterator rbegin()</code> <code>const</code>	Reversible Container	Returns a <code>const_reverse_iterator</code> pointing to the beginning of the reversed <code>deque</code>
<code>const_reverse_iterator rend()</code> <code>const</code>	Reversible Container	Returns a <code>const_reverse_iterator</code> pointing to the end of the reversed <code>deque</code>
<code>size_type size() const</code>	Container	Returns the size of the <code>deque</code>
<code>size_type max_size() const</code>	Container	Returns the largest possible size of the <code>deque</code>
<code>bool empty() const</code>	Container	<code>true</code> if the <code>deque</code> 's size is 0
<code>reference operator[](size_type n)</code>	Random Access Container	Returns the n'th element
<code>const_reference operator[](size_type n) const</code>	Random Access Container	Returns the n'th element
<code>deque()</code>	Container	Creates an empty <code>deque</code>
<code>deque(size_type n)</code>	Sequence	Creates a <code>deque</code> with n elements
<code>deque(size_type n, const T&amp; t)</code>	Sequence	Creates a <code>deque</code> with n copies of t
<code>deque(const deque&amp;)</code>	Container	The copy constructor
<code>template &lt;class InputIterator&gt;</code> <code>deque(InputIterator f,</code> <code>InputIterator l)</code>	Sequence	Creates a <code>deque</code> with a copy of a range
<code>deque()</code>	Container	The <code>destructor</code>
<code>deque&amp; operator=(const deque&amp;)</code>	Container	The assignment operator
<code>reference front()</code>	Front In- sertion Sequence	Returns the first element
<code>const_reference front() const</code>	Front In- sertion Sequence	Returns the first element
<code>reference back()</code>	Back In- sertion Sequence	Returns the last element
<code>const_reference back() const</code>	Back In- sertion Sequence	Returns the last element

<code>void push_front(const T&amp;)</code>	Front In- sertion Sequence	Inserts a new element at the beginning
<code>void push_back(const T&amp;)</code>	Back In- sertion Sequence	Inserts a new element at the end
<code>void pop_front()</code>	Front In- sertion Sequence	Sequence Removes the first element
<code>void pop_back()</code>	Back In- sertion Sequence	Removes the last element
<code>void swap(deque&amp;)</code>	Container	Swaps the contents of two deques
<code>iterator insert(iterator pos, const T&amp; x)</code>	Sequence	Inserts <code>x</code> before <code>pos</code>
<code>template &lt;class InputIterator&gt; void insert(iterator pos, InputIterator f, InputIterator l)</code>	Sequence	Inserts the range <code>[f, l)</code> before <code>pos</code>
<code>void insert(iterator pos, size_type n, const T&amp; x)</code>	Sequence	Inserts <code>n</code> copies of <code>x</code> before <code>pos</code>
<code>iterator erase(iterator pos)</code>	Sequence	Erases the element at position <code>pos</code>
<code>iterator erase(iterator first, iterator last)</code>	Sequence	Erases the range <code>[first, last)</code>
<code>void clear()</code>	Sequence	Erases all of the elements
<code>void resize(n, t = T())</code>	Sequence	Inserts or erases elements at the end such that the size becomes <code>n</code>
<code>bool operator==(const deque&amp;, const deque&amp;)</code>	Forward Container	Tests two deques for equality. Compares element by element using <code>==</code> operator for each element
<code>bool operator&lt;(const deque&amp;, const deque&amp;)</code>	Forward Container	Lexicographical comparison

## 2.8 Iterators in APdeque

APdeque implements iterators in different manner than SGId deque. For this reason there exist some differences in semantics of several member functions. The following provides a short summary of APdeque's member function semantics in respect to invalidating iterators. Later, main differences in invalidating of iterators between APdeque and SGId deque are highlighted.

The semantics of iterator invalidation for deque is as follows:

- `push_front` invalidates all iterators.
- `push_back` does not invalidate any iterators except beyond the last element iterators.
- `insert` in the middle may invalidate all iterators (either before insertion position or afterwards).
- `insert` at the beginning invalidates all iterators.
- `insert` at the end does not invalidate any iterators except beyond the last element iterators.
- `erase` in the middle may invalidate all iterators (either before erase position or afterwards.)
- `erase` at the beginning (including `pop_front`) invalidates all iterators.
- `erase` at the end (including `pop_back`) invalidates only beyond the last element iterators.
- `resize` does not invalidate all iterators, only those pointing to elements changed by `resize` will invalidate.

First, one of the most important differences between APdeque and SGId deque is that `push_back`, `pop_back`, `erase` and `insert` at the end do not invalidate any iterators except the ones pointing to elements being erased, and the ones pointing beyond the last element (in case of insertion). `push_back` is one of the most commonly used member functions in many popular applications, and this additional functionality of APdeque can become very handy, and make some tasks easier.

Second, `push_front` and `pop_front` invalidate iterators by one position. That means that, for example when `pop_front` was used, all iterators can be decremented by one and still will point to the same elements (except the iterator pointing to the popped element). The situation is similar in the case of `push_front`, except iterators need to be incremented by one to point to the same elements. The situation where `insert` or `erase` are used at the beginning of a deque is very similar, except that iterators have to be incremented or decremented by the number of elements being inserted or erased. Finally, `resize` does not invalidate all iterators. If `resize` decreases the number of elements only iterators pointing to elements erased are invalidated. If elements are added the only iterator invalidated is the beyond the last element iterator. `resize` is frequently used in many standard applications, and if it does not invalidate iterators it may make many tasks easier.

### 3 Design Issues

This section describes issues encountered during the design of `APdeque`. For further clarification, this section is divided into two subsections, the Abstraction level section and the Implementation level section. The Abstraction Level section contains issues of the interface, semantics, and performance requirements. The Implementation Level, on the other hand, describes the choice of internal algorithms and data structures that meet the abstraction level requirements.

#### 3.1 Abstraction Level Design Issues

`APdeque`'s interface is the same as the one provided by `SGIdeque`. The interface is what is required by the concepts that `deque` models.

`APdeque` uses a circular buffer to achieve amortized constant front and back insertion, as well as removal. The circular buffer does not differ much from a vector, except it does not have a fixed start and end position. To insert in the front/back the start/end iterator is moved to the left/right, and an element is inserted, respectively.

`APdeque` provides additional functionality of keeping iterators valid in many cases when `SGIdeque` does not. The details of the iterator invalidation semantics are covered in Reference Manual section. To achieve this functionality `APdeque` iterator uses an offset from the beginning position of the circular buffer, rather than keeping an iterator to the underlying vector. This decision was made in order to avoid invalidation of iterators when the underlying vector gets reallocated; for example, if `deque`'s iterator would keep a pointer to the position in the underlying vector, then as soon as vector gets deallocated this position would become invalid. If this decision was not made, it would be extremely hard to keep track of iterators declared by a user and to update their values in the case where reallocating of internal vector is needed. On the other hand, if iterator keeps only the offset information it is independent from the underlying vector. The actual position of an element can be calculated when necessary using offset information.

#### 3.2 Implementation Level Design Issues

It was decided to keep the `deque`'s iterator external to the `deque` class itself. This allows for easy creation of `const_iterator` and `iterator` types in `deque` without having to implement them as two different classes. The only difference between the `const` and regular iterator is the template arguments they take. `deque_iterator` class takes argument parameters for pointer and reference types. By passing `const T*` and `const T&` as the mentioned template parameters, it is easy to create `const` iterators. `iterator` and `const_iterator` were made `deque`'s friends. Hence, the iterator may access `deque`'s private members that are needed to calculate position of an element in the underlying vector.

Additional design decisions will be described in detail throughout the Source Code section.

## 4 Source Code

### 4.1 deque's compiler directives

First, APdeque includes the header file ap.h (See Appendix A) which contains all necessary header files. Class deque is required to have a forward definition in order for the deque\_iterator to have a data member of type deque.

⟨deque compiler directives ?⟩ ≡

```
#ifndef CIRCULAR_ARRAY_H_PROJECT_TEAM_UGRAD
#define CIRCULAR_ARRAY_H_PROJECT_TEAM_UGRAD

#include "ap.h"

namespace ap
{
    template<class T, class Allocator = alloc>
    class deque;
}
◇
```

Macro referenced in ?.

### 4.2 deque\_iterator

deque\_iterator takes four template arguments. The first, is the type of elements stored in the deque, second is the reference type, third is a pointer type, and fourth is the memory allocator. Class deque\_iterator inherits from random\_access\_iterator<T, ptrdiff\_t> as recommended by SGI's STL documentation to provide the basic requirements of Random Access Iterator. All typedefs listed are required by STL's specification of a Random Access Iterator. The deque\_ptr points to the deque, which the iterator was defined for. It can be a const pointer, because deque\_iterator does not modify any of deque's internal members. It only uses a pointer to access elements of the vector that need to be changed. deque\_iterator provides the default and copy constructors. In addition, it provides a constructor for the internal use of creating valid iterators.

The public data member offset is used to determine the particular element that a deque iterator is pointing to (See 1.2.2). The resultant implementation makes the deque\_iterator independent of the deque's underlying vector. The respective vector can be allocated and deallocated frequently without invalidating existing iterators. The deque class provides all information necessary to calculate the element's physical location.

### 4.2.1 typedefs, public data members, and constructors

⟨deque\_iterator typedefs, data members, and constructors ?⟩ ≡

```
template <class T, class Ref, class Ptr, class Allocator = alloc>
class deque_iterator : public random_access_iterator<T,ptrdiff_t>
{
public:
    typedef deque_iterator<T, T&, T*,Allocator> iterator;
    typedef deque_iterator<T, const T&, const T*, Allocator> const_iterator;
    typedef Ptr pointer;
    typedef Ref reference;
    typedef T value_type;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    typedef deque_iterator self;

public:
    const deque<T, Allocator> *deque_ptr;
    difference_type offset;

public:
    deque_iterator(const deque<T,Allocator> * d, difference_type n)
        : deque_ptr(d), offset(n){}
    deque_iterator() : offset(0) {}
    deque_iterator(const iterator& x) : offset(x.offset), deque_ptr(x.deque_ptr) {}
```

◇

Macro referenced in ?.

### 4.2.2 Dereference Operator \*

The dereference operator calculates the absolute position of the element in the underlying vector, and uses `vector`'s `[]` operator to access the element. Deque's member `start` specifies where the front of circular buffer is. The formula to calculate the position of element is:

$(\text{start} + \text{offset} - \text{begin}()) \% \text{size}()$

First, the distance of the element from the beginning of the vector is calculated. Then the exact position in the vector is calculated by taking the modulus of the distance of the element from the beginning with the size of `vector`.

⟨deque\_iterator operator\* ?⟩ ≡

```
reference operator*() const
{
    difference_type temp;
    temp = ((deque_ptr->start + offset) - deque_ptr->internal_rep->begin())
    % deque_ptr->internal_rep->size();
    return ((*deque_ptr).internal_rep)[temp];
}
```

◇

Macro referenced in ?.

### 4.2.3 Post/Pre Increment and Decrement Operators (++ and --)

Pre-increment and pre-decrement operators only need to perform their corresponding increment or decrement operations on the `offset`. Post-increment and post-decrement create temporary variable to be returned before actually incrementing or decrementing the `offset`.

⟨deque\_iterator increment and decrement ?⟩ ≡

```
self & operator++()
{
    ++offset;
    return *this;
}
self operator++(int)
{
    self tmp = *this;
    ++offset;
    return tmp;
}
self& operator--()
{
    --offset;
    return *this;
}
self operator--(int)
{
    self tmp = *this;
    --offset;
    return tmp;
}
```

◇

Macro referenced in ?.

### 4.2.4 Plus and Minus Operators (+ and -)

The plus and minus overloaded operators use the `offset`'s plus and minus operators as their underlying layer. Only the overloaded minus operator, which takes an `iterator` as its argument, uses the argument's iterator offset to subtract it from the instance's offset.

⟨deque\_iterator plus and minus ?⟩ ≡

```
self operator+(difference_type n) const
{
    self tmp = *this;
    tmp.offset += n;
    return tmp;
}
self& operator-=(difference_type n)
```

```

    {
        offset -= n;
        return *this;
    }
self& operator+=(difference_type n)
{
    offset += n;
    return *this;
}
self operator-(difference_type n) const
{
    self tmp = *this;
    tmp.offset -= n;
    return tmp;
}
difference_type operator-(const self & i) const
{
    return (offset - i.offset);
}

```

◇

Macro referenced in ?.

#### 4.2.5 Element Access Operator ([ ])

The overloaded reference operator takes the instance of the iterator and increments itself by *n*. The resulting iterator is dereferenced and returned.

⟨deque\_iterator operator[] ?⟩ ≡

```
reference operator[](difference_type n) const { return *(*this + n); }
```

◇

Macro referenced in ?.

#### 4.2.6 Comparison Operators (==, !=, >, <=, >=, and <)

The logical operators compare the offsets of two iterators; for example, if two iterators have the same offset they are equal, and operator== will return true. One condition that needs to be noted is that all of the logical operators will always return false if the iterators are specified for two different deques.

⟨deque\_iterator comparison operators ?⟩ ≡

```

bool operator==(const self& x) const
{ return (offset == x.offset && deque_ptr == x.deque_ptr); }
bool operator!=(const self& x) const
{ return !(offset == x.offset && deque_ptr == x.deque_ptr); }
bool operator>(const self& x) const
{ return x.offset < offset && deque_ptr == x.deque_ptr; }
bool operator<=(const self& x) const
{ return !(x.offset < offset && deque_ptr == x.deque_ptr); }

```

```

        bool operator>=(const self& x) const
    { return !(offset < x.offset && deque_ptr == x.deque_ptr); }
        bool operator<(const self& x) const
    { return x.offset > offset && deque_ptr == x.deque_ptr; }
};

```

◇

Macro referenced in ?.

## 4.3 deque

### 4.3.1 typedefs and friends

⟨deque typedefs and friends ?⟩ ≡

```

template<class T, class Allocator>
class deque
{
public:
    friend class deque_iterator<T, T& , T* , Allocator>;
    friend class deque_iterator<T, const T& , const T* , Allocator>;

    typedef T value_type;
    typedef value_type* pointer;
    typedef const value_type* const_pointer;
    typedef value_type& reference;
    typedef const value_type& const_reference;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;

    typedef deque_iterator<T, T& , T* , Allocator>::iterator iterator;
    typedef deque_iterator<T, T& , T* , Allocator>::const_iterator const_iterator;

#ifdef __STL_CLASS_PARTIAL_SPECIALIZATION
    typedef reverse_iterator<const_iterator> const_reverse_iterator;
    typedef reverse_iterator<iterator> reverse_iterator;
#else /* __STL_CLASS_PARTIAL_SPECIALIZATION */
    typedef reverse_iterator
<const_iterator, value_type, const_reference, difference_type>
const_reverse_iterator;
    typedef reverse_iterator<iterator, value_type, reference, difference_type>
reverse_iterator;
#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */

```

◇

Macro referenced in ?.

### 4.3.2 Constructors

All the following constructors take a template parameter type `T`, which is the element type stored in the deque, and allocator, which defaults to `alloc`.

The default constructor initially allocates a vector of size 100 for the internal representation of the deque, which represents the free block of space. Also, it initializes all deque's private data members (See 1.2.2) to the appropriate values.

⟨deque default constructor ?⟩ ≡

```

    deque()
    {
        internal_rep = new vector<T, Allocator>(100);
        first_push_f = 1;
        d_size_type = sizeof(T);
        size_block = 100;
        d_num_elements = 0;
        start = internal_rep->begin();
        finish = internal_rep->begin();
    }

```

◇

Macro referenced in ?.

⟨deque n element constructor ?⟩ ≡

```

    deque(int n)
    {
        internal_rep = new vector<T,Allocator>(n);
        first_push_f = 1;
        d_size_type = sizeof(T);
        size_block = n;
        d_num_elements = 0;
        start = internal_rep->begin();
        finish = internal_rep->begin();
    }

```

◇

Macro referenced in ?.

⟨deque copy n constructor ?⟩ ≡

```

    deque( size_type n, const T & x )
    {
        copy_n((size_type) n, (value_type) x);
    }

```

◇

Macro referenced in ?.

The copy constructor achieves its task in three steps. First, it allocates a new vector of the same size as deque d (the argument). Second, it copies the elements of deque d into the newly created vector via the STL copy algorithm. Third, it initializes all deque's private data members (See 1.2.2) to the appropriate values.

⟨deque copy constructor ?⟩ ≡

```

        deque( deque & d)
    {
        internal_rep = new vector<T,Allocator>( d.d_num_elements );
        copy(d.begin() , d.end() , internal_rep->begin());
        first_push_f = d.first_push_f;
        d_size_type = d.d_size_type;
        size_block = d.size_block;
        d_num_elements = d.d_num_elements;
        start = internal_rep->begin();
        finish = (internal_rep->begin() + d_num_elements);
    }

```

◇

Macro referenced in ?.

The copy range constructor is similar to the copy constructor mentioned above. It differs in the fact that it copies a given range instead of arbitrarily copying the entire deque. This is done by calling the `copy_dispatch` private member function (See 4.3.16) with the range `[first, last)` as part of its input parameters.

⟨deque copy range constructor ?⟩ ≡

```

        template<class _InputIterator>
        deque(_InputIterator f ,_InputIterator l )
    {
        typedef typename _Is_integer<_InputIterator>::_Integral _Integral;
        copy_dispatch(f, l, _Integral());
    }

```

◇

Macro referenced in ?.

### 4.3.3 Destructor

The destructor needed to be manually defined because the design called for the constructor to manually allocate physical memory using the `new` operator. It is interesting to note that only the manually allocated memory that was allocated needed to be deleted. This is because all other memory allocations would be deleted automatically since calling the `delete()` on the `internal_rep` triggers the destructor of the underlying vector.

⟨deque destructor ?⟩ ≡

```

        ~deque()
    {
        delete internal_rep;
    }

```

◇

Macro referenced in ?.

#### 4.3.4 Push Back and Push Front

`push_back()` first checks to see if the deque is logically full. If so, it resizes the deque by calling the private methods `realloc()`, `reinstantiate()`, and `reassign_pointers()`. Whether or not the deque has resized, the logical representation of the circular buffer is guaranteed to be maintained as a pre-condition of push back and a post-condition of the private methods mentioned above. After pushing the element to the back of the deque it must ensure the logical representation of the deque is maintained. This is done by making sure that `finish` is either incremented or set to point to the beginning of the underlying storage container(the vector), and the number of elements in the deque must be incremented.

`<deque push back ?> ≡`

```
void push_back(const T & element)
{
    if ( d_num_elements == (size_block - 1) )
    {
        realloc();
        reinstantiate();
        reassign_pointers();
    }
    *finish = element;
    if ( (finish + 1) == internal_rep->end() )
        finish = internal_rep->begin();
    else
        ++finish;
    ++d_num_elements;
    return;
}
```

◇

Macro referenced in ?.

`push_front()`, first checks whether the deque is full (similar to `push_back()`), and resizes if it is. Secondly, it handles the case when the deque is empty. If the deque is empty this means that `start` and `finish` point to the same element. This could cause confusion if this occurs at the end of the physical storage container(`finish` must wrap to the beginning). If the deque is not empty then the problematic case is when `start` points to the beginning(`start` must wrap to the end).

`<deque push front ?> ≡`

```
void push_front(const T & element)
{
    if ( d_num_elements == (size_block - 1) )
```

```

    {
        realloc();
        reinstantiate();
        reassign_pointers();
    }
if ( d_num_elements == 0 )
    {
        first_push_f = 0;
        *start = element;
        ++d_num_elements;
        if( finish == (internal_rep->end() - 1) )
finish = internal_rep->begin();
        else
        ++finish;
    }
else
    {
        if( start == internal_rep->begin() )
        {
            start = (internal_rep->end() - 1);
            *start = element;
            ++d_num_elements;
        }
        else
        {
            --start;
            *start = element;
            ++d_num_elements;
        }
    }
return;
}

```

◇

Macro referenced in ?.

#### 4.3.5 Pop Back and Pop Front

`pop_back()` moves the finish iterator to the appropriate place, maintaining the logical representation while decrementing the number of elements in the deque.

`<deque pop back ?>` ≡

```

    void pop_back()
    {
        if ( finish == internal_rep->begin() )
            finish = (internal_rep->end() - 1);
        else
            --finish;
        --d_num_elements;
        return;
    }

```

```
}
```

◇

Macro referenced in ?.

`pop_front()` moves the start iterator to the appropriate place, maintaining the logical representation, and decrements the number of elements in the deque.

⟨deque pop front ?⟩ ≡

```
void pop_front()
{
    if ( (start + 1) == internal_rep->end() )
        start = internal_rep->begin();
    else
        ++start;
    --d_num_elements;
    return;
}
```

◇

Macro referenced in ?.

#### 4.3.6 Insert

`insert()` has the fairly complicated task of shifting all the elements, that occur before the position pointed to by the parameter `pos` while transparently maintaining the underlying logical representation. When the deque becomes full, it resizes by calculating whether to shift towards the beginning or end of the physical storage (depending on the state). Then it increments the number of element so that the iterators remain valid; for example, they are calculated by the offset, which is dependent upon the number of elements.

⟨deque insert x before pos ?⟩ ≡

```
iterator insert( iterator pos, const T & x )
{
    iterator orig = pos;
    iterator cur;
    T temp;
    T temp2;
    difference_type begin_diff;
    difference_type end_diff;
    if ( d_num_elements == (size_block - 1) )
    {
        realloc();
        reinstantiate();
        reassign_pointers();
    }
    begin_diff = pos - begin();
    end_diff   = end() - pos;
    ++d_num_elements;
}
```

```

if ( end_diff < begin_diff )
{
    if( (finish + 1) == internal_rep->end() )
finish = internal_rep->begin();
    else
++finish;
    cur = (end() - 1);
    while( cur != pos )
    {
        *cur = *(cur-1);
        --cur;
    }
    *pos = x;
}
else
{
    if ( start == internal_rep->begin() )
start = (internal_rep->end() - 1);
    else
--start;
    cur = begin();
    pos;
    while(cur != (pos) )
    {
        *cur = *(cur + 1);
        ++cur;
    }
    *(pos) = x;
}
return orig;
}

```

◇

Macro referenced in ?.

This overloaded insert member function simply calls the above insert n times with the value x.

⟨deque insert n copies of x before pos ?⟩ ≡

```

void insert(iterator pos, size_type n , const T & x)
{
    for(int i=0; i < n; i++)
        pos = insert(pos, x);
    return;
}

```

◇

Macro referenced in ?.

⟨deque insert range before pos ?⟩ ≡

```

template<class _InputIterator>
void insert( iterator pos, _InputIterator f, _InputIterator l )
{
    typedef typename _Is_integer<_InputIterator>::_Integral _Integral;
insert_dispatch( pos, f, l, _Integral());
}

template <class _Integer>
void insert_dispatch(iterator pos, _Integer n, _Integer x, __true_type)
{
    insert(pos, (size_type) n, (value_type) x);
}

template <class _InputIterator>
void insert_dispatch(iterator pos, _InputIterator first, _InputIterator last,
__false_type)
{
    while( first != last )
    {
        insert(pos , *first );
        ++first;
    }
    return;
}

```

◇  
Macro referenced in ?.

#### 4.3.7 Resize

The `resize` deque member function follows the same schematic implementation of the original STL deque. The difference here is in the parameters that are passed to `erase` and `insert`. The implementation makes use of the `begin()` and `end()` functions to retrieve the necessary (as in the STL) parameter deque iterators for it to functions correctly. Note that the `resize` member function is overloaded to handle a user defined const value type (this is the same as STL).

⟨deque `resize` ?⟩ ≡

```

void resize(size_type new_size, const value_type& x)
{
    const size_type length = size();
    if (new_size < length)
        erase((begin()) + new_size, end());
    else
        insert(end(), new_size - length, x);
}

void resize(size_type new_size) { resize(new_size, value_type()); }

```

◇  
Macro referenced in ?.

### 4.3.8 Swap

`swap`, swaps all the private data members of the deque with the deque given as a reference parameter.

`<deque swap ?>`  $\equiv$

```
void swap(deque& x)
{
    std::swap(start, x.start);
    std::swap(finish, x.finish);
    std::swap(size_block, x.size_block);
    std::swap(d_size_type, x.d_size_type);
    std::swap(d_num_elements, x.d_num_elements);
    std::swap(first_push_f, x.first_push_f);
    std::swap(internal_rep, x.internal_rep);
    std::swap(temp_resize, x.temp_resize);
}
```

◇

Macro referenced in ?.

### 4.3.9 Overloaded Operator(s)

`operator=` first tests to make sure that the deque that it is copying is not the same. If it is not, it makes a copy in the same way as the copy constructor.

`<deque operator= ?>`  $\equiv$

```
deque & operator=( deque& d)
{
    if (&d != this)
    {
        delete internal_rep;
        internal_rep = new vector<T,Allocator>( d.d_num_elements );
        copy(d.begin() , d.end() , internal_rep->begin());
        first_push_f = d.first_push_f;
        d_size_type = d.d_size_type;
        size_block = d.size_block;
        d_num_elements = d.d_num_elements;
        start = internal_rep->begin();
        finish = (internal_rep->begin() + d_num_elements);
    }
    return *this;
}
```

◇

Macro referenced in ?.

The element access operator calls a `deque_iterator` constructor with an offset of `n`, and returns the element that `tmp` points to. This member function also supports a `const` reference using operator overloading.

⟨deque operator[] ?⟩ ≡

```
        reference operator[](size_type n)
    {
        iterator tmp(this , n );
        return *tmp;
    }

        const_reference operator[](size_type n) const
    {
        const_iterator tmp(this , n );
        return *tmp;
    }
```

◇

Macro referenced in ?.

#### 4.3.10 Erase and Clear

First, erase finds the distance from the beginning of the deque to the position of the element to be erased. If that distance is less than half the size of the deque, then each element from position - 1 to the beginning of the deque is shifted right one element and the start position of the deque is incremented by one. Else, if that distance is greater than half the size of the deque, then each element from position + 1 to the end of the deque is shifted left one element, and the finish position of the deque is decremented by one. Finally, the size of the deque (`d_num_elements`) is decremented by one, and the position (`next`) of the element immediately following the erased element is returned.

⟨deque erase ?⟩ ≡

```
        iterator erase(iterator pos)
    {
        iterator next = pos;
        difference_type index = (pos - begin());
        if (size_type(index) < (this->size() >> 1))
        {
            while(pos!=begin())
            {
                *pos = *(pos-1);
                pos--;
            }
            if (start+1 == internal_rep->end())
                start = internal_rep->begin();
            else
                start++;
        }
        else
        {
            while (pos!=end()-1)

```

```

    {
        *pos = *(pos+1);
        pos++;
    }
    if (finish == internal_rep->begin())
        finish = internal_rep->end()-1;
    else
        finish--;
    }
d_num_elements--;
return next;
}

```

◇

Macro referenced in ?.

Erase range, erases each element from **first** to **last** by calling the erase member function (described above), and returns the position immediately following **last**.

⟨deque erase range ?⟩ ≡

```

        iterator erase(iterator first, iterator last)
    {
        iterator temp=last;
        iterator next = first;
        while(next != temp)
        {
            if (!empty())
            {
                next = erase(next);
                temp--;
            }
        }
        return next;
    }

```

◇

Macro referenced in ?.

The **clear()** member function erases all elements, [**begin()**, **end()**), of the deque by calling the erase range member function (described above).

⟨deque clear ?⟩ ≡

```

        void clear()
    {
        erase(begin(),end());
    }

```

◇

Macro referenced in ?.

#### 4.3.11 Size Functions

`size()` returns the `d_num_elements` (See 1.2.2) in the deque. `max_size()` returns the maximum the deque could be.

⟨deque size functions ?⟩ ≡

```
        size_type size() const
    {
        return d_num_elements;
    }

        size_type max_size() const
    {
        return (size_type(-1) );
    }
```

◇

Macro referenced in ?.

#### 4.3.12 `begin()` and `end()`

The `begin()/end()` member function calls a `deque_iterator` constructor with an offset of `0/d_num_elements`, and returns the iterator that points to the beginning/end of the deque, respectively. These member functions also supports constant iterators using function overloading.

⟨deque begin and end ?⟩ ≡

```
        iterator begin() {return iterator(this , 0);}
        iterator end() {return iterator(this , d_num_elements );}
        const_iterator begin() const {return const_iterator(this , 0);}
        const_iterator end() const {return const_iterator(this , d_num_elements );}
```

◇

Macro referenced in ?.

#### 4.3.13 `front()` and `back()`

`front()` dereferences what the `begin()` iterator points to. `back()` creates a temporary iterator (`tmp`) assigned to `end()`, decrements it by one since `end()` does not point to any elements in the deque, and returns the dereferenced the temporary iterator. This member functions also support a const reference using function overloading.

⟨deque front and back ?⟩ ≡

```
        reference front() {return *begin();}
        reference back()
    {
        iterator tmp = end();
        --tmp;
```

```

        return *tmp;
    }
    const_reference front() const {return *begin();}
    const_reference back() const
    {
        const_iterator tmp = end();
        --tmp;
        return *tmp;
    }
}

```

◇

Macro referenced in ?.

#### 4.3.14 empty()

`empty()` checks if `start` and `finish` (See 1.2.2) point to the same location, and if they do the function returns true, otherwise false.

⟨deque empty ?⟩ ≡

```

        bool empty() const {return start==finish;}
    }

```

◇

Macro referenced in ?.

#### 4.3.15 rbegin() and rend()

The `rbegin()/rend()` member function calls a `deque_iterator` copy constructor with `begin()/end()`, and returns the `reverse_iterator` that points to the end/beginning of the deque, respectively. These member functions also supports constant iterators using function overloading.

⟨deque rbegin and rend ?⟩ ≡

```

        reverse_iterator rbegin() {return reverse_iterator(end());}
        reverse_iterator rend() {return reverse_iterator(begin());}
        const_reverse_iterator rbegin() const {return reverse_iterator(begin());}
        const_reverse_iterator rend() const {return const_reverse_iterator(end());}
    }

```

◇

Macro referenced in ?.

#### 4.3.16 Private data members and member functions

The following private data members are described in section 1.2.2.

⟨deque private data members ?⟩ ≡

```

private:
    int size_block;
    int d_size_type;
    int d_num_elements;

```

```

int first_push_f;

vector<T,Allocator>::iterator start;
vector<T,Allocator>::iterator finish;

vector<T, Allocator> *internal_rep;
vector<T, Allocator> *temp_resize;

```

◇

Macro referenced in ?.

The member functions `realloc()`, `reinstantiate()`, and `reassign_pointers()`, are the member functions associated with re-growing the APdeque. The `realloc()` member function allocates a vector of twice the size of the current `internal_rep` vector. Next `reinsantiate()` copies all of the elements of the current vector into the newly allocated vector. Finally, and to conclude the reallocation process, `reassign_pointers()` releases the memory associated with the current `internal_rep`, and then reassigns the vector pointers `start` and `finish`.

⟨deque realloc ?⟩ ≡

```

void realloc()
{
size_block *= 2;
temp_resize = new vector<T, Allocator>(size_block);
return;
}

```

◇

Macro referenced in ?.

⟨deque reinstantiate ?⟩ ≡

```

void reinstantiate()
{
copy(begin() , end() , temp_resize->begin());
return;
}

```

◇

Macro referenced in ?.

⟨deque reassign\_pointers ?⟩ ≡

```

void reassign_pointers()
{
delete internal_rep;
internal_rep = temp_resize;
start = internal_rep->begin();
finish = (internal_rep->begin() + d_num_elements);
temp_resize = NULL;
return;
}

```

◇

Macro referenced in ?.

⟨deque copy\_dispatch1 ?⟩ ≡

```
template <class _Integer>
void copy_dispatch(_Integer n, _Integer x, __true_type)
{
    copy_n((size_type) n, (value_type) x);
}
```

◇

Macro referenced in ?.

⟨deque copy\_dispatch2 ?⟩ ≡

```
template <class _InputIterator>
void copy_dispatch(_InputIterator first, _InputIterator last, __false_type)
{
    internal_rep = new vector<T,Allocator>();
    first_push_f = 1;
    d_size_type = sizeof(T);
    d_num_elements = 0;
    while (first != last)
    {
        internal_rep->push_back(*first);
        ++first;
        d_num_elements++;
    }
    size_block = d_num_elements;
    start = internal_rep->begin();
    finish = (internal_rep->end() - 1);
    return;
}
```

◇

Macro referenced in ?.

⟨deque copy\_n ?⟩ ≡

```
void copy_n(size_type n, value_type x)
{
    internal_rep = new vector<T,Allocator>(n , x);
    d_size_type = sizeof(T);
    size_block = n;
    d_num_elements = n;
    start = internal_rep->begin();
    finish = internal_rep->end()-1;
}
```

```
};//end deque class
```

◇

Macro referenced in ?.

## 4.4 deque's global members (== and <)

The deque's global member functions include operator ==, and <. This allows any two deques instantiated to be logically compared using these two operations. `operator==` checks if the size of both deques are the same, and if each element is equal using the generic function `equal`. `operator<` does a lexicographical compare of each element using the generic function `lexicographical_compare`.

⟨deque global members ?⟩ ≡

```
template <class T, class Allocator>
inline bool operator==(const deque<T, Allocator>& x, const deque<T, Allocator>& y)
{
    return x.size() == y.size() && equal(x.begin(), x.end(), y.begin());
}

template <class T, class Allocator>
inline bool operator<(const deque<T, Allocator>& x, const deque<T, Allocator>& y)
{
    return lexicographical_compare(x.begin(), x.end(), y.begin(), y.end());
}
}/* end namespace */
#endif
◇
```

Macro referenced in ?.

## 5 Test Plans and Results

### 5.1 Overview

The following tests mainly involved comparing APdeque with the SGId deque. However, for every test that validates the APdeque with SGId deque, the three possible states (See section 5.3) that APdeque can attain must be considered. Based on APdeque's design, the performance testing does not require APdeque to be in all three states. This is because the check being performed when the internal representation is in a non-regular state will not degrade the performance, since its done regardless of how it is oriented. The testing platform used to verify APdeque's validity and performance calculations is given in one class called **Test**. This class provides timing and performance functionality to generate log files to see how APdeque compares with SGId deque and SGIVector.

### 5.2 Functionality Testing

Testing for validity is done by asserting whether the APdeque has performed the same operations as SGId deque. All boundaries for the iterators and other members of the container must be considered. Due to the limitations of modern computers, some tests are done with as much rigor as possible, but they do

not reflect all possible theoretical boundary tests. For example, to test the containers resize function, it would be trivial to figure out that one boundary test is `resize(0)` and `resize(max_size)`. The problem is that most computers do not have the capacity to hold `max_size` amount of elements in the container. An alternative approach is to test inductively. In other words, to test the `resize()` member of `APdeque` all that is needed is to test at 0,1,2,4, etc.

### 5.3 Possible States

There are four possible states that are inherent to `APdeque`, since it is a circular buffer. All states must be boundary tested in order to verify `APdeque`'s validity.

#### 5.3.1 StateFull

To say `APdeque` is full is a misnomer. In actuality `APdeque` uses one space in the container to hold the last pointer to facilitate resizing efficiently. Therefore, when calling this method, in class `Test`, it will actually fill the container with `size() - 1` amount of elements. It is important to note that `StateFull` is also the same state as when the container contains space at the end.



Internal representation of Container when full

`<StateFull ?> ≡`

```
void
  StateFull(Container &C,int size)
  {
    StateEmpty(C);
    for(int i = 0; i < size; i++)
    {
      C.push_back(i);
    }
    return;
  }
```

◇

Macro referenced in ?.

#### 5.3.2 StateBegin

`StateBegin` will shift the elements already in the container to make all the free space at the beginning of the internal representation of the container. `StateBegin` will first fill the container with elements using `StateFull`, and then

it will perform a couple of `pop_front()` operations followed by one `push_back()` operation.



**Internal representation of Container when free space is in the beginning**

⟨StateBegin ?⟩ ≡

```

void
  StateBegin(Container &C,int size)
  {
    StateEmpty(C);
    StateFull(C,size);
    C.pop_front();
    C.pop_front();
    C.pop_front();
    C.pop_front();
    C.pop_front();
    C.pop_front();
    C.pop_front();
    C.pop_front();
    C.pop_front();
    C.pop_front();
    C.push_back(size);
    return;
  }

```

◇

Macro referenced in ?.

### 5.3.3 StateMiddle

StateMiddle will place all of the elements in the middle, leaving space in the front and the back. Similar to StateBegin, StateMiddle will use StateFull to fill the container, and then pop element from the front and back.



**Internal representation of Container when elements are in the middle**

⟨StateMiddle ?⟩ ≡

```
void
  StateMiddle(Container &C,int size)
  {
    StateEmpty(C);
    StateFull(C,size);
    C.pop_front();
    C.pop_front();
    C.pop_front();
    C.pop_front();
    C.pop_front();
    C.pop_back();
    C.pop_back();
    C.pop_back();
    C.pop_back();
    C.pop_back();
    return;
  }
```

◇

Macro referenced in ?.

### 5.3.4 StateWrapped

**StateWrapped** will place the free space in the middle of the container. It will call **StateFull**, and then perform the appropriate operations to put the free space in the middle of the container.



**Internal representation of Container when free space is at the middle**

⟨StateWrapped ?⟩ ≡

```
void
  StateWrapped(Container &C,int size)
  {
    StateEmpty(C);
    StateFull(C,size);
    C.pop_front();
    C.pop_front();
    C.pop_front();
    C.pop_front();
    C.pop_front();
    C.push_back(size);
    C.push_back(size+1);
    C.push_back(size+2);
    C.push_back(size+3);
    C.push_back(size+4);
    return;
  }
```

◇

Macro referenced in ?.

### 5.3.5 StateEmpty

StateEmpty is a trivial state, that will pop all n elements of the container.

⟨StateEmpty ?⟩ ≡

```
void
  StateEmpty(Container &C)
  {
    for (int i = 0; i < C.size(); i++)
    {
      C.pop_back();
    }
    return;
  }
```

◇

Macro referenced in ?.

The following is a small program to demonstrate the states.

"testStates.C" ? ≡

```
#include "TestPackage.h"

int main()
{
  ap::deque<int> apdeque;
  std::deque<int> stddeque;
  Test<ap::deque<int> > aptest;
```

```

Test<std::deque<int> > stdtest;

aptest.StateFull(apdeque,100);
stdtest.StateFull(stddeque,100);

std::cout << "Here are the elements of AP's deque." << std::endl;
std::copy(apdeque.begin(),apdeque.end(),ostream_iterator<int>(std::cout," "));
std::cout << std::endl;
std::cout << "Here are the elements of STD's deque." << std::endl;
std::copy(stddeque.begin(),stddeque.end(),ostream_iterator<int>(std::cout," "));
std::cout << std::endl;

aptest.StateBegin(apdeque,100);
stdtest.StateBegin(stddeque,100);

std::cout << "Here are the elements of AP's deque." << std::endl;
std::copy(apdeque.begin(),apdeque.end(),ostream_iterator<int>(std::cout," "));
std::cout << std::endl;
std::cout << "Here are the elements of STD's deque." << std::endl;
std::copy(stddeque.begin(),stddeque.end(),ostream_iterator<int>(std::cout," "));
std::cout << std::endl;

aptest.StateWrapped(apdeque,100);
stdtest.StateWrapped(stddeque,100);

std::cout << "Here are the elements of AP's deque." << std::endl;
std::copy(apdeque.begin(),apdeque.end(),ostream_iterator<int>(std::cout," "));
std::cout << std::endl;
std::cout << "Here are the elements of STD's deque." << std::endl;
std::copy(stddeque.begin(),stddeque.end(),ostream_iterator<int>(std::cout," "));
std::cout << std::endl;

aptest.StateMiddle(apdeque,100);
stdtest.StateMiddle(stddeque,100);

std::cout << "Here are the elements of AP's deque." << std::endl;
std::copy(apdeque.begin(),apdeque.end(),ostream_iterator<int>(std::cout," "));
std::cout << std::endl;
std::cout << "Here are the elements of STD's deque." << std::endl;
std::copy(stddeque.begin(),stddeque.end(),ostream_iterator<int>(std::cout," "));
std::cout << std::endl;

aptest.StateEmpty(apdeque);
stdtest.StateEmpty(stddeque);

if (apdeque.empty())
    std::cout << "apdeque is empty..." << std::endl;
else
    std::cout << "apdeque is empty..." << std::endl;

```

```

    if (stddeque.empty())
        std::cout << "stddeque is empty..." << std::endl;
    else
        std::cout << "stddeque is empty..." << std::endl;

    return 0;
}
◇

```

## 5.4 Constructor Tests

The set of constructors for a deque consists of the default, n element, copy, copy n, and range constructors. Testing these members is trivial with the use of STL's equal algorithm. The following are a few tests to ensure that APdeque complies with the SGIdeque.

"testConstructors.C" ? ≡

```

#include "TestPackage.h"

int main()
{
    ap::deque<int> apdequeDefault;
    std::deque<int> stddequeDefault;
    Test<ap::deque<int> > aptest;
    Test<std::deque<int> > stdttest;

    std::cout << "apdequeDefault's size by default is "
        << apdequeDefault.size() << std::endl;
    std::cout << "stddequeDefault's size by default is "
        << stddequeDefault.size() << std::endl;

    if ((apdequeDefault.empty()) && (stddequeDefault.empty()))
        std::cout << "Both deque's are empty as they should be." << std::endl;
    else
        std::cout << "One of the deque's is not empty." << std::endl;

    ap::deque<int> apdequeN(37);
    std::deque<int> stddequeN(37);
    std::cout << "apdequeN's size by N is " << apdequeDefault.size() << std::endl;
    std::cout << "stddequeN's size by N is " << stddequeDefault.size() << std::endl;

    if ((apdequeDefault.empty()) && (stddequeDefault.empty()))
        std::cout << "Both deque's are empty as they should be." << std::endl;
    else
        std::cout << "One of the deque's is not empty." << std::endl;

    aptest.StateFull(apdequeDefault,100);
    stdttest.StateFull(stddequeDefault,100);
    ap::deque<int> apdequeCopy(apdequeDefault);
}

```

```

if (std::equal(apdequeCopy.begin(), apdequeCopy.end(), apdequeDefault.begin()))
    std::cout << "The Copy Constructor worked perfectly" << std::endl;
else
    std::cout << "XXX The Copy Constructor DID NOT work perfectly XXX" << std::endl;

ap::deque<int>::iterator apdequeFirst = apdequeDefault.begin() + 10;
ap::deque<int>::iterator apdequeLast = apdequeDefault.begin() + 40;
ap::deque<int> apdequeRange(apdequeFirst, apdequeLast);
if (std::equal(apdequeFirst, apdequeLast, apdequeRange.begin()))
    std::cout << "The Range Constructor worked perfectly" << std::endl;
else
    std::cout << "XXX The Range Constructor DID NOT work perfectly XXX" << std::endl;

std::deque<long int> stddequeCopyn(100, -1);
ap::deque<long int> apdequeCopyn(100, -1);
if (std::equal(apdequeCopyn.begin(), apdequeCopyn.end(), stddequeCopyn.begin()))
    std::cout << "The Copyn Constructor worked perfectly" << std::endl;
else
    std::cout << "XXX The Copyn Constructor DID NOT work perfectly XXX" << std::endl;

apdequeDefault.~deque(); // just check if it runs ... algorithmically correct

return 0;
}
◇

```

## 5.5 Iterator Tests

Testing iterators involves performing operations on APdeque's member functions that return iterators. The following are some tests to verify that APdeque's iterators are working exactly like SGIdeque's iterators, using certain member functions. Some tests in here may not seem like iterator tests, but they actually are because of the underlying implementation of APdeque.

"testIterators.C" ? ≡

```

#include "TestPackage.h"

int main()
{
    ap::deque<int> apdeque;
    std::deque<int> stddeque;
    Test<ap::deque<int> > aptest;
    Test<std::deque<int> > stdtest;

    aptest.StateFull(apdeque, 100);
    stdtest.StateFull(stddeque, 100);

    ap::deque<int>::iterator apdequeBegin = apdeque.begin();

```

```

ap::deque<int>::iterator apdequeEnd = (apdeque.end()-1);
for(; apdequeBegin < apdequeEnd; ++apdequeBegin,--apdequeEnd)
    if ((*apdequeEnd + *apdequeBegin) != 99)
        {
        std::cout << "Iterators did NOT work Perfectly value is "
            << (*apdequeEnd + *apdequeBegin) << std::endl;;
        }
std::cout << "Iterators worked Perfectly" << std::endl;;

if (apdeque.front() == stddeque.front())
    std::cout << "reference front works" << std::endl;
else
    std::cout << "reference front does NOT work" << std::endl;
if (apdeque.back() == stddeque.back())
    std::cout << "reference back works" << std::endl;
else
    std::cout << "reference back does NOT work" << std::endl;

if (apdeque.size() == stddeque.size())
    std::cout << "The size operation works perfectly" << std::endl;
else
    std::cout << "The size operation DID NOT work perfectly" << std::endl;

if (apdeque.max_size() == stddeque.max_size())
    std::cout << "The max_size operation works perfectly" << std::endl;
else
    std::cout << "The max_size operation DID NOT work perfectly" << std::endl;

for (int i = 0; i < 100; i++)
    if (apdeque[i] != stddeque[i])
        std::cout << "Bracket operator DID NOT work perfectly at " << i << std::endl;
std::cout << "Bracket operator worked perfectly." << std::endl;
return 0;
}
◇

```

## 5.6 Mutating Operation Tests

Testing mutating operations involves using STL's generic mutating algorithms with APdeque and SGIdeque. The following test will assure that a random sample of STL's generic algorithms on APdeque and SGIdeque will produce the different results. Also, some other test were performed on deque's member functions, namely swap and operator=.

"testMutators.C" ? ≡

```

#include "TestPackage.h"

int main()

```

```

{
    ap::deque<int> apdeque;
    std::deque<int> stddeque;
    Test<ap::deque<int> > aptest;
    Test<std::deque<int> > stdtest;

    aptest.StateFull(apdeque,50);
    stdtest.StateFull(stddeque,50);

    ap::deque<int> apdequeCopy = apdeque;
    if (std::equal(apdequeCopy.begin(),apdequeCopy.end(),apdeque.begin()))
        std::cout << "The Copy Assignment worked perfectly" << std::endl;
    else
        std::cout << "XXX The Copy Assignment DID NOT work perfectly XXX" << std::endl;

    ap::deque<int> apdequeReversed;
    ap::deque<int>::iterator apdequeIterator = apdeque.begin();
    std::reverse_copy(apdeque.begin(),apdeque.end(),insert_iterator<ap::deque<int> >
        (apdequeReversed,apdequeReversed.begin()));
    ap::deque<int>::reverse_iterator iap_reverseof_iterator = apdequeReversed.rbegin();

    std::cout << "This is apdeque: " << endl;
    std::copy(apdeque.begin(), apdeque.end(), std::ostream_iterator<int>(cout, " "));
    std::cout << std::endl;
    std::cout << "This is apdequeReversed: " << endl;
    std::copy(apdequeReversed.begin(), apdequeReversed.end(), std::ostream_iterator<int>
        (cout, " "));
    std::cout << std::endl;
    if (std::equal(apdequeReversed.begin(),apdequeReversed.end(),apdeque.rbegin()))
        std::cout << "The Reverse Copy worked perfectly before the swap." << std::endl;
    else
        std::cout << "XXX The Reverse Copy DID NOT work perfectly before the swap XXX"
            << std::endl;

    std::cout << "-----SWAP-----" << std::endl;
    apdeque.swap(apdequeReversed);

    if (std::equal(apdequeReversed.begin(),apdequeReversed.end(),apdeque.rbegin()))
        std::cout << "The Swap worked perfectly." << std::endl;
    else
        std::cout << "XXX The Swap DID NOT work perfectly XXX" << std::endl;
    return 0;
}
◇

```

## 5.7 Insert and Delete Operations Tests

The insert and delete operations deal with verifying the mutating member functions; such as, `insert`, `erase`, `operator=` as well as global comparison operators `operator==` and `operator<`. The trivial mutating operations like `push_back()`

and `pop_back()` have already been tested with the state members of the test package. The insertion operation tests concludes the validity tests of APdeque.

```
"testInsertions.C" ? ≡

#include "TestPackage.h"

int main()
{
    ap::deque<int> apdeque;
    std::deque<int> stddeque;
    Test<ap::deque<int> > aptest;
    Test<std::deque<int> > stdtest;

    aptest.StateFull(apdeque,50);
    stdtest.StateFull(stddeque,50);

    ap::deque<int> apdequeCopy = apdeque;

    apdeque.insert(apdeque.begin(), -1);
    stddeque.insert(stddeque.begin(), -1);
    if (std::equal(apdeque.begin(),apdeque.end(),stddeque.begin()))
        std::cout << "The Insert pos x worked perfectly." << std::endl;
    else
        std::cout << "XXX The Insert pos x DID NOT work perfectly XXX" << std::endl;
    apdeque.insert(apdeque.end(), 1000, -1);
    stddeque.insert(stddeque.end(), 1000, -1);
    if (std::equal(apdeque.begin(),apdeque.end(),stddeque.begin()))
        std::cout << "The Insert pos n x worked perfectly." << std::endl;
    else
        std::cout << "XXX The Insert pos n x DID NOT work perfectly XXX" << std::endl;

    apdeque.erase(apdeque.begin());
    stddeque.erase(stddeque.begin());
    if (std::equal(apdeque.begin(),apdeque.end(),stddeque.begin()))
        std::cout << "The Erase pos at the beginning worked perfectly." << std::endl;
    else
        std::cout << "XXX The Erase pos at the beginning DID NOT work perfectly XXX"
            << std::endl;

    apdeque.erase(apdeque.end()-1);
    stddeque.erase(stddeque.end()-1);
    if (std::equal(apdeque.begin(),apdeque.end(),stddeque.begin()))
        std::cout << "The Erase pos at the end worked perfectly." << std::endl;
    else
        std::cout << "XXX The Erase pos at the end DID NOT work perfectly XXX"
            << std::endl;

    apdeque.erase(ap::deque<int>::iterator(apdeque.begin()+apdeque.size()/2));
    stddeque.erase(std::deque<int>::iterator(stddeque.begin()+stddeque.size()/2));
}
```

```

if (std::equal(apdeque.begin(), apdeque.end(), stddeque.begin()))
    std::cout << "The Erase pos in the middle worked perfectly." << std::endl;
else
    std::cout << "XXX The Erase pos in the middle DID NOT work perfectly XXX"
        << std::endl;

apdeque.erase(ap::deque<int>::iterator(apdeque.begin()+apdeque.size()/2),
    apdeque.end());
stddeque.erase(std::deque<int>::iterator(stddeque.begin()+stddeque.size()/2),
    stddeque.end());
if (std::equal(apdeque.begin(), apdeque.end(), stddeque.begin()))
    std::cout << "The Erase first last from the middle to the end worked perfectly."
        << std::endl;
else
    std::cout << "XXX The Erase first last from the middle to the end DID NOT work
        perfectly XXX" << std::endl;

apdeque.erase(apdeque.begin(), ap::deque<int>::iterator
    (apdeque.begin()+apdeque.size()/2));
stddeque.erase(stddeque.begin(), std::deque<int>::iterator
    (stddeque.begin()+stddeque.size()/2));
if (std::equal(apdeque.begin(), apdeque.end(), stddeque.begin()))
    std::cout << "The Erase first last from begin to the middle worked perfectly."
        << std::endl;
else
    std::cout << "XXX The Erase first last from begin middle DID NOT work
        perfectly XXX" << std::endl;

apdeque.clear();
if (apdeque.empty())
    std::cout << "Clear worked perfectly." << std::endl;
else
    std::cout << "XXX Clear did NOT work perfectly XXX" << std::endl;

aptest.StateFull(apdeque, 1);
stdtest.StateFull(stddeque, 1);

for(int i = 0; i < 10; i++)
    {
        apdeque.resize(apdeque.size()*i, i);
        stddeque.resize(stddeque.size()*i, i);
        if (std::equal(apdeque.begin(), apdeque.end(), stddeque.begin()))
            std::cout << "The Resize worked perfectly." << std::endl;
        else
            std::cout << "XXX The Resize DID NOT work perfectly XXX" << std::endl;
    }

ap::deque<int> apdequeSecond(apdeque);
if (apdeque == apdequeSecond)
    std::cout << "Operator== Passed" << std::endl;

```

```

else
    std::cout << "XXX Operator== Failed XXX" << std::endl;

    return 0;
}
◇

```

## 5.8 Performance Testing

### 5.8.1 Introduction

The underlying implementation of APdeque is drastically different than SGI's. Many tradeoffs had to be considered in the implementation of certain parts of APdeque. The main tradeoff was how to deal with invalidated iterators when a mutating operation such as `resize` or `insert` occurred. SGI does not even take this into consideration, and provides no guarantees on keeping iterators valid. APdeque's iterators use an offset value to determine where an iterator is pointing to in the container. This alleviates iterator destruction when the underlying container is destroyed to create a new one, but the tradeoff is performance in certain operations. SGId deque's iterators always hold a pointer to the element in the container, whereas APdeque uses an offset value, which provides for fast increment, decrement, addition and subtraction, but degrades while dereferencing. The following performance tests will show the results of these tradeoffs in comparison with SGId deque and SGIvector (where applicable).

### 5.8.2 How Performance was Tested

Again, the test package was used to evaluate the performance of APdeque against SGId deque. The High Resolution Timer (See Appendix A.2) created by Chris McEvoy was used to calculate the time. All performance tests were done on Intel machines running Linux. Using this timer, only a few helper member functions in the test package were needed for computing the average of the multiple test runs. All results are given in clock ticks, where  $1 * 10^6$  clock ticks are the equivalent of one second.

⟨ComputeAvg ?⟩ ≡

```

long int ComputeAvg(Container &C)
{
    long int total = std::accumulate(C.begin(),C.end(),0);
    return (total/C.size());
}
◇

```

Macro referenced in ?.

To start the timer, just request it with the `Start()` member function of the test package.

⟨Start ?⟩ ≡

```
void Start()
{
    timing.Start();
}
```

◇

Macro referenced in ?.

To end the timer, just request it with the `End()` member function of the test package, and the resulting time (clock ticks) will be returned.

⟨End ?⟩ ≡

```
long int End()
{
    return timing.ElapsedTime();
}
```

◇

Macro referenced in ?.

All performance tests were performed in the same manner. Each performance run consisted of nested loops. The outer loop varies the container size and the inner loop repeats the performance test to compute the average over several runs. This sampling provides more accurate performance analysis.

### 5.8.3 Adaptor Performance

Two adaptors, stack and priority queue, were used to test the performance of `APdeque` against `SGIdeque` and `SGIvector`.

"AdaptorPerformance.C" ? ≡

```
#include "ap.h"
#include<iterator>
#include <queue>
#include "TestPackage.h"

#define START 1024
#define RUNS 3
#define TRIES 1
#define DATASIZE 1

int main()
{
    Test<ap::deque<int> > apTest("AP");
    Test<std::deque<int> > stdTest("STD");
    Test<std::vector<int> > vecTest("VEC");
    ap::deque<int> apPerformance;
    std::deque<int> stdPerformance;
```

```

std::vector<int> vecPerformance;

ap::deque<int> apTimes;
std::deque<int> stdTimes;
std::vector<int> vecTimes;

std::stack<vector<int> ,std::deque<vector<int> > > std_stack;
std::stack<vector<int> ,ap::deque<vector<int> > > ap_stack;
std::stack<vector<int> ,std::vector<vector<int> > > vec_stack;

int size = START;
std::vector<int> data;
vecTest.StateFull(data,DATASIZE);

for (int runs = 0; runs < RUNS; runs++)
{
    for (int tries = 0; tries < TRIES; tries++)
    {
        stdTest.Start();
        for (int i = 0; i < size; i++)
        {
            std_stack.push(data);
        }
        for (int i = 0; i < size; i++)
        {
            std_stack.top();
            std_stack.pop();
        }
        stdTimes.push_back(stdTest.End());

        apTest.Start();
        for (int i = 0; i < size; i++)
        {
            ap_stack.push(data);
        }
        for (int i = 0; i < size; i++)
        {
            ap_stack.top();
            ap_stack.pop();
        }
        apTimes.push_back(apTest.End());

        vecTest.Start();
        for (int i = 0; i < size; i++)
        {
            vec_stack.push(data);
        }
        for (int i = 0; i < size; i++)
        {

```

```

        vec_stack.top();
        vec_stack.pop();
    }
    vecTimes.push_back(vecTest.End());
}

apPerformance.push_back(apTest.ComputeAvg(apTimes));
stdPerformance.push_back(stdTest.ComputeAvg(stdTimes));
vecPerformance.push_back(vecTest.ComputeAvg(vecTimes));

size *= 2;
}

size = START;
std::cout << "Stack Performance" << std::endl;
std::cout << "Size\tAP\tSTD\tVEC\t" << std::endl;
for(int Q = 0; Q < apPerformance.size(); Q++)
{
    std::cout << size << "\t"
    << apPerformance[Q] << "\t"
    << stdPerformance[Q] << "\t"
    << vecPerformance[Q] << std::endl;
    size *= 2;
}

apPerformance.clear();
stdPerformance.clear();
vecPerformance.clear();

std::priority_queue<vector<int> >, std::deque<vector<int> > > std_priority_queue;
std::priority_queue<vector<int> >, ap::deque<vector<int> > > ap_priority_queue;
std::priority_queue<vector<int> >, std::vector<vector<int> > > vec_priority_queue;

size = START;
vecTest.StateFull(data,DATASIZE);

for (int runs = 0; runs < RUNS; runs++)
{
    for (int tries = 0; tries < TRIES; tries++)
    {
        stdTest.Start();
        for (int i = 0; i < size; i++)
        {
            std_priority_queue.push(data);
        }
        for (int i = 0; i < size; i++)
        {
            std_priority_queue.top();
            std_priority_queue.pop();
        }
    }
}

```

```

    }
    stdTimes.push_back(stdTest.End());

    apTest.Start();
    for (int i = 0; i < size; i++)
    {
        ap_priority_queue.push(data);
    }
    for (int i = 0; i < size; i++)
    {
        ap_priority_queue.top();
        ap_priority_queue.pop();
    }
    apTimes.push_back(apTest.End());

    vecTest.Start();
    for (int i = 0; i < size; i++)
    {
        vec_priority_queue.push(data);
    }
    for (int i = 0; i < size; i++)
    {
        vec_priority_queue.top();
        vec_priority_queue.pop();
    }
    vecTimes.push_back(vecTest.End());
}

apPerformance.push_back(apTest.ComputeAvg(apTimes));
stdPerformance.push_back(stdTest.ComputeAvg(stdTimes));
vecPerformance.push_back(vecTest.ComputeAvg(vecTimes));

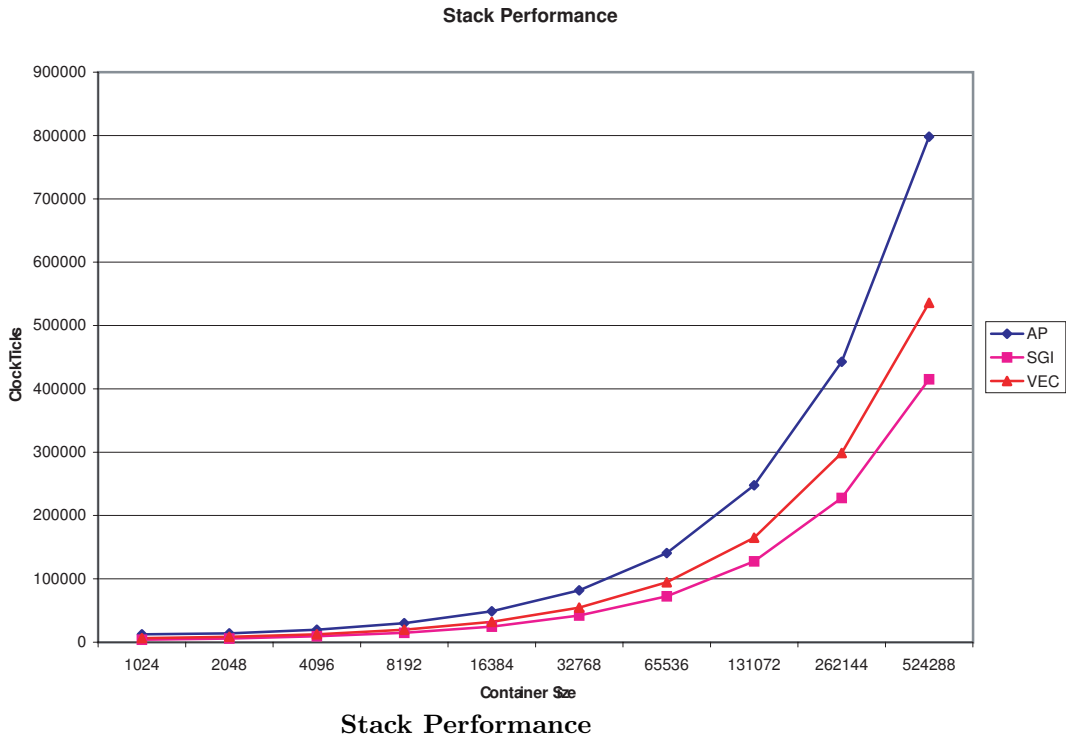
size *= 2;
}

size = START;
std::cout << "Priority Queue Performance" << std::endl;
std::cout << "Size\tAP\tSTD\tVEC\t" << std::endl;
for(int Q = 0; Q < apPerformance.size(); Q++)
{
    std::cout << size << "\t"
    << apPerformance[Q] << "\t"
    << stdPerformance[Q] << "\t"
    << vecPerformance[Q] << std::endl;
    size *= 2;
}
return 0;
}
◇

```

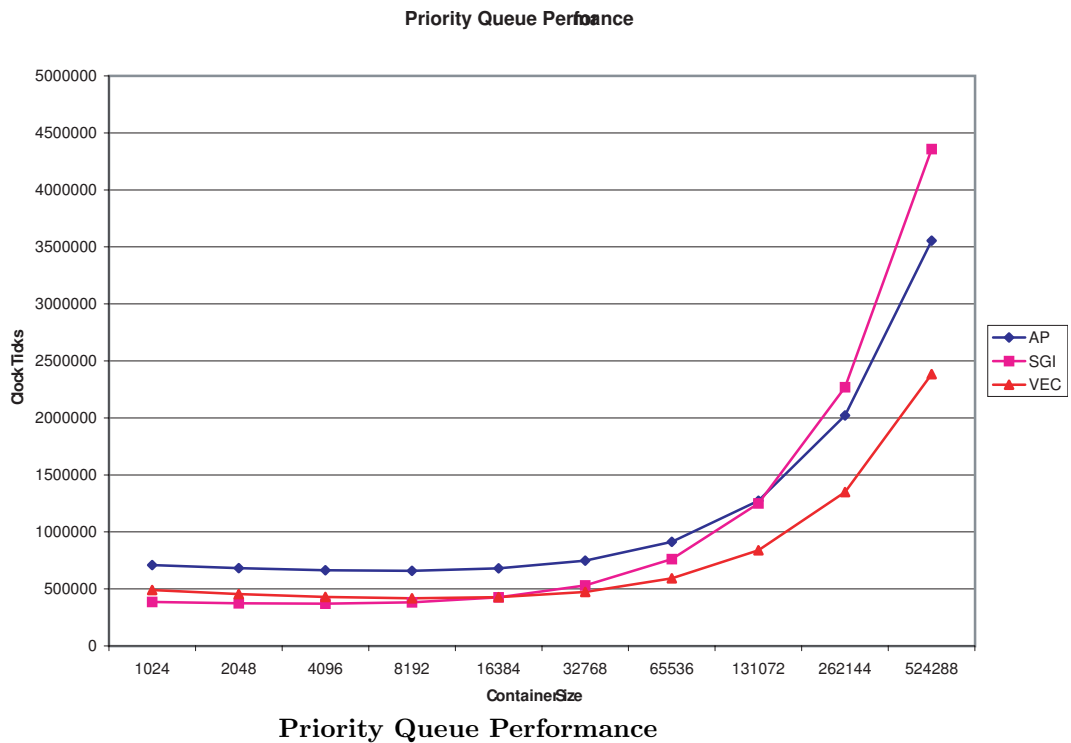
### 5.8.4 Stack Performance

The tests involving stack showed that SGIvector slightly out performed APdeque, and SGIdeque slightly performed SGIvector. This is expected based on the testing results of `push_back()`, because the stack test primarily consists of pushes, which are essentially `push_backs`. This also demonstrates that APdeque performs consistently with the added indirection of the stack adapter.



### 5.8.5 Priority Queue Performance

The tests involving the priority queue showed that APdeque out performed SGIdeque, and SGIvector slightly out performed APdeque. This can be attributed to the underlying implementation of vector.



### 5.8.6 Algorithm Performance

The algorithm tests involved using two algorithms. The tests used were `sort()` and `random_shuffle()` both part of STL. These two tests show how the APdeque performed against SGId deque and SGIvector, which is after all the underlying container for APdeque.

"AlgorithmPerformance.C" ? ≡

```
#include "ap.h"
#include<iterator>
#include <queue>
#include "TestPackage.h"

#define START 1024
#define RUNS 3
#define TRIES 1
#define DATASIZE 1

int main()
{
    Test<ap::deque<int> > apTest("AP");
```

```

Test<std::deque<int> > stdTest("STD");
Test<std::vector<int> > vecTest("VEC");
ap::deque<int> apPerformanceShuffle;
std::deque<int> stdPerformanceShuffle;
std::vector<int> vecPerformanceShuffle;

ap::deque<int> apPerformanceSort;
std::deque<int> stdPerformanceSort;
std::vector<int> vecPerformanceSort;

ap::deque<int> apTimesShuffle;
std::deque<int> stdTimesShuffle;
std::vector<int> vecTimesShuffle;

ap::deque<int> apTimesSort;
std::deque<int> stdTimesSort;
std::vector<int> vecTimesSort;

std::deque<int> stddeque;
ap::deque<int> apdeque;
std::vector<int> vec;

apTest.StateFull(apdeque,START);
stdTest.StateFull(stddeque,START);
vecTest.StateFull(vec,START);

int size = START;

size = START;
for (int runs = 0; runs < RUNS; runs++)
{
    for (int tries = 0; tries < TRIES; tries++)
    {
        stdTest.Start();
        std::random_shuffle(stddeque.begin(),stddeque.end());
        stdTimesShuffle.push_back(stdTest.End());

        stdTest.Start();
        std::sort(stddeque.begin(),stddeque.end());
        stdTimesSort.push_back(stdTest.End());

        apTest.Start();
        std::random_shuffle(apdeque.begin(),apdeque.end());
        apTimesShuffle.push_back(apTest.End());

        apTest.Start();
        std::sort(apdeque.begin(),apdeque.end());
        apTimesSort.push_back(apTest.End());

        vecTest.Start();

```

```

        std::random_shuffle(vec.begin(),vec.end());
        vecTimesShuffle.push_back(vecTest.End());

        vecTest.Start();
        std::sort(vec.begin(),vec.end());
        vecTimesSort.push_back(vecTest.End());
    }

    apPerformanceShuffle.push_back(apTest.ComputeAvg(apTimesShuffle));
    stdPerformanceShuffle.push_back(stdTest.ComputeAvg(stdTimesShuffle));
    vecPerformanceShuffle.push_back(vecTest.ComputeAvg(vecTimesShuffle));

    apPerformanceSort.push_back(apTest.ComputeAvg(apTimesSort));
    stdPerformanceSort.push_back(stdTest.ComputeAvg(stdTimesSort));
    vecPerformanceSort.push_back(vecTest.ComputeAvg(vecTimesSort));

    apTest.extend(apdeque,apdeque.size());
    stdTest.extend(stddeque,stddeque.size());
    vecTest.extend(vec,vec.size());

    size *= 2;
}

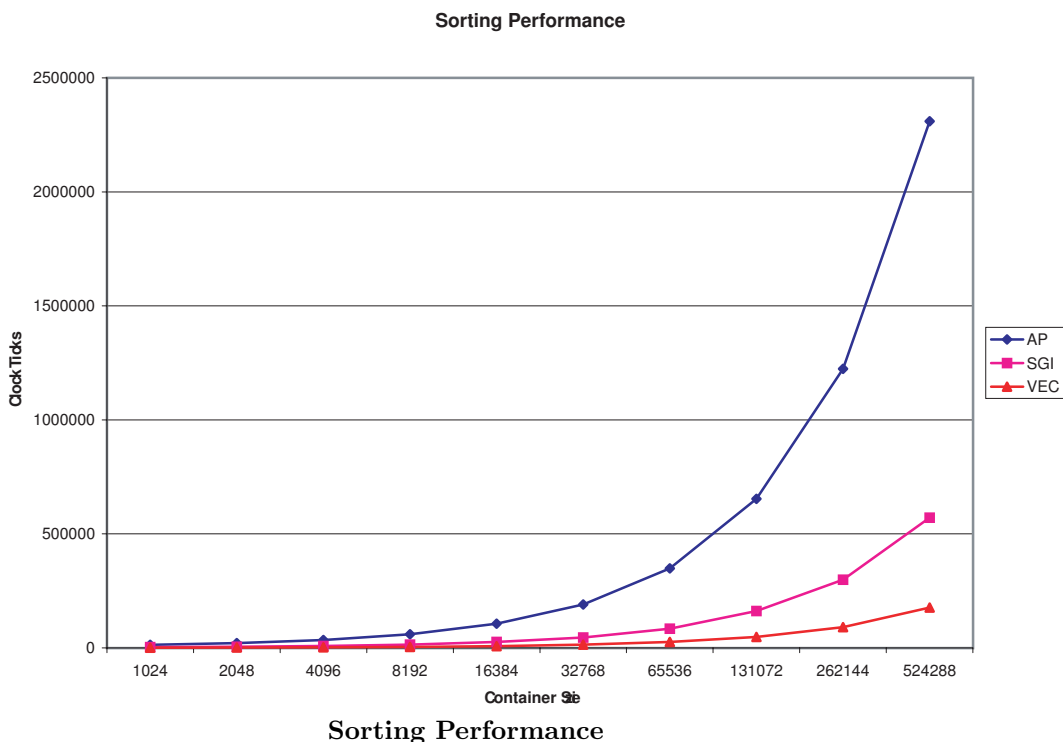
size = START;
std::cout << "Random Shuffle Performance" << std::endl;
std::cout << "Size\tAP\tSTD\tVEC\t" << std::endl;
for(int Q = 0; Q < apPerformanceShuffle.size(); Q++)
{
    std::cout << size << "\t"
    << apPerformanceShuffle[Q] << "\t"
    << stdPerformanceShuffle[Q] << "\t"
    << vecPerformanceShuffle[Q] << std::endl;
    size *= 2;
}

size = START;
std::cout << "Sorting Performance" << std::endl;
std::cout << "Size\tAP\tSTD\tVEC\t" << std::endl;
for(int Q = 0; Q < apPerformanceSort.size(); Q++)
{
    std::cout << size << "\t"
    << apPerformanceSort[Q] << "\t"
    << stdPerformanceSort[Q] << "\t"
    << vecPerformanceSort[Q] << std::endl;
    size *= 2;
}
return 0;
}
◇

```

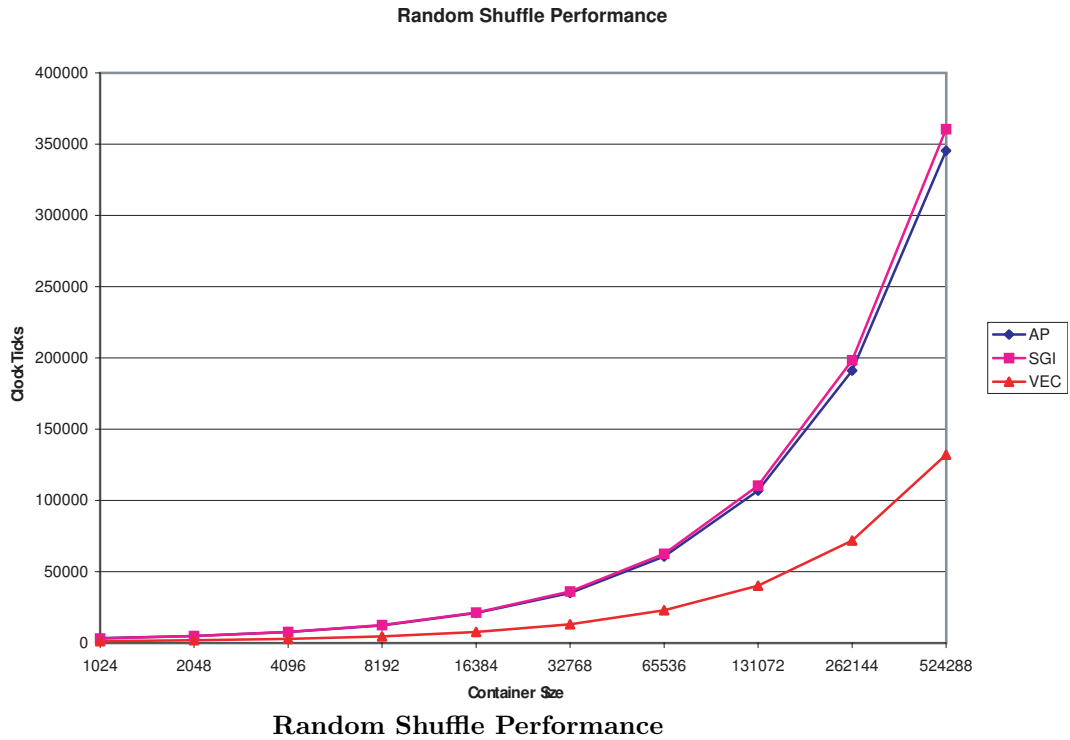
### 5.8.7 Sorting Performance

The sorting algorithm in STL, regardless of which version, uses the comparison operator often. This means that the performance of the containers relies heavily on the comparison operation, which in turn will rely on the dereferencing of iterators. APdeque degrades when dereferencing, which shows in the graph below. This degradation is due to the added feature of keeping iterators valid after operations like resize, which according to STL is not necessary. Future work will be concentrated on this performance test.



### 5.8.8 Random Shuffle Performance

As expected, due to the underlying implementation, the calculation performed when dereferencing an iterator is slow. However, the tradeoff lies in the quick retrieval of the iterator as seen in the graph below. APdeque achieved comparable results with SGId deque. Since this is a prototype of the concept being implemented there are potential optimizations that can be made, which would make APdeque more comparable to SGIvector, which is the ultimate goal.



### 5.8.9 Inserting Element Performance

There are a couple of tests to be performed when analyzing the performance of APdeque. Insertion of elements in each container provides a good comparison, and shows the combination of the degradation of APdeque's iterators and the high performance of it's insertion function. This analysis is the basis of future work, since this shows where the tradeoffs will be evident when changing the iterators and optimizing APdeque.

The performance analysis is separated into two categories: Insert operations, and Push/Pop operations. The inserting analysis code performs the same type of inserting whether the insertion is taking place in the beginning, middle, or the end. The pushing and popping is done in the same manner varying the amount of push/pops in each run.

"InsertionPerformance.C" ? ≡

```
#include "ap.h"
#include<iterator>
#include <queue>
#include "TestPackage.h"

#define START 1024
```

```

#define RUNS 3
#define TRIES 1
#define DATASIZE 1

int main()
{
    Test<ap::deque<int> > apTest("AP");
    Test<std::deque<int> > stdTest("STD");
    Test<std::vector<int> > vecTest("VEC");

    ap::deque<int> apPerformance;
    std::deque<int> stdPerformance;
    std::vector<int> vecPerformance;

    ap::deque<int> apTimes;
    std::deque<int> stdTimes;
    std::vector<int> vecTimes;

    int size = START;
    std::vector<int> data;
    vecTest.StateFull(data,DATASIZE);

    ap::deque<vector<int> > apdeque;
    std::deque<vector<int> > stddeque;
    std::vector<vector<int> > vec;

    for (int runs = 0; runs < RUNS; runs++)
    {
        for (int tries = 0; tries < TRIES; tries++)
        {
            apdeque.push_back(data);
            apTest.Start();
            for (int i = 0; i < size; i++)
            {
                apdeque.insert(apdeque.begin(),data);
            }
            apTimes.push_back(apTest.End());

            stddeque.push_back(data);
            stdTest.Start();
            for (int i = 0; i < size; i++)
            {
                stddeque.insert(stddeque.begin(),data);
            }
            stdTimes.push_back(apTest.End());

            vec.push_back(data);
            vecTest.Start();
            for (int i = 0; i < size; i++)

```

```

        {
            vec.insert(vec.begin(),data);
        }
vecTimes.push_back(vecTest.End());

apdequeue.clear();
stddequeue.clear();
vec.clear();

}

apPerformance.push_back(apTest.ComputeAvg(apTimes));
stdPerformance.push_back(stdTest.ComputeAvg(stdTimes));
vecPerformance.push_back(vecTest.ComputeAvg(vecTimes));

size *= 2;
}

size = START;
std::cout << "Insert Element in beginning Performance" << std::endl;
std::cout << "Size\tAP\tSTD\tVEC\t" << std::endl;
for(int Q = 0; Q < apPerformance.size(); Q++)
{
    std::cout << size << "\t"
    << apPerformance[Q] << "\t"
    << stdPerformance[Q] << "\t"
    << vecPerformance[Q] << std::endl;
    size *= 2;
}

apPerformance.clear();
stdPerformance.clear();
vecPerformance.clear();

apdequeue.clear();
stddequeue.clear();
vec.clear();

for (int runs = 0; runs < RUNS; runs++)
{
    for (int tries = 0; tries < TRIES; tries++)
    {
        apdequeue.push_back(data);
        apTest.Start();
        for (int i = 0; i < size; i++)
        {
            apdequeue.insert(apdequeue.begin()+(apdequeue.size()/2),data);
        }
        apTimes.push_back(apTest.End());
    }
}

```

```

stddeque.push_back(data);
stdTest.Start();
for (int i = 0; i < size; i++)
{
    stddeque.insert(stddeque.begin()+(stddeque.size()/2),data);
}
stdTimes.push_back(apTest.End());

vec.push_back(data);
vecTest.Start();
for (int i = 0; i < size; i++)
{
    vec.insert(vec.begin()+(vec.size()/2),data);
}
vecTimes.push_back(vecTest.End());

apdeque.clear();
stddeque.clear();
vec.clear();
}

apPerformance.push_back(apTest.ComputeAvg(apTimes));
stdPerformance.push_back(stdTest.ComputeAvg(stdTimes));
vecPerformance.push_back(vecTest.ComputeAvg(vecTimes));

size *= 2;
}

size = START;
std::cout << "Insert Element in middle Performance" << std::endl;
std::cout << "Size\tAP\tSTD\tVEC\t" << std::endl;
for(int Q = 0; Q < apPerformance.size(); Q++)
{
    std::cout << size << "\t"
    << apPerformance[Q] << "\t"
    << stdPerformance[Q] << "\t"
    << vecPerformance[Q] << std::endl;
    size *= 2;
}

apPerformance.clear();
stdPerformance.clear();
vecPerformance.clear();

apdeque.clear();
stddeque.clear();
vec.clear();

```

```

for (int runs = 0; runs < RUNS; runs++)
{
    for (int tries = 0; tries < TRIES; tries++)
    {
        apdeque.push_back(data);
        apdeque.push_back(data);
        apTest.Start();
        for (int i = 0; i < size; i++)
        {
            apdeque.insert((apdeque.begin()+i),data);
        }
        apTimes.push_back(apTest.End());

        stddeque.push_back(data);
        stddeque.push_back(data);
        stdTest.Start();
        for (int i = 0; i < size; i++)
        {
            stddeque.insert((stddeque.begin()+i),data);
        }
        stdTimes.push_back(apTest.End());

        vec.push_back(data);
        vec.push_back(data);
        vecTest.Start();
        for (int i = 0; i < size; i++)
        {
            vec.insert(vec.begin()+i,data);
        }
        vecTimes.push_back(vecTest.End());

        apdeque.clear();
        stddeque.clear();
        vec.clear();
    }

    apPerformance.push_back(apTest.ComputeAvg(apTimes));
    stdPerformance.push_back(stdTest.ComputeAvg(stdTimes));
    vecPerformance.push_back(vecTest.ComputeAvg(vecTimes));

    size *= 2;
}

size = START;
std::cout << "Insert Element in end Performance" << std::endl;
std::cout << "Size\tAP\tSTD\tVEC\t" << std::endl;
for(int Q = 0; Q < apPerformance.size(); Q++)
{
    std::cout << size << "\t"

```

```

        << apPerformance[Q] << "\t"
        << stdPerformance[Q] << "\t"
        << vecPerformance[Q] << std::endl;
        size *= 2;
    }

    return 0;
}
◇

```

### 5.8.10 Pushing and Popping Performance

The analysis for pushing and popping proves that the APdeque iterators perform reasonably well in comparison to SGIdeque when pushing in the front or the back. SGIdeque demonstrates superior performance with regards to pushing. The underlying container representation provides fast insertion when a reallocation occurs. This is because it rarely reallocates, instead it just adds more blocks to the array of pointers. However, if APdeque's iterator implementation were to change, then there is a possibility of achieving results closer to the SGIdeque implementation.

"PushPopPerformance.C" ? ≡

```

#include "ap.h"
#include<iterator>
#include <queue>
#include "TestPackage.h"

#define START 1024
#define RUNS 3
#define TRIES 1
#define DATASIZE 1

int main()
{

    Test<ap::deque<int> > apTest("AP");
    Test<std::deque<int> > stdTest("STD");
    Test<std::vector<int> > vecTest("VEC");

    ap::deque<int> apPerformancePush;
    std::deque<int> stdPerformancePush;
    std::vector<int> vecPerformancePush;

    ap::deque<int> apTimesPush;
    std::deque<int> stdTimesPush;
    std::vector<int> vecTimesPush;
}

```

```

ap::deque<int> apPerformancePop;
std::deque<int> stdPerformancePop;
std::vector<int> vecPerformancePop;

ap::deque<int> apTimesPop;
std::deque<int> stdTimesPop;
std::vector<int> vecTimesPop;

int size = START;
std::vector<int> data;
vecTest.StateFull(data,DATASIZE);

ap::deque<vector<int> > apdeque;
std::deque<vector<int> > stddeque;
std::vector<vector<int> > vec;

for (int runs = 0; runs < RUNS; runs++)
{
    for (int tries = 0; tries < TRIES; tries++)
    {
        apTest.Start();
        for (int i = 0; i < size; i++)
        {
            apdeque.push_back(data);
        }
        apTimesPush.push_back(apTest.End());

        apTest.Start();
        for (int i = 0; i < size-1; i++)
        {
            apdeque.pop_back();
        }
        apTimesPop.push_back(apTest.End());

        stdTest.Start();
        for (int i = 0; i < size; i++)
        {
            stddeque.push_back(data);
        }
        stdTimesPush.push_back(apTest.End());

        stdTest.Start();
        for (int i = 0; i < size-1; i++)
        {
            stddeque.pop_back();
        }
        stdTimesPop.push_back(apTest.End());
    }
}

```

```

vecTest.Start();
for (int i = 0; i < size; i++)
{
    vec.push_back(data);
}
vecTimesPush.push_back(vecTest.End());

vecTest.Start();
for (int i = 0; i < size-1; i++)
{
    vec.pop_back();
}
vecTimesPop.push_back(vecTest.End());

}

apPerformancePush.push_back(apTest.ComputeAvg(apTimesPush));
stdPerformancePush.push_back(stdTest.ComputeAvg(stdTimesPush));
vecPerformancePush.push_back(vecTest.ComputeAvg(vecTimesPush));

apPerformancePop.push_back(apTest.ComputeAvg(apTimesPop));
stdPerformancePop.push_back(stdTest.ComputeAvg(stdTimesPop));
vecPerformancePop.push_back(vecTest.ComputeAvg(vecTimesPop));

size *= 2;
}

size = START;
std::cout << "Pushing Back Element Performance" << std::endl;
std::cout << "Size\tAP\tSTD\tVEC\t" << std::endl;
for(int Q = 0; Q < apPerformancePush.size(); Q++)
{
    std::cout << size << "\t"
    << apPerformancePush[Q] << "\t"
    << stdPerformancePush[Q] << "\t"
    << vecPerformancePush[Q] << std::endl;
    size *= 2;
}

size = START;
std::cout << "Popping Back Element Performance" << std::endl;
std::cout << "Size\tAP\tSTD\tVEC\t" << std::endl;
for(int Q = 0; Q < apPerformancePop.size(); Q++)
{
    std::cout << size << "\t"
    << apPerformancePop[Q] << "\t"
    << stdPerformancePop[Q] << "\t"
    << vecPerformancePop[Q] << std::endl;
    size *= 2;
}

```

```

}

apPerformancePush.clear();
stdPerformancePush.clear();
vecPerformancePush.clear();

apPerformancePop.clear();
stdPerformancePop.clear();
vecPerformancePop.clear();

apdeque.clear();
stddeque.clear();
vec.clear();

for (int runs = 0; runs < RUNS; runs++)
{
    for (int tries = 0; tries < TRIES; tries++)
    {
        apTest.Start();
        for (int i = 0; i < size; i++)
        {
            apdeque.push_front(data);
        }
        apTimesPush.push_back(apTest.End());

        apTest.Start();
        for (int i = 0; i < size-1; i++)
        {
            apdeque.pop_front();
        }
        apTimesPop.push_back(apTest.End());

        stdTest.Start();
        for (int i = 0; i < size; i++)
        {
            stddeque.push_front(data);
        }
        stdTimesPush.push_back(apTest.End());

        stdTest.Start();
        for (int i = 0; i < size-1; i++)
        {
            stddeque.pop_front();
        }
        stdTimesPop.push_back(apTest.End());
    }
}

```

```

        apPerformancePush.push_back(apTest.ComputeAvg(apTimesPush));
        stdPerformancePush.push_back(stdTest.ComputeAvg(stdTimesPush));

        apPerformancePop.push_back(apTest.ComputeAvg(apTimesPop));
        stdPerformancePop.push_back(stdTest.ComputeAvg(stdTimesPop));

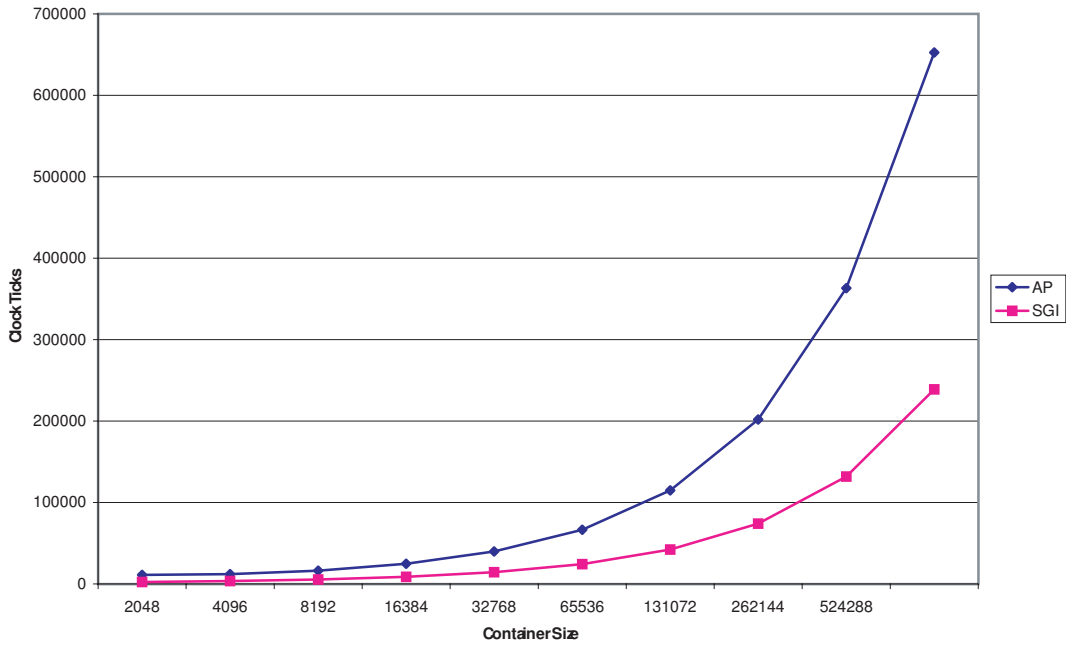
        size *= 2;
    }

    size = START;
    std::cout << "Pushing Front Element Performance" << std::endl;
    std::cout << "Size\tAP\tSTD\tVEC\t" << std::endl;
    for(int Q = 0; Q < apPerformancePush.size(); Q++)
    {
        std::cout << size << "\t"
        << apPerformancePush[Q] << "\t"
        << stdPerformancePush[Q] << std::endl;
        size *= 2;
    }

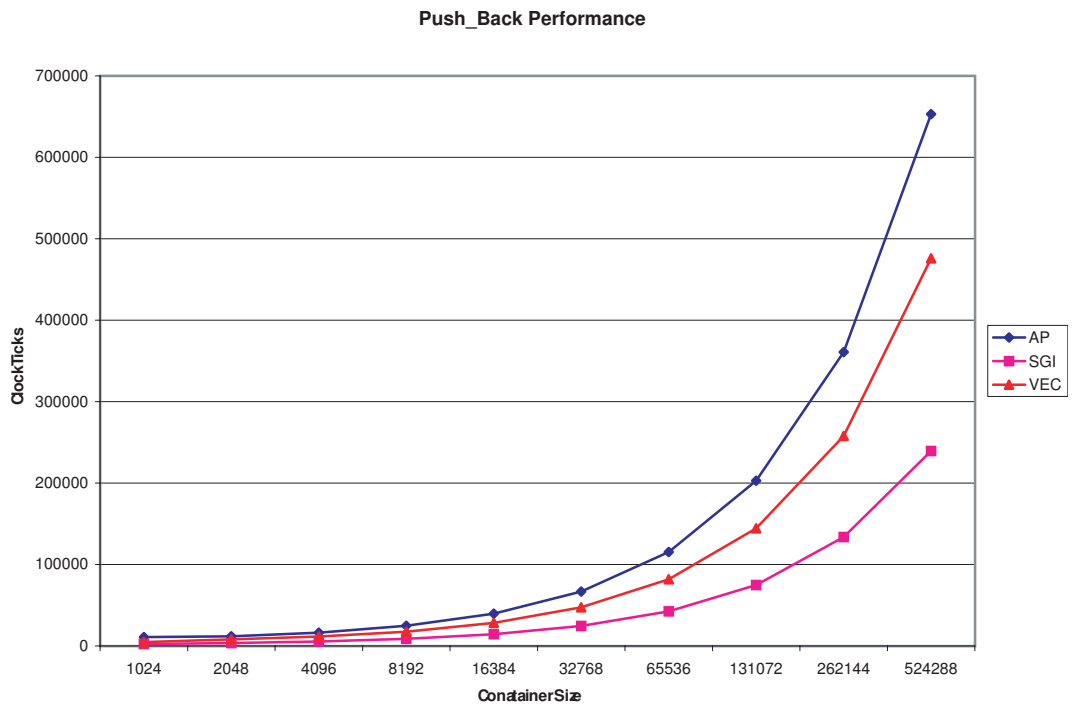
    size = START;
    std::cout << "Popping Front Element Performance" << std::endl;
    std::cout << "Size\tAP\tSTD\tVEC\t" << std::endl;
    for(int Q = 0; Q < apPerformancePop.size(); Q++)
    {
        std::cout << size << "\t"
        << apPerformancePop[Q] << "\t"
        << stdPerformancePop[Q] << std::endl;
        size *= 2;
    }
    return 0;
}
◇

```

Push\_front Performance



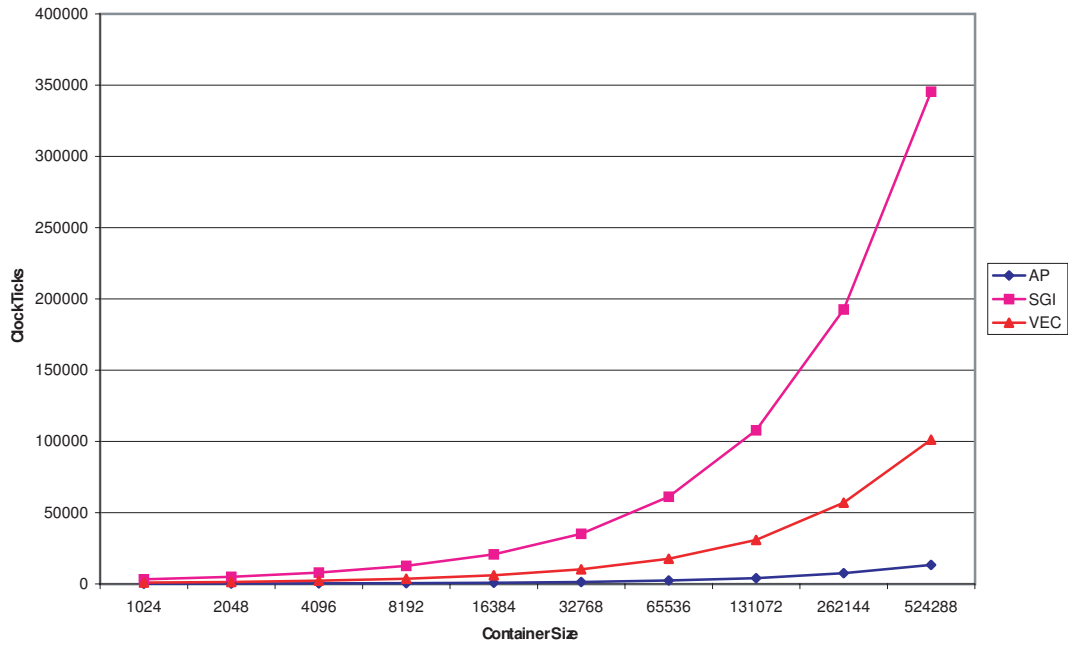
Push Front Performance



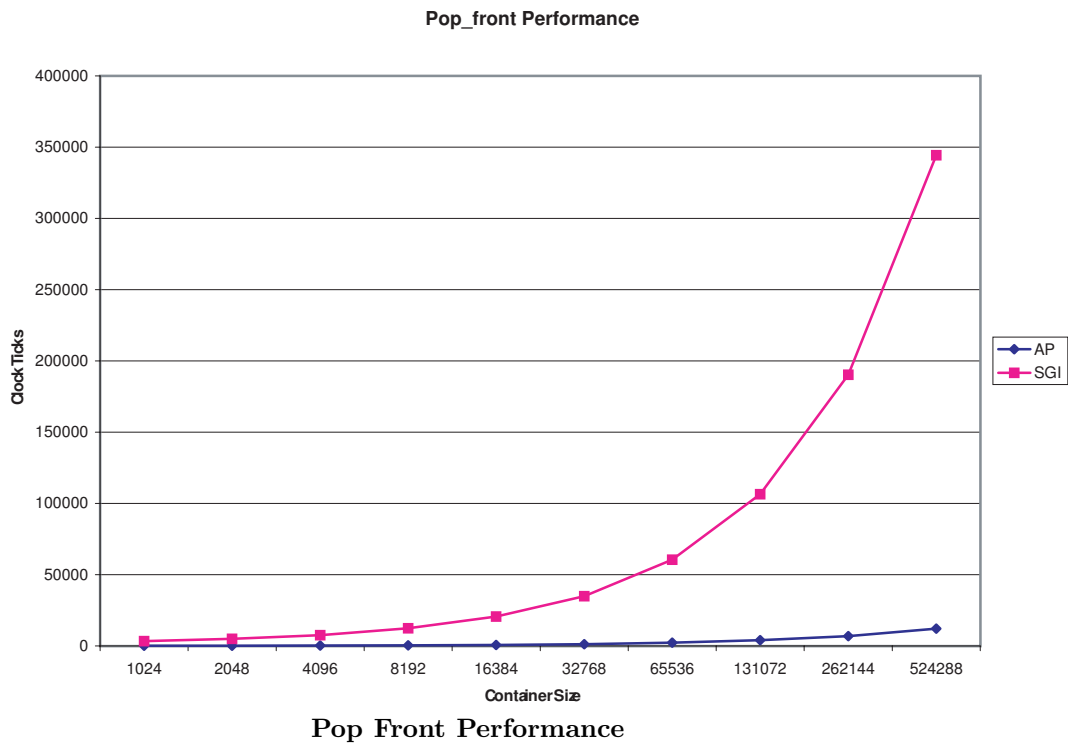
**Push Back Performance**

These charts prove how the circular buffer is advantageous, in some aspects, to other strategies when implementing a deque container. APdeque's utilization of the underlying vector iterators provide some streaming fast operations, which is evident when popping. This is because APdeque's pop functions only involve changing the `start/finish` values appropriately.

Pop\_back Performance



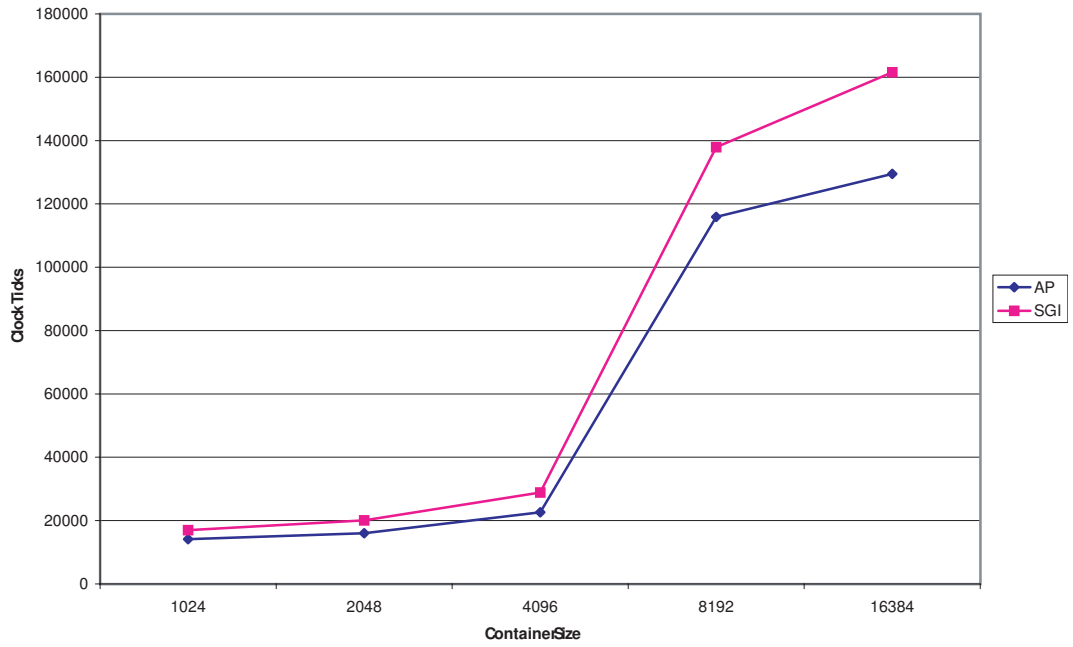
Pop Back Performance



### 5.8.11 Insertion Performance

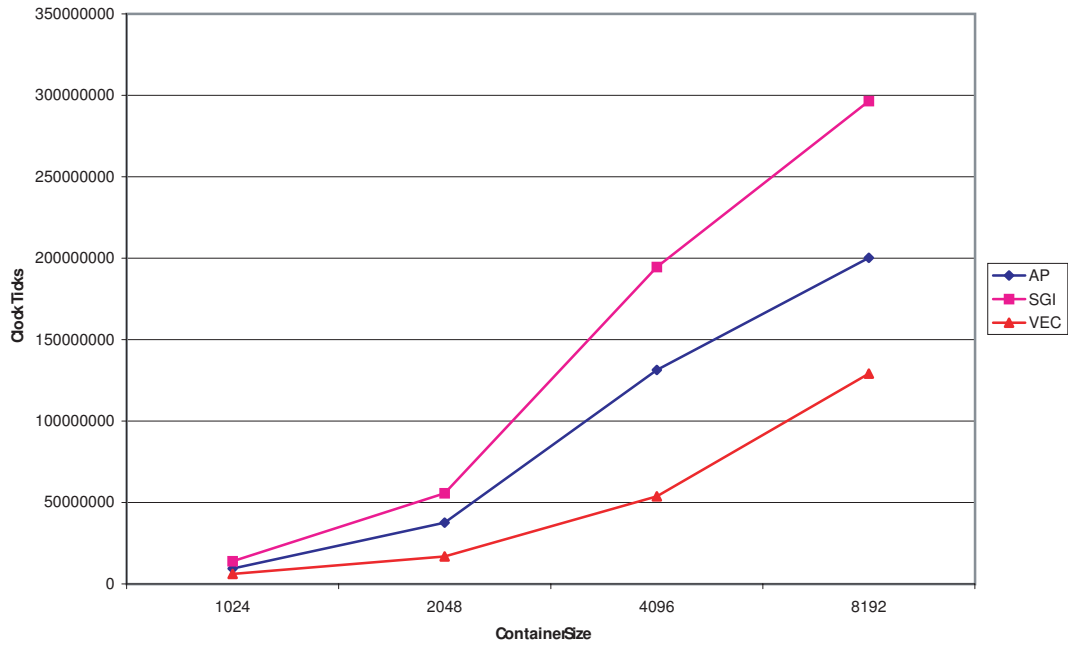
APdeque's insertion operation has been optimized to utilize the double ended queue concept. When inserting, APdeque shifts according to its position, this provides an incredible performance increase over both SGIdeque and even SGIvector.

Insert at Beginning Performance



Insert at Beginning Performance

Insert Middle Performance



Insert at Middle Performance

## 6 Future Work

The proposed APdeque was a proof of concept design, and by no means it is fully optimized; in fact, it would be possible to optimize most of APdeque's member functions. The idea behind the research was to ensure that APdeque could be applied, provide more guarantees on iterator validity, and perform generic algorithms and member functions that are efficiently comparable to SGIdeque.

The performance of APdeque's `iterator`, especially dereferencing elements, was recognized as one of the biggest drawbacks, in terms of speed, of the proposed implementation. The Primary goal of the future work would be to eliminate this potential deficiency of APdeque. Two radically different approaches are considered to address the problem:

- Do not provide guarantees on iterator validity, and make APdeque efficiently comparable to SGIVector
- Provide guarantees on iterator validity, and make APdeque efficiently comparable to SGIdeque

Those two approaches are going to be described in following sections.

### 6.1 Removing Iterator Validity Guarantees Approach

Removing guarantees of iterator validity allows to avoid additional overhead associated with this requirement. One of the possible implementations is where iterators use vector iterators internally to point to elements. This would make the dereference operator virtually as fast as the vector's iterator dereference. Pointer arithmetic operators (ex. `+` or `-`) would have to include additional code associated with the circular buffer. In summary, it is expected that this approach could improve performance dramatically.

### 6.2 Providing Iterator Validity Guarantees Approach

The goal of this approach is to improve the existing APdeque, without removing any requirements. It is the preferred approach since the new functionality may be useful in many situations. Two ways of improving iterator performance are considered. The first possible way is to improve existing code. For example, the APdeque's iterator uses the assignment operation which is time consuming. It also performs function calls to find out the internal vector's beginning and end. The function call overhead could be avoided by storing the values of these functions. Another possibility is to combine the offset approach with keeping an iterator to the internal vector. It could be implemented using a flag type scheme. Then the iterators would have both an offset member and a vector iterator member. The deque class could contain a flag, for example the flag could represent the version number of a deque. When the iterator is created it would copy the current deque's version number. As long as the version number would

agree with the one in the deque, the iterator could use the internal vector iterator to dereference an element. If the version would differ, then offset could be used to update an iterator (update the internal vector iterator to point to a valid element). On the other hand, an event driven scheme could be used. Iterators would use a vector iterator as internal pointer to an element. Deque could keep track of iterators to it's elements. When the underlying vector changes (reallocates), then all iterators would be updated and kept valid. This scheme may be much more complicated.

### **6.3 Summary**

All the approaches have their advantages and disadvantages. To decide which of them is more desirable, all could be implemented and tested. Then, performance analysis would have to be conducted to check which approach gives the best performance gain.

"Deque.h" ? ≡

- ⟨ deque compiler directives ? ⟩
- ⟨ deque\_iterator typedefs, data members, and constructors ? ⟩
- ⟨ deque\_iterator operator\* ? ⟩
- ⟨ deque\_iterator increment and decrement ? ⟩
- ⟨ deque\_iterator plus and minus ? ⟩
- ⟨ deque\_iterator operator[] ? ⟩
- ⟨ deque\_iterator comparison operators ? ⟩
- ⟨ deque typedefs and friends ? ⟩
- ⟨ deque default constructor ? ⟩
- ⟨ deque n element constructor ? ⟩
- ⟨ deque copy n constructor ? ⟩
- ⟨ deque copy constructor ? ⟩
- ⟨ deque copy range constructor ? ⟩
- ⟨ deque destructor ? ⟩
- ⟨ deque push back ? ⟩
- ⟨ deque push front ? ⟩
- ⟨ deque pop back ? ⟩
- ⟨ deque pop front ? ⟩
- ⟨ deque insert x before pos ? ⟩
- ⟨ deque insert n copies of x before pos ? ⟩
- ⟨ deque insert range before pos ? ⟩
- ⟨ deque resize ? ⟩
- ⟨ deque swap ? ⟩
- ⟨ deque operator= ? ⟩
- ⟨ deque operator[] ? ⟩
- ⟨ deque erase ? ⟩
- ⟨ deque erase range ? ⟩
- ⟨ deque clear ? ⟩
- ⟨ deque size functions ? ⟩
- ⟨ deque begin and end ? ⟩
- ⟨ deque front and back ? ⟩
- ⟨ deque empty ? ⟩
- ⟨ deque rbegin and rend ? ⟩
- ⟨ deque private data members ? ⟩
- ⟨ deque realloc ? ⟩
- ⟨ deque reinstantiate ? ⟩
- ⟨ deque reassign\_pointers ? ⟩
- ⟨ deque copy\_dispatch1 ? ⟩
- ⟨ deque copy\_dispatch2 ? ⟩
- ⟨ deque copy\_n ? ⟩
- ⟨ deque global members ? ⟩

◇

## A Appendix A

### A.1 ap.h

```
"ap.h" ? ≡  
  
#ifndef AP_H_PROJECT_TEAM_UGRAD  
#define AP_H_PROJECT_TEAM_UGRAD  
  
#include <vector>  
#include <string>  
#include <deque>  
#include <stack>  
#include <queue>  
#include <algorithm>  
#include <numeric>  
#include <iostream>  
#include "Deque.h"  
  
#ifndef __TYPE_TRAITS_H  
#include <type_traits.h>  
#endif  
  
#define __LINUX__  
#include "hirestimer.h"  
#endif  
◇
```

### A.2 hirestimer.h

```
"hirestimer.h" ? ≡  
  
class HighResTimer {  
protected:  
    unsigned int startTime;  
    char id[30];  
  
public:  
    HighResTimer(const char* id);  
    ~HighResTimer() {}  
  
    void Start();  
    unsigned int ElapsedTime();  
    virtual unsigned int GetTime() = 0;  
    virtual unsigned int TicksPerSecond() = 0;  
};  
  
HighResTimer::HighResTimer(const char *id)  
{
```

```

        startTime = 0;
        strncpy(this->id, id, 30);
        this->id[29] = 0;
    }

    void HighResTimer::Start()
    {
        startTime = GetTime();
    }

    unsigned int HighResTimer::ElapsedTime()
    {
        return GetTime() - startTime;
    }

#ifdef __WINDOWS__

#include <windows.h>
#include <mmsystem.h>

class WindowsTimer : public HighResTimer {
public:
    WindowsTimer(const char* id) : HighResTimer(id) {}
    ~WindowsTimer() {}

    unsigned int GetTime();
    unsigned int TicksPerSecond();
};

unsigned int WindowsTimer::GetTime()
{
    LARGE_INTEGER curtime;
    QueryPerformanceCounter(&curtime);
    return curtime.LowPart;
}

unsigned int WindowsTimer::TicksPerSecond()
{
    return 1193000;
}

#endif __WINDOWS__

#ifdef __LINUX__

#include <sys/time.h>
#include <sys/times.h>
#include <sys/types.h>
#include <unistd.h>

```

```

class LinuxTimer : public HighResTimer {
public:
    LinuxTimer(const char* id) : HighResTimer(id) {}
    ~LinuxTimer() {}

    unsigned int GetTime();
    unsigned int TicksPerSecond();
};

unsigned int LinuxTimer::GetTime()
{
    struct timeval curtime;
    gettimeofday(&curtime, NULL);
    return (curtime.tv_sec) * 1000000 + curtime.tv_usec;
}

unsigned int LinuxTimer::TicksPerSecond()
{
    return 1000000;
}

#endif __LINUX__

#ifdef __BEOS__

#include <OS.h>

class BeOSTimer : public HighResTimer {
public:
    BeOSTimer(const char* id) : HighResTimer(id) {}
    ~BeOSTimer() {}

    unsigned int GetTime();
    unsigned int TicksPerSecond();
};

unsigned int BeOSTimer::GetTime()
{
    return system_time();
}

unsigned int HighResTimer::TicksPerSecond()
{
    return 1000000;
}

#endif __BEOS__

```

```

#ifdef __MACOS__

#include <QuickTimeComponents.h>

class MacTimer : public HighResTimer {
public:
    static ComponentInstance clockComponent;
    static int instance;

public:
    MacTimer(const char* id);
    ~MacTimer();

    unsigned int GetTime();
    unsigned int TicksPerSecond();
};

int MacTimer::instance = 0;

MacTimer::MacTimer(const char* id) : HighResTimer(id)
{
    if (instance == 0) {
        clockComponent=
            OpenDefaultComponent('clck', 'micr');
        if (clockComponent) assert(0);
    }

    ++instance;
}

MacTimer::~MacTimer()
{
    --instance;
    if (instance == 0) CloseComponent(clockComponent);
}

unsigned int MacTimer::GetTime()
{
    TimeRecord timeRec;
    ClockGetTime(clockComponent, &timeRec);
    return timeRec.value.lo;
}

unsigned int MacTimer::TicksPerSecond()
{
    return 1000000;
}

#endif __MACOS__

```

◇

## B Appendix B

### B.1 TestPackage.h

Header file for Testing Package

```
"TestPackage.h" ? ≡

#ifdef __INCLUDE_TPACKAGE_H
#define __INCLUDE_TPACKAGE_H

#include "ap.h"

template<typename Container>
class Test
{
private:
    LinuxTimer timing;

public:
    Test(char *s="Default") : timing(s) { }
    <StateFull ?>
    <StateBegin ?>
    <StateMiddle ?>
    <StateWrapped ?>
    <StateEmpty ?>
    void extend(Container &C,int size)
    {
        for (int i = 0; i < size; i++)
        {
            C.push_back(size+i);
        }
        return;
    }
    <ComputeAvg ?>
    <Start ?>
    <End ?>
};

#else
#endif
◇
```

## C Bibliography

### References

- [Briggs] P. Briggs, *Nuweb, a simple literate programming tool*, Version 0.87, 1989.
- [Knuth84] D.E. Knuth, Literate programming. *Computer Journal* 27 (1984), 97–111.
- [Mu96] D.R. Musser. *Measuring Computing Times and Operation Counts*, <http://www.cs.rpi.edu/musser/gp/timing.html>.
- [MS96] D.R. Musser, A. Saini. *STL Tutorial and Reference Guide: Programming with Standard Template Library*. Addison-Wesley, Reading, MA, 1996.
- [AUSTERN] Mathew H. Austern *Generic Programming and the STL*. Addison-Wesley, Reading, MA, 1998.
- [SGI96] Silicon Graphics Standard Template Library Programming Guide, online guide, <http://www.sgi.com/Technology/STL/>.