Towards stronger guarantees: Safer iterators

Bo Simonsen
Department of Computing, University of Copenhagen,
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
bosim@diku.dk

Abstract. Iterators are a fundamental part of the STL. They are the link between generic algorithms and containers such that it is possible for the generic algorithms to traverse and access the elements in the container in a homogeneous way. The iterator validity property ensures that iterators are kept valid at any time except when the element is deleted. Modifications of the container should not affect the already existing iterators. This property is a part of the stronger guarantees criteria for the CPH STL, which means any container should provide this property, which is not trivial to obtain for containers which are based on arrays. In this paper we will provide an implementation which provides iterator validity for a dynamic array, and we will study data structures for obtaining snapshot iterators and persistence.

1. Introduction

In this paper we consider the problem of implementing robust containers. The guarantees satisfied by these containers are not specified in the C++ standard [16], but they are CPH STL [10] specific requirements. The guarantees are enhancements of safety, usability, and reliability from the library user's point of view and every `safe_` realization must provide these guarantees. The *stronger guarantees* [19, 4] are defined to be:

Time optimality: All container operations should be as efficient (or faster) as specified in the C++ standard where the complexity of each operation is defined.

Space efficiency: The amount of storage space used should be linear on the number of elements stored in the container (i.e. the amount of extra space is $\mathcal{O}(n)$ words where n denotes the number of elements currently stored in the data structure).

Strong exception safety: If a container operation fails (i.e. an exception is thrown), the container must be in the same state as before the operation which caused the exception.

Iterator validity: Iterators are kept valid at any time such that iterators given out are not affected by container operations (e.g. insert, erase) except when an element is deleted.

CPH STL Report 2009-1, November 2008, Revised February 2009.

In this paper we focus on the last guarantee. We are mainly concerned in developing variants of dynamic arrays which provide robust traversal mechanisms over the stored elements.

The paper is organized according to the key aspects of iterator safety: problem definition (Section 2), storage organization (Section 3), snapshot iterators (Section 4), and partially persistent search trees (Section 5). Finally, some benchmark results are reported (Section 6) and concluding remarks offered (Section 7). The code, which was written to carry out this study, can be found in the appendix (Table of contents on p. 27).

2. Iterator validity

We now look deeper into problems related to iterator validity. Many containers in the CPH STL use nodes for storing data and internal structure (i.e. pointers to other elements in the container). An example is a binary tree which uses a node for storing data, pointers to the child nodes, and a pointer to the parent node. Usually next- and previous nodes are also stored in the node, in order to satisfy the criteria of performing iterator operations in constant time [24, 18]. When an insertion occurs, the tree is searched until the place where the element should be inserted is found and then the element is inserted by allocating space and adjusting the pointers which also includes rebalancing operations. This does not affect the already existing iterators since a node exists at the same address in memory until it is deallocated, rebalancing and insertion only adjust pointers in the nodes, which means these operations are not changing the address of the node. So far, we can conclude there is not any problems related to iterator validity with this type of containers.

The STL provides other kinds of containers: these are `deque` and `vector` which are based on arrays such that it is possible to access every element in constant time. We will now look deeper into the implementation of the `vector` container. This container provides a dynamic array, which means that the array will be expanded when it is full; the operation which expands the array is completely transparent to the user. Here iterator validity turns out to be a problem. The first observation is: when the array is expanded, new memory of a larger size will be allocated and the old array will be deallocated, and the elements will be moved to the new array. If the iterator is designed such that it holds a regular pointer to the element, the iterator gets invalidated such that it cannot be used (if one tries to use it a runtime error can occur). To solve this problem we can modify the iterator such that it contains a tuple of a pointer to the container and an index. Now reallocation of the array is not a problem since the pointer to the container is stable and a method (usually `operator[]`) in the container can be used to access the requested element. But there are still problems of keeping the iterators valid. Let us consider the following scenario: we have an array with two elements A and C, and we have iterators to both elements. Now,

we insert B between A and C. The iterator which pointed to A is still valid, i.e. it still points to A, but the iterator which pointed to C in the first place does now point to B, which means that this iterator is not valid any more. This happens because the iterator uses indexes to locate the associated elements, but the indexes are not static since when a new element is inserted the existing elements will be moved. This is exactly the problem we need to solve in this paper. The problem is briefly mentioned in the literature [9, Section 6.5], we will use that approach to solve it by introducing the so-called *application object*, which we will denote as an entry.

3. Entries

Our proposal to solve the problem described in the previous section is to introduce the entry. What we desire is a similar storage policy for arrays as we have for tree containers where the data is located inside a node which resides at a stable memory address. We define a *storage policy* as a policy [17, 2] which states how data should be stored. This means that we allow the user to design his own storage policies and our container uses these. The most common application of policies in generic programming is the allocator (which contains methods for allocating memory) and the comparator (which is used to compare the elements in the container).

We define the *entry* to be the storage policy for arrays with the same properties as nodes. This change in behavior means that the array will now contain pointers to entries, and entries will contain pointers to their respective position in the array (i.e. a bidirectional relationship). Iterators will now point at entries, and when the pointer to an entry is moved in the array, the pointer from the entry to the array will be corrected correspondingly. This solution keeps iterators valid at any time. The entry mechanism is implemented by modifying `dynamic_array` which implements a dynamic array; the new implementation is called `iv_dynamic_array` (iv is short for iterator valid), and we will look at some implementation related details later. An overview of this solution is shown in Figure 1. The entry is an application of the proxy design pattern [14]. In context of design patterns the entry is a proxy between the container (i.e. the array) and iterators. Since the value is stored in the entry, the entry does not act as proxy for the iterators when dereferencing the iterator, but when an advance operation takes place, the entry acts as a proxy for the iterator to access the array.

Already now we can see this solution yields some obvious advantages including: performance and iterator simplicity. With respect to performance the old implementation of the dynamic array was constructing and destructing the data elements every time they were moved. Constructing and destructing elements can be a very expensive operation, since the constructors and destructors can contain complex code (e.g constructing a list with an arbitrary number of elements). With the new solution we only move pointers which is an atomic operation. Another advantage is simplicity with

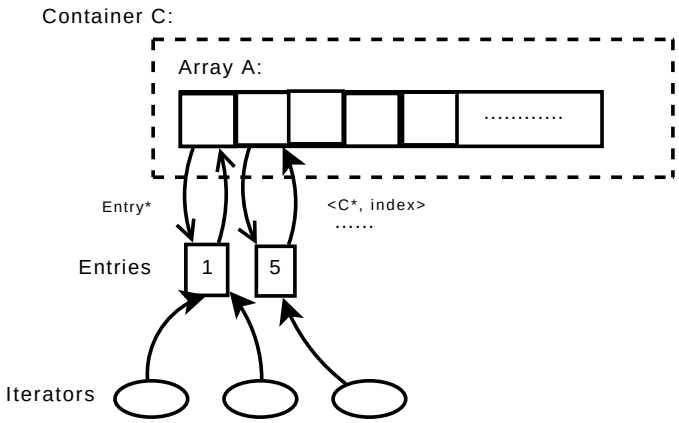


Figure 1. A overview of the entry solution

respect to iteration. As we saw in the previous section, the iterators for nodes and for arrays are usually built in two different ways. With the introduction of the entry class, they can be constructed in almost the same way, since the iterators store a regular pointer (not a tuple). The node and the entry classes are still different i.e. the dynamic array provides random access however containers which are using nodes do not, but the fact that they have some of the same properties are definitely an advantage.

This solution is not perfect there are also some obvious disadvantages; The most significant of them is reduced cache efficiency. The ordinary dynamic array stored the elements directly in the array, so parts of the array could be in the cache, which would give us a better cache utilization (i.e. fast access time). Of course the same could happen for the entry implementation, but accessing an element in the array only gives us the pointer to the entry. The entries could be widely distributed in memory so the chance that two entries are in the cache is not likely, however the chance that two consecutive elements from the regular dynamic array are in the cache is larger.

There is one final advantage that makes this solution very attractive: it provides iterator validity with no extra cost with respect to the complexity requirements given in the C++ standard. With respect to space usage this solution gives a memory overhead of 3 words per element but the space usage is still linear.

3.1 The implementation

The entry mechanism has been implemented during a refactoring process [5], where the original code from the dynamic array was the foundation. An overview of the design can be found in Figure 2. The first step was to design the entry class (code in Appendix A.1). This class consists primarily of two attributes: `v` and `p`. The attribute `v` contains the value which is stored in the entry, and the attribute `p` contains the pointer to the array, which is

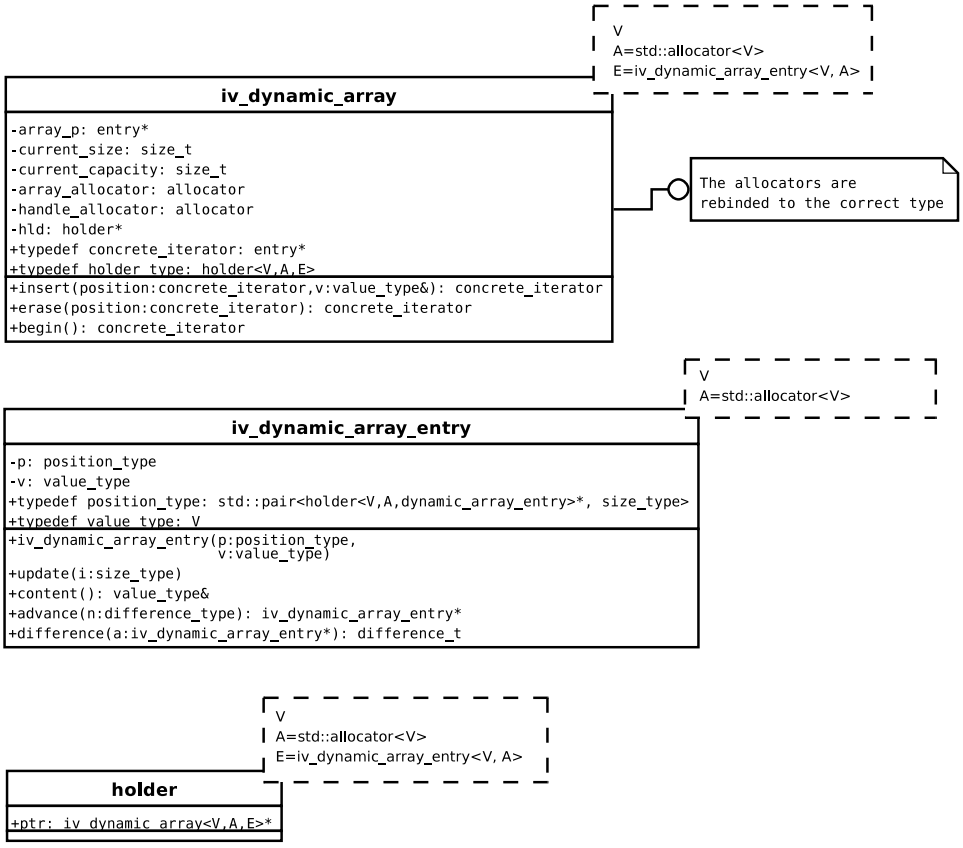


Figure 2. A overview of the entry design

defined by a pair of a pointer to the container (which is encapsulated using a "holder", which is later described) and the index of which position the entry resides in the array. In order to change this pointer we introduced the `update` method which is used when the entry is moved in the array. The entry also contains a parameterized constructor which takes an instance of `p` and `v`. In the CPH STL the entry iterator has become similar to the node iterator such that the entry class implements the methods: `content`, `advance`, `distance`, to define the behaviour of the iterator as defined in [24].

The reason why we have the pointer from the entry back to the container is that it is needed to advance the iterator; `distance` only needs the index in order to calculate the distance. The `advance` method needs to return a pointer to the entry which corresponds to the element located n positions from this entry. In order to get this pointer, we need to access the array of the container. Therefore we need the pointer from the entry to the container.

The next step was to refactor the code of the container. The first thing was to change the container so it contained an array of pointers to entries instead of an array of `V`. Afterwards some patterns of what kind of refactoring

there was needed were identified:

- When inserting a new element we now create an instance of the entry by allocating it and afterwards the copy constructor is invoked.
- When moving elements in the array, we simply move the pointer and use the `update` method to correct the pointer from the entry to the array.
- When the elements are copied to the new array where the element resides at the same position, simply copy the pointer.

The result of the refactoring can be found in Appendix A.2 and Appendix A.3.

There has been a few implementation specific problems: The first one was we need a sentinel entry. Let us examine the scenario where we have an empty array and we insert an element. The insertion call would look like `container.insert(container.begin(), 1)`. The array is of default capacity (which is 1), which means there is one slot in the array, but without a sentinel entry this slot would point to `NULL`; without handling this case it would cause an error. Allocating a sentinel entry (with an uninitialized value since elements do not necessarily have a default constructor) in the constructor of the iterator-valid dynamic array solves this problem. The sentinel entry will always reside in the end of the array which is useful when implementing `end()` since it should return "one element past the end" and the sentinel entry is exactly this element.

The second implementation issue is related to `swap`. As mentioned each entry contains a pointer to the container. If it was a regular pointer swap would be done in linear time, since we need to traverse the array and correct the pointer to the container for each entry in both containers. According to the C++ standard [16] `swap` should be done in constant time, so there is a demand for another solution than the straight forward $\mathcal{O}(n)$ approach. The idea in the new solution is to encapsulate the pointer to the container in a small object referred to as a *holder*. Now each entry contains a pointer to a holder, and a holder is allocated for each container, and each container has a pointer to their holder. When performing `swap`: we first swap the pointer which is contained in the holder, and afterwards we swap the holder, which results in that `swap` is done in constant time.

3.2 Finger search

Finger search trees are described in [22]. The idea of a finger search tree is given a finger (i.e. an iterator/reference to a node) we can search n positions forward or backward in $\mathcal{O}(\lg \delta)$ time, where δ is the number of elements between the source node and destination node. Instead of using a regular search tree where we would start our search at the root, we can now start our search at an arbitrary node. This is profitable since we would use $\mathcal{O}(\lg n)$ time for searching a regular search tree. Finger search can be done because the elements in the tree are levelwise linked together, so every node will contain parent, left, right, previous and next pointers. Forward search is

done by walking up (using the parent pointer) from the source element, then forward (using the next pointer) and after that down (using left/right pointers) until we searched n positions forward. An example of a finger search tree is shown in Figure 3. In the context of the dynamic array this

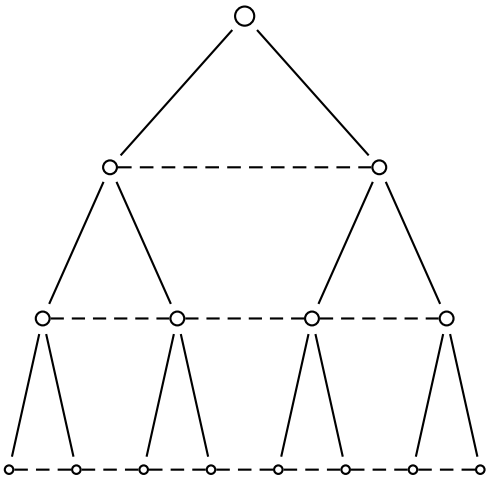


Figure 3. An example finger search tree

data structure could be useful, in order to decouple the entry from the array such that the pointer from the entry back to the array could be removed; this would make the holder class redundant too. Each entry would be a leaf-node in the finger search tree. The finger search tree could now be used in order to advance the iterator; what will happen is that we will perform finger search from the current entry and n steps forward, or n steps backward depending on the sign of n.

Already now we can see that extra space will be required (because of the extra pointers in the entry, and internal nodes too), and iterator operations cannot be performed in constant time as described in [24, 18], but $\mathcal{O}(\lg \delta)$ is a relative small factor, so we will at least consider the complexity for insert and delete operations in a finger search tree. The insert operation should be performed when inserting an element into the array, and the same principle for the delete operation.

According to [22] insertion in the finger search tree is done in expected constant time. Complexity guarantees for deletion and update are not described. According to [7] deletion and update can be done in constant time;, however the solution is very complicated.

Our current solution does not require additional space for performing advance operations, and iterator operations are still done in constant time, the only disadvantage is the very strong connection between the entry and the array, what we desire are only one pointer between the array and the entry. But the cost of using finger search trees is too high to gain any advantages in this context.

4. Snapshot iterators

The paper by Kofler [21] describes an interesting approach to obtain even stronger iterators, these are so-called robust iterators. A *robust iterator* allows modification of the container during traversal of the elements. This means that it would be possible to use the following code for an arbitrary container of the type `container_type`:

```
container_type c;
...
container_type::iterator it;
for(it=c.begin(); it != c.end(); ++it) {
  c.erase(it);
}
```

But in the current C++ standard this is not allowed since iterators will become invalidated after the element which the iterators are pointing to is deleted. To implement robust iterators in the STL could be a very complicated task, and it would even violate the C++ standard since we use iterators which are deleted. To obtain a robust iterator for a container which provides bidirectional iterators and node based, is not that difficult. We will simply mark the element as deleted. But for a container which provides random access iterators it is difficult. In our context of the entry solution the element gets removed from the array but the entry would still be alive, and there may become inconsistency related to the pointer from the entry to the array (it does not get updated any more). The associated structure described in the section on finger search could help us, but we concluded it was inefficient and complicated to implement.

Instead of implementing a robust iterator construction, we implement a snapshot iterator construction instead, which provides the same and some additional properties as the robust iterator. A *snapshot iterator* is defined to contain a snapshot of a container which is kept valid at any time even if the container is destroyed. In concurrent programming a snapshot iterator would be useful. When doing traversal of a container there is a possibility of the container can be modified by other threads, which results in inconsistence. If we did not have the snapshot iterator, this problem could be solved by locks/mutexes but this would slow down execution (i.e. we should wait for the traversal to finish before doing modifications). The snapshot iterator is not part of the C++ standard, but it is meant to be a CPH STL specific extension. According to [4] the CPH STL should also support concurrent programing, where the snapshot iterator could be an essential extension.

Implementing a regular snapshot iterator where each element in the container is copy constructed is trivial, but we require that each element exists once in memory in order to obtain better space efficiency. Objects can be very large and with the current implementation of the dynamic array we use linear space with no constants, we want to keep this property, even if we have several snapshots.

The snapshot iterator is implemented as a *lightweight container*, in this context it means that the container only provides a subset of the methods

which a minimal container provides. It could also be implemented as a mix between a container and an iterator, such that the it should be possible to seek to the beginning and the end of the snapshot, and it should be possible to increase/decrease and dereference the iterator. In STL terminology, the snapshot iterator can be addressed as a container (it contains several elements) so it would be natural to implement it as a container.

The snapshot iterator provides the methods: `begin()`, `end()`, and `operator[]`. This means we can get iterators to the beginning and the end of the snapshot iterator, and we can explicitly get element number n using the `[]` operator. The snapshot iterator uses an ordinary array (since it is just created once, there is no need for extending) of iterators. These iterators origins from the container of which the snapshot was created. This means that the snapshot iterator is not bound to the entry, since it is located at the abstraction layer [24], and it is only using components from the abstraction layer.

We desire that we can implement the example, where we used robust iterators to manipulate the container while doing traversal with a snapshot iterator:

```
container_type c;
typedef containter_type::iterator container_iterator;

...

typedef cphstl::snapshot_iterator<container_iterator>
    snapshot_iterator_type;
/* create a snapshot of the entire data structure */
snapshot_iterator_type s(c.begin(), c.end());
snapshot_iterator_type::iterator ss = s.begin();

while(ss != s.end()) {
  std::cout << "ss:␣" << *(*ss) << std::endl;
  c.erase(*ss);
  ++ss;
}
```

We will now briefly discuss the implementation and design of the snapshot iterator. In Figure 4 an overview of the design is shown. The design consists of two classes, the `snapshot_iterator` class which represents the lightweight container and the `snapshot_iterator_entry` which is used as a storage policy, and furthermore contains methods which define iteration (as described in the previous section). The `snapshot_iterator` class takes three template parameters: the iterator type `I`, the allocator `A`, and the `E` entry. The entry defaults to the `snapshot_iterator_entry` which is implemented in a similar way as the entry described in the previous section; the only difference is that here we have an array of entries (snapshot iterator entries) and not an array of pointers to entries. The `snapshot_iterator` class is created using the parameterized constructor which takes two iterators: the first argument is the iterator pointing to the first element in the sequence, and the second argument is the iterator pointing to the last element in the sequence. The sequence is the set of elements in the container of which we want the snapshot. The parameterized constructor allocates space for the

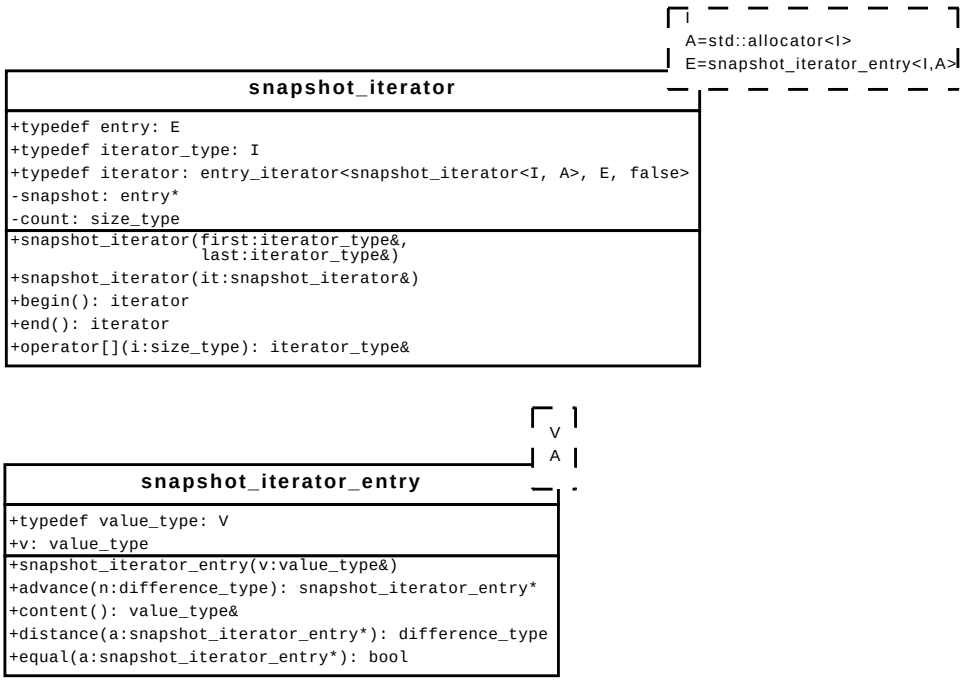


Figure 4. An overview of the snapshot iterator design

array (which is freed in the destructor), and copy construct the iterator for every single element between the two iterators given as argument.

The current design is robust such that modifications (e.g. insertions) of the container will not affect the snapshot. This is because our underlying containers provide iterator validity, but since we only store the actual value in one place, the snapshot can become inconsistent because of deletion. What happens when a deletion occurs is that the entry will be freed and the snapshot iterator contains an iterator pointing to this element, which is now a pointer pointing to a deallocated element. Using this iterator will cause a runtime error; this makes our snapshot iterator construction insufficient. There is a demand for a mechanism where we keep the entries alive in memory even after deletion, in order to keep the snapshot iterator valid at any time. Another snapshot iterator construction is described in [13, Section 3.2], but their construction has no support for deletion either, and is only meant to be used during traversal.

What we want our new mechanism to do is: When a deletion takes place, the element will be removed from the array, but the entry should still be alive until there is no references pointing at it, then it is safe to deallocate the memory. In compiler theory there are several techniques for garbage collection which could be useful in order to solve this problem, since our problem of destroying an object until there is no references to it is an application of the garbage collection mechanism. One of these techniques is *reference*

counting [3], where we keep track of the number of references there exist for each object using a counter; when this counter reaches zero the object will be freed. To implement this causes a complicated design problem since we want to preserve the existing implementation of the iterator-valid dynamic array. An overview of this design is shown in Figure 5. The key element

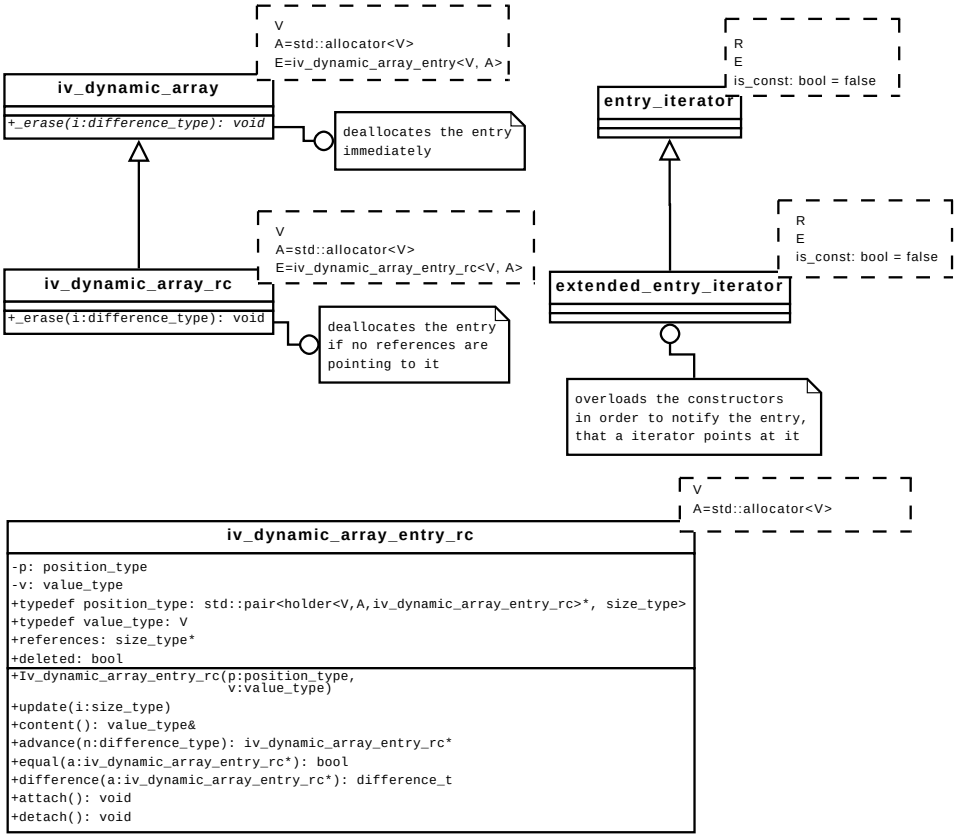


Figure 5. An overview of the design of the dynamic array supporting reference counting

in this design is inheritance, since most of the functionality is identical for the iterator valid dynamic array and the reference counted iterator-valid dynamic array. Where the functionality is not identical is where we deallocate the entry. For the plain iterator-valid dynamic array we should deallocate the entry right away, but for the reference counted iterator-valid dynamic array we will deallocate the entry if and only if the reference counter has reached zero. Else we will mark it as being deleted and the memory will be freed afterwards. With this design there is a few additional requirements for the entry class associated with the reference counted iterator-valid dynamic array: beyond the "reference counter" and the "deleted" attribute it would require two methods. These methods are `attach()` and `detach()`: The `attach()` method will simply increment the reference counter; the `detach()`

method will decrement the reference counter, and if the entry is marked as deleted and the reference counter has reached zero the entry will be freed. The `advance()` method is a bit different too since we will call `detach()` on the old element and `attach()` on the new element, since the iterator has moved its position.

The last thing we need is to handle is the constructors, and the destructor of the iterators. The reference counter should be increased when the constructors are invoked and it should be decreased when the destructor is invoked. This is not possible to do with the existing iterator infrastructure, so we will introduce the `extended_entry_iterator`. The *extended entry iterator* allows `attach()` to be called during construction, simply by defining the parameterized constructor and copy constructor, the same principle for the destructor is used. It inherits the regular methods from the entry iterator such that we still have a high degree of code reuse. Since some methods create iterators with explicit types inside the methods, some additional methods have been overloaded in order to get the implementation working. We can now use our new reference counted iterator-valid dynamic array in this way:

```
typedef int V;
typedef std::allocator<int> A;
typedef cphstl::iv_dynamic_array_rc<V, A, cphstl::
    iv_dynamic_array_entry_rc<V,A> > KERNEL;
typedef cphstl::extended_entry_iterator<KERNEL, cphstl::
    iv_dynamic_array_entry_rc<V,A>, false> ITERATOR;
typedef cphstl::extended_entry_iterator<KERNEL, cphstl::
    iv_dynamic_array_entry_rc<V,A>, true> CONST_ITERATOR;
typedef cphstl::vector<V, A, KERNEL, ITERATOR, CONST_ITERATOR> cont;
typedef cont::iterator cont_it;
```

Notice that the type of the iterators is given as template argument this is one of the results of the already mentioned iterator refactoring. With this instantiation of our container class it is now safe to use the snapshot iterator with this instance of the container.

The advantages are clear: We can take a snapshot of a data structure which is consistent at any time, without using additional space for storing values. Another detail is that it is possible in an idiomatic way for the user to get random access to a container which is based on bidirectional iterators simply by creating a snapshot. It will of course take $\mathcal{O}(n)$ time and space, but performing e.g. divide-and-conquer algorithms might be simpler and maybe faster.

The only disadvantage which has been found is: If the user tries to use the snapshot iterator on a data structure which is not using reference counting he gets a runtime error; there should definitely be implemented some kind of safety. One proposal is to make a typedef inside the containers iterator if it is snapshot ready, and then use static assertions in the snapshot iterator to check that this typedef exists.

The source code for the snapshot iterator can be found in Appendix A.5, and the code for the reference counted iterator-valid dynamic array can be found in Appendix A.7 and the associated entry in Appendix A.8. The

source code for the extended entry iterator can be found in Appendix A.6.

4.1 Semantics

In the foreword of [11], the key design principles behind the STL are mentioned. One of them is *value semantics* this means that each container is viewed as a composition of values, when the container is copied every value is copied and when the container is destroyed every value is destroyed. This yields that every insert method in any container takes values and not objects (node, entry); iterators are only used to locate elements in the container. In the algorithmic literature [9] *pointer semantics* is used, so every method for any container takes an object, since memory allocation makes complexity analysis hard.

We desire that it should possible to use pointer semantics in the STL. An example of where it would be useful is: we have two containers of the same type and we want to move elements between these containers. The usual way of doing that in the STL is to delete the element from the source container and insert it in the destination container, but with value semantics this causes memory allocation and deallocation. Furthermore, the iterators given out will be invalidated. This happens because the container is based on values and it is the containers responsibility to handle memory allocations.

But we want it to be done faster such that we can take one element out from the source container and insert it into the destination container without deallocating/allocating memory and still keep iterators valid. With our existing implementation of a data structure which supports reference counting, this can be done with very few changes to the existing code. Let us consider what happens when we take an element out (i.e. erase it) from the container which uses reference counting: we have an iterator pointing to the object we want to erase. The erase method is invoked, but the element is not deallocated since there is still at least one iterator pointing to it, but it is marked as deleted, and it is removed from the data structure (e.g. the array).

We can now use the iterator to insert it into the destination container, but with the current implementation we will be forced to allocate a new node/entry which is done by the insert method. So there is a demand for a new insert method which does the following: Do relevant changes to the data structure in order to insert the element, but insert the element which the input iterator points at without copying it and finally mark the element as not deleted.

This has been implemented, by adding a method `ps_insert` (ps for pointer semantics) to the `iv_dynamic_array_rc` class. It is placed there because only the reference counted iterator-valid dynamic array should offer this method. In order to make this method visible to the user, the bridge class needs to be extended. This has been done by creating a class `ps_vector` which inherits from the vector class and then adds the extra method `ps_insert`. This method is not added directly into `vector` because

the vector class should only consist of public methods which are specified in the C++ standard; the CPH STL relevant extensions should be made by creating a new class, and let that class inherit from the original class. We can now use the code below to move elements without performing memory allocations:

```
cont v;
...
it1 = v.insert(v.begin(), 27);
v.erase(it1);

cont v2;
v2.ps_insert(v2.begin(), it1);

std::cout << "Test:␣" << *it1 << std::endl;
assert(v2[0] == *it1);
assert(v.size() == 0);
```

To summarize the advantages: This extension can speed up computation, reduce memory usage and increase flexibility of iterator usage (iterator validity is preserved). Again, only one disadvantage has been found: this concept requires reference counting, some static assertions should be made to ensure that a user does not use this concept with a container that does not support reference counting.

The code for the `ps_vector` class can be found in Appendix A.9.

5. Persistent arrays

The snapshot iterator uses linear space (to store iterators) for every snapshot. This can result in quadratic space usage e.g. if we insert n numbers iteratively and we take a snapshot after each insertion, this results in $n + \sum_{i=0}^{n} i = n + \mathcal{O}(n^2) = \mathcal{O}(n^2)$ space usage. The snapshot iterator is a straight forward solution, and it can be done better with respect to space efficiency. A *persistent data structure* is specifically designed to keep previous revisions of the data structure. There are two types of persistent data structures, a *partially persistent* data structure which means that updates can only be performed at the current revision and not for previous revisions which *fully persistent* data structures support. In this study we will only focus on partially persistent data structures, since our snapshot iterator mechanism only requires that; we will not alter the contents of snapshot iterators, so there is no need for fully persistent data structures.

We will look into persistent binary tree data structures, a description of these are given in [23]. The tree structure is useful for our array implementation, since what our snapshot iterator did was: it stored the order and references to elements. We can do the same with a persistent tree, since we can get the tree structure for revision r and do an in-order traversal to construct the snapshot. We will not construct the entire snapshot, but simply perform iteration. How pointers to entries are stored in arrays and trees are shown in Figure 6. Iteration over the elements stored in a revision can be done in $\mathcal{O}(\lg n)$ [9], but with the snapshot iterator we could do it in constant

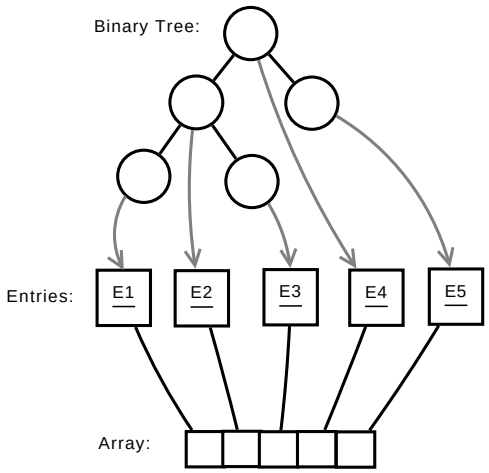


Figure 6. The tree versus the array representation of pointers to entries. The tree in the top, and the array in the bottom.

time however if we can reduce space usage this price is worth paying. We will now study the two methods of obtaining a persistent binary search tree described in [23].

5.1 Path copying

The path copying method is a very simple mechanism for obtaining a persistent binary search tree. The idea in *path copying* is to make a copy of each node before modifying it. This means when an insertion occurs we need to copy each node on the path until we finally reach the position of where the element should be inserted. For balanced search trees we need to take a copy of each node there is involved in rotations. We still maintain the same pointer to unmodified nodes (i.e. no extra space), such that our hope is that we will use $\mathcal{O}(\lg n)$ extra space per revision, because the path has length $\mathcal{O}(\lg n)$ and each rotation will only require one additional allocation. For m insertions this would mean: $\sum_{i=0}^{m} \mathcal{O}(\lg i) = \mathcal{O}(m \lg m)$ extra space; instead of $\mathcal{O}(n^2)$ which we would use to do the same with the snapshot iterator. An example of an insertion in a persistent binary search tree using path copying is shown in Figure 7. As one can see in the figure we keep pointers to the root node of each revision in an array. This means that we can only hold a constant number of revisions. Besides the array we should also store the index of the current revision. We do also need yet another array to store the size of each revision.

To store an unbounded number of revisions, we can use an ordinary binary search tree, to store the root pointer and size of each revision. Access and insertion time for n nodes (m revisions) will now be $\mathcal{O}(\lg m + \lg n)$. We can use a plain dynamic array but this would increase insertion complexity because the array will get expanded during insertions which will cost $\mathcal{O}(m)$

time. The operations `push_front` and `pop_back` can be done in constant time according to [20]. This solution is optimal since access and insertion time is done in $mathcalO(\lg n)$. With respect to implementation we can

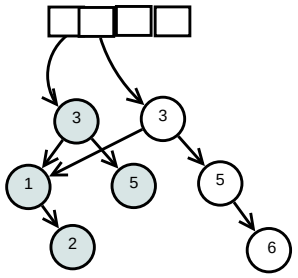


Figure 7. An insertion using path copying. Grey nodes are revision 0 and white nodes are revision 1

already now see there is some problems related to the implementation: It is not possible to maintain a pointer to the parent node inside each node. Let us consider the scenario given in Figure 7. There is no problem in setting parent pointers for the modified (white) nodes, but when we reach the root node, the other subtree is not consistent with respect to this revision, the parent pointer of the first node in the left subtree still points at the root node from the other revision. To make the tree consistent for the current revision, we need to copy the entire left subtree, which results in that we have a copy of the entire tree, and then the concept of this data structure is no longer be relevant.

The path copy mechanism has been implemented using an AVL tree [1]. All existing implementations of the red-black tree [15] in the CPH STL were based under the assumption that parent pointers could be used. Therefore it was not possible to refactor an existing implementation, so the AVL implementation was written from scratch. The path copy mechanism works for any kind of balanced binary search trees.

Most of the implementation work has been simple. With respect to insert/erase operations, we search the tree recursively, until we find the node we searched for, and then perform the relevant balancing operations. We ensure that input pointers for the recursive methods are already in the current revision, which means they are copied before the call. An example is insert: we check if we should go left, then we copy the left node (and link the new allocated node to the current node) and call the insert function again with the left node as parameter.

The component (persistent AVL tree) is designed as an independent component, so it is not bounded to where we need it in this case the dynamic array. This means CPH STL can offer this kind of data structure, and other components can use it as well, in order to solve problems where this data structure is a part of the solution. As an example, this data structure could be relevant for the string implementation in the CPH STL [8], when chan-

ges in a string occur we can use the persistent search tree to represent the change, and thereby save space. The source code of the implementation can be found in Appendix A.11, Appendix A.10, Appendix A.12.

We will now look at some non-trivial implementation details: The biggest implementation issue was how to handle iteration. The usual way to implement iterators for tree data structures is to have a linked list as an associated data structure, where we can traverse the list and get the elements in sorted order (equivalent to in-order traversal). With our implementation this became difficult to maintain; of course we can keep the original revision for the nodes and then link them together on insert such that our iterators will only select the nodes from the requested or older revision while traversing the list. This gives us a problem related to exception safety for the data structure. A natural property for our data structure is strong exception safety, when an exception occurs we will just decrease the current revision variable (and deallocate the memory which was allocated before the exception but this is a detail). Now, the data structure is in the same state as before the exception causing operation was invoked. To summarize: we do not want any modifications to be made in nodes from the previous revisions.

The other possibility of creating iterators is to use the method described in [9] of finding successor and predecessor, but it requires that we have a pointer inside each node to its parent, which we do not have. It would also violate the CPH STL requirement that iterator operations should not take more than constant time. But with this particular data structure it is hard not to violate this requirement.

We have now considered the most common approaches for iterator design. We need another approach: since our tree is static for each revision, modifications will cause a new revision we can use a stack to keep the parent pointers. The idea is when a node x is our current position, the stack consist of the path from x to the root, and now we can use the method from [9]. We could of course just search from the root for the upper bound of x to obtain the successor node and for the lower bound of x to obtain the predecessor node but this requires that the node knows the comparator which can be problematic. This method requires constant space per iterator, however the stack method requires $\Theta(\lg n)$ space per iterator, but the stack method is definitely faster, and does not require the comparator. The idea is discussed in [6]. This introduces a variant of the node iterator called `stack_iterator` (code in Appendix A.16 and Appendix A.15), the only difference is that it holds a pair of a pointer of the node type and the stack instead of just a node pointer, and the `successor` and `predecessor` methods are called with the stack as reference. If we should use the approach which uses the comparator we could not use the existing node iterator either, since the root pointer has to be stored inside each iterator. It cannot be stored inside the node; it would result in inconsistency since a revision can consist of nodes from older revisions.

We do also need a sentinel node since `end()` should return "one past the end". This is handled like a regular node, except for it should always be

the right most element and its value should never be compared. We use a Boolean value to check if a node is the sentinel or not.

Rebalancing is done by computing the balance of the two subtrees of each node on the path from node x to its root. We denote the balance b, which is the difference between the height of the right subtree and the left subtree. If $b \notin [-1; 1]$, we perform rotations depending on the structure of the subtrees. This is normal behavior of an AVL tree but since this tree is persistent, we need to do some extra work. Inside each node we store the revision of when the node was created; when performing rebalancing, we need to check if the node's revision is the current revision we do nothing, else we copy the node since we will not modify nodes from older revisions than the current.

The node's revision variable serves another purpose also. It is used in the destructor of the container. Usually a binary tree is destructed by traversing the entire tree and then deallocate each node. We cannot use this approach here since a revision can contain nodes from older revisions. This means that we would free the same node two or more times. Therefore we traverse every revision and push the nodes which belong in that particular revision on a stack, and finally all nodes on the stack gets deallocated.

Complexity for container operations can be found in Table 1 and for iterator operations in Table 2. The space requirements for the container is $\sum_{i=1}^{m} O(lg\ i) = \mathcal{O}(m\ lg\ m)$ for m revisions, since with one revision we do either add a new node or delete a node, we copy the path which is logarithmic, and perform rebalancing which is also logarithmic.

5.2 Interface design

The STL interface for `set` which is specified in the C++ standard is not designed to provide data structures like a persistent search tree. If we would not allow our users to get earlier revisions of the data structure it would work. But this is one of the purposes of having this data structure. What we desire is to use the data structure in this way:

```
int main() {
  typedef int V;
  typedef std::less<V> C;
  typedef std::allocator<V> A;
  typedef cphstl::persistent_avl_tree<V> R;
  typedef cphstl::persistent_set<V, C, A, R, cphstl::stack_iterator<R
      ::node_type, false>,
          cphstl::stack_iterator<R::node_type, true> > abstraction;

  abstraction s;
  typedef abstraction::iterator si;

  s.insert(5);
  ...
  it = s[2].begin();
  while(it != s[2].end()) {
    ..
  }
  std::cout << s[3].size() << std::endl;
  /* current revision */
  it = s.begin();
```

Table 1. Features, implementation and complexity of methods in the persistent AVL tree (n number of nodes in current revision, m revisions)

Feature	Implementation	**Complexity**
constructor	Initialize member variables, and allocate the sentinel node	$\mathcal{O}(1)$
destructor	Traverse all revisions, find nodes which belong to this revision and push them on the stack, and finally pop all elements from the stack and free them	$\mathcal{O}(m \lg m)$
`insert(V const& v)`	Find where to insert the element, copy nodes on the path, and perform rebalancing on the way back	$\Theta(\lg n)$
`erase(V const& v)`	Same as insert, but delete the element instead	$\Theta(\lg n)$
`begin()`	Find the left most node, push the path on the stack	$\Theta(\lg n)$
`end()`	Find the right most node (the sentinel), push the path on the stack	$\Theta(\lg n)$

```
}
```

This means that we have to design a variant of the `set` bridge class. The proposal of this design is shown in Figure 8. The `persistent_set` class inherits from the `set` class. The `begin()`, `end()`, and `size()` operations are overloaded and the respective methods will be called in the realization with the revision as parameter. The revision should be the current as shown in the code above. The `persistent_set_adapter` class is used to implement the `operator[]`, when this operator is invoked by `s[x]` an object should be returned. This object contains the methods `begin()`, `end()` and `size()` which should be invoked in the realization for revision `x`. By storing a pointer to `R` and the revision parameter inside the `persistent_set_adapter` class, we can invoke the methods in the realization for the requested revision. It does not contain any further methods since the data structure is partially persistent; we do not want modifications of older revisions. We use a pointer to `R` and not a reference, to avoid the realization to be copy constructed inside the `persistent_set_adapter` class.

5.3 Connection to dynamic array

The purpose of implementing this data structure was to obtain better possibility of creating a snapshot iterator implementation for the iterator-valid

Table 2. Features, implementation and complexity of methods in the iterator/node class

Feature	Implementation	**Time**
constructors	Construct the object	$\mathcal{O}(1)$
`it++`	Increment the iterator using `successor()`	$\mathcal{O}(\lg n)$
`it--`	Decrement the iterator using `predecessor()`	$\mathcal{O}(\lg n)$

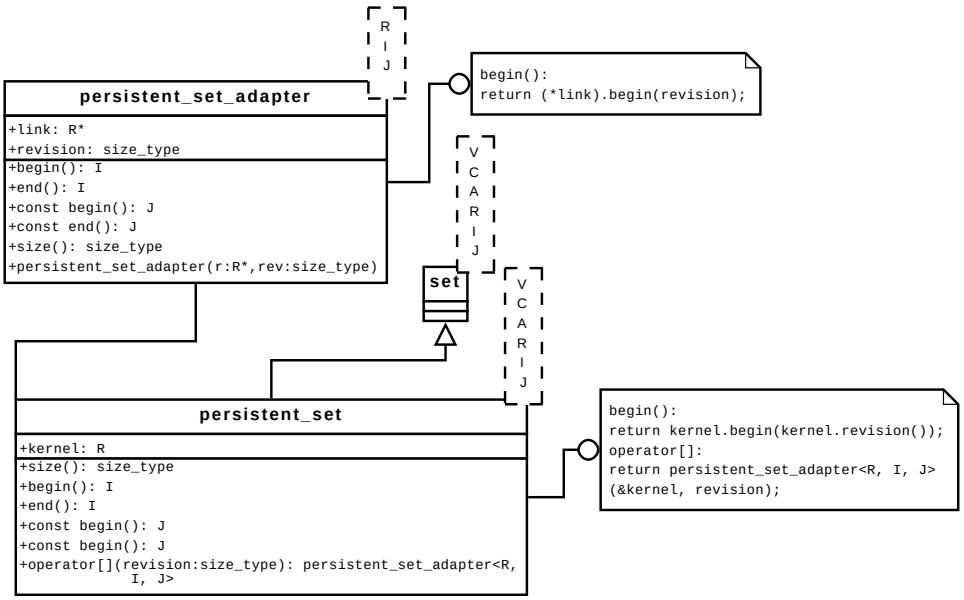


Figure 8. The proposed design of the `persistent_set` bridge class

dynamic array with less space usage than quadratic. The iterator-valid dynamic array should use this implementation by updating (insert/erase) the persistent tree every time a modification happens to the array. We did already move the functionality which deallocated an entry when we designed the reference counting variant of the iterator-valid dynamic array. We moved it into the `_erase` method. We will do the same for insert by creating a method called `_insert` where the allocation of the entry takes place. For the persistent dynamic array (which inherits from `iv_dynamic_array`), the `_insert` method should have some additional functionality namely it should insert the entry in the tree. The same for `_erase` it should erase the entry from the tree. We have a tree of pointers to entries. This means that we need to design a comparator, since the distances between the entries are compared, and a node class since `content()` must return the value contained in the entry. These are given to the persistent set as template parameters, so the task is simply to design two classes, and give them as template ar-

guments to the persistent set. This is done in the persistent dynamic array since the instance of the persistent set is held there. The reason why we need a new node class is iterator related. When we dereference the iterator given by set, we should return the value held inside the entry and not the pointer to the entry which is held in the node, since we should never expose the entry to the user (according to the iterator pattern [14]). The only additional method in the `persistent_dynamic_array` class which should be available to the user is `snapshot()`. This method returns a pair of iterators which is the start iterator and the end iterator. The user can use the iterators at any time to traverse the elements which was present in the array when the `snapshot()` method was invoked. We need to extend the vector class in order to make the snapshot method visible to the user. Now it is possible to use this code:

```
typedef int V;
typedef std::allocator<int> A;
typedef cphstl::persistent_dynamic_array<V, A, cphstl::
    iv_dynamic_array_entry<V,A> > KERNEL;
typedef cphstl::entry_iterator<KERNEL, cphstl::iv_dynamic_array_entry<
    V,A>, false> ITERATOR;
typedef cphstl::entry_iterator<KERNEL, cphstl::iv_dynamic_array_entry<
    V,A>, true> CONST_ITERATOR;
typedef cphstl::persistent_vector<V, A, KERNEL, ITERATOR,
    CONST_ITERATOR> cont;
typedef cont::iterator cont_it;
typedef cont::persistent_iterator cont_pit;

int main() {
  cont v;

  cont_it it1 = v.insert(v.begin(), 5);
  cont_it it2 = v.insert(v.begin(), 7);
  cont_it it3 = v.insert(v.begin(), 9);

  std::pair<cont_pit, cont_pit> p = v.snapshot();

  v.erase(it2);

  for(cont_pit it = p.first; it != p.second; ++it) {
    std::cout << "Elem:␣" << *it << std::endl;
  }
}
```

The traverse will give us 5,7, and 9, even if further modifications are performed to the container. The code for the persistent dynamic array can be found in Appendix A.17 and the code for the persistent vector can be found in Appendix A.18.

5.4 Node copying

The key point in the paper [23] was that it is possible to use $O(1)$ (amortized) space per revision. This can be obtained by using a technique called *limited node copying*. This method requires every node to hold one extra pointer. When a new node is inserted, it will be attached to its parent using the extra pointer. A single bit is used to determine if the extra pointer represents a left or a right subtree. We also need timestamp variables inside each node,

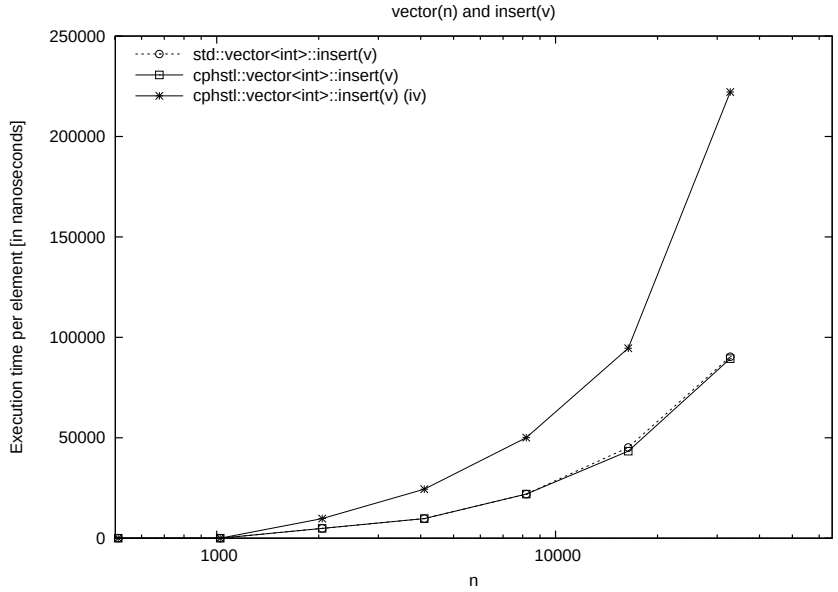
to identify which revision the subtrees belongs to. To be more precise we need a timestamp variable for the left subtree, the right subtree, and the extra subtree. When we are performing an insertion and the extra pointer in the parent is already used we will simply copy the parent node and attach it to the parent's parent extra pointer along with the timestamp such that this node will not be used when we are requesting older revisions. If no extra pointers on the path to the root is vacant, this modification will result in copying all nodes on the path to the root node. The implementation should be done with red-black trees [15], where nodes are either red or black. Nodes are coloured according to certain rules, and rotations are done according to the colours of the neighbour nodes. In the persistent version of the red-black tree the colour of each node should only be stored for the current revision, since our tree is partially persistent i.e. we will not make modifications to earlier revisions. Because we do not copy the entire path, it is now possible to have parent pointers inside each node, which will reduce space usage and complexity of iterators.

But this solution compared to the path copying method is not completely unproblematic. The following observations have been made: This data structure is not exception safe by nature like the persistent AVL tree, because we modify the nodes from the previous revision, so this data structure will be nearly as hard as any other data structure to make exception safe.

6. Benchmarks

In this section, we will present a few benchmarks which show performance of the solution described in Section 3. There are two kinds of benchmarks: benchmarks of insertion and removals. Furthermore they are made using two different types: a built-in type (`int`) and a class type (`my_class`). We used two different data types in order to prove our theory of that the entry solution will perform better on class types. In the self-defined type `my_class` a list of 10 elements was created during object construction. In Figure 9 the `std::vector` and the plain dynamic array are compared to the iterator-valid dynamic array. The containers are used with integers. The plain dynamic array and `std::vector` are as expected identical with respect to efficiency. However the iterator-valid dynamic array performs worser; the reason might be lack of cache utilization as earlier described. In Figure 10 the same containers are used as in Figure 9, but here the containers are used with the type `my_class`. Here the iterator-valid dynamic array performs best, since it is cheaper to move pointers instead of destructing and constructing objects. The efficiency of `std::vector` is reasonable, however the efficiency of the plain dynamic array is unacceptable; there might be an error in the implementation.

a)



b)

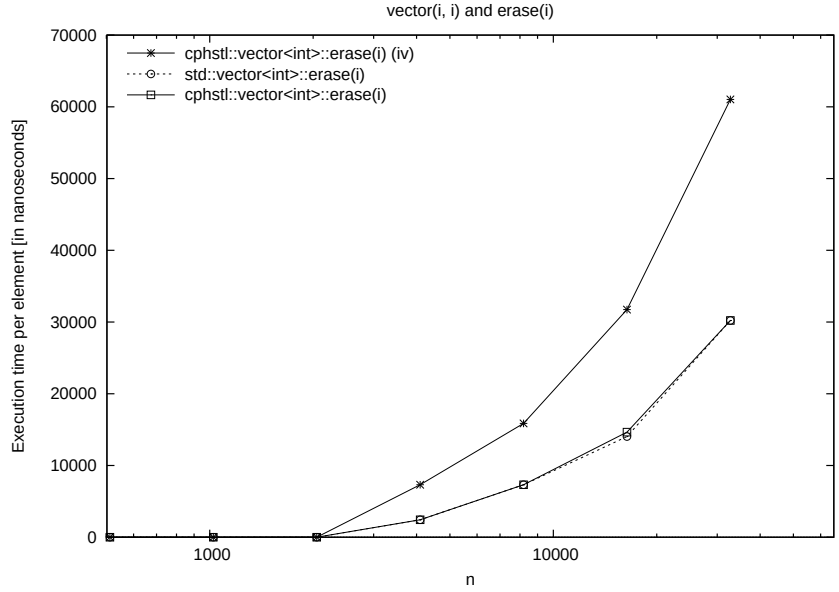
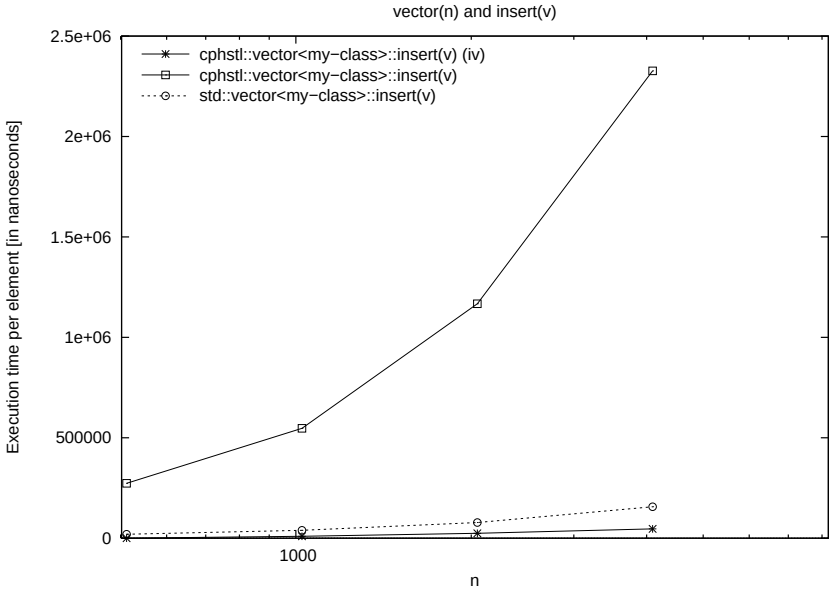


Figure 9. Benchmark of dynamic array (with and without iterator validity) compared to `std::vector<int>`, on integer type, a) insertion b) removal.

a)



b)

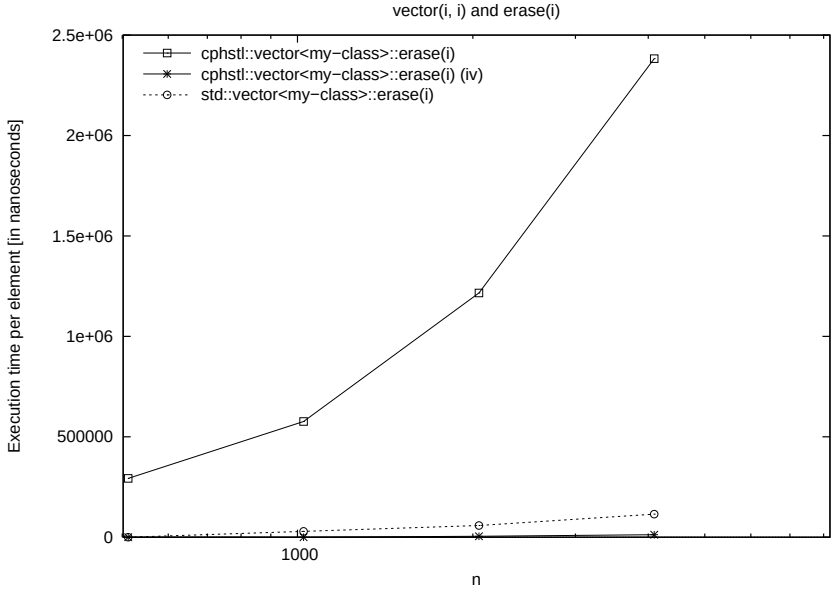


Figure 10. Benchmark of dynamic array (with and without iterator validity) compared to `std::vector<int>` on a class type, a) insertion b) removal.

7. Conclusion

From this study we can conclude the following:

- The entry solution performs worser than expected for built-in types, but better than expected for self-defined types (i.e. class types). This means that the cost of iterator validity depends on which types are used.
- The snapshot iterator is persistent to modifications (insert/erase) of the container. Because we introduced reference counting, we did also add pointer semantics to our data structure with respect to insertion. The missing pointer semantics has been a problem regarding implementation of some data structures e.g. double-ended queue [12].
- The persistent binary tree implementation is not optimal and neither complete, but this work creates a foundation of the design regarding persistent binary tree data structures in the CPH STL.

8. Future work

Here are some thoughts which could be considered useful in order to complete this work:

- In [17] it is showed how traits can be use to differ between class types and the built-in types (`int, char`). We could use these to select the best container for the task, when we are using class types the iterator-valid dynamic array is selected else the normal dynamic array is selected. This will only work if the user does not need iterator validity.
- An exception safe version of the dynamic array has been developed. Our iterator-valid version should be merged with that version in order to obtain a component with stronger guarantees.
- Implement the missing methods into the persistent AVL tree such that the container will become fully compliant to the C++ standard.

Acknowledgements

I want to thank everybody who contributed to the dynamic array implementation in the CPH STL; these people has made this study possible. The individuals who worked on the dynamic array include: Tina A. G. Andersen, Mads D. Kristensen, Ulrik Schou Jørgensen and Claus Ullerlund. Finally I want to thank my supervisor, Jyrki Katajainen, for his guidance and ideas for this project.

References

[1] G. M. Adel'son-Vel'skiĭ and E. M. Landis, An algorithm for the organization of information, *Soviet Mathematics* **3**, 5 (1962), 1259–1263.

[2] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Pattern Applied*, Addison-Wesley (2001).

[3] A. W. Appel, *Modern Compiler Implementation in C*, Cambridge University Press (1998).
[4] C. Artho, C. Jensen, and J. Katajainen, Project description: Foundations and tools for building well-behaved systems, CPH STL Report 2008-5, Department of Computing, University of Copenhagen (2008).
[5] K. Beck, J. Brant, M. Fowler, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Professional (1999).
[6] G. E. Blelloch, B. M. Maggs, S. Leung, and M. Woo, Space-efficient finger search on degree-balanced search trees, *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (2003), 374–383.
[7] G. S. Brodal, G. Lagogiannis, C. Makris, A. Tsakalidis, and K. Tsichlas, Optimal finger search trees in the pointer machine, *Journal of Computer and System Sciences* **67**, 2 (2003), 381–418.
[8] F. Bruman, Designing a generic string class for the cph stl, CPH STL Report 2006-7, Department of Computing, University of Copenhagen (2006).
[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms 2nd edition*, The MIT Press (2001).
[10] Department of Computing, University of Copenhagen, The CPH STL, Website accessible at `http://www.cphstl.dk/` (2000–2008).
[11] G. J. Derge, D. R. Musser, and A. Saini, *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*, 2nd Edition, Addison-Wesley (2001).
[12] A. Elmasry, C. Jensen, and J. Katajainen, Two new methods for transforming priority queues into double-ended priority queues, *Computing* **83** (2008), 193–204.
[13] E. Gamess, D. R. Musser, and A. J. Sánchez-Ruíz, Complete traversals and their implementation using the standard template library, *CLEI Electronic Journal* **1**, 2 (1998).
[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, Addison-Wesley Professional (1995).
[15] L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts, A new representation for linear lists, *Proceedings of the 9th Annual ACM Symposium on the Theory of Computing*, ACM (1977), 49–60.
[16] International Organization for Standardization, *ISO/IEC 14882:2003: Programming languages — C++* (2003).
[17] N. M. Josuttis and D. Vandevoorde, *C++ Templates*, Pearson Education, Inc. (2003).
[18] J. Katajainen, Project proposal: Associative containers with strong guarantees, CPH STL Report 2007-4, Department of Computing, University of Copenhagen (2007).
[19] J. Katajainen, Stronger guarantees for standard-library containers, CPH STL Report 2007-3, Department of Computing, University of Copenhagen (2007).
[20] J. Katajainen and B. B. Mortensen, Experiences with the design and implementation of space-efficient deques, *Proceedings of the 5th Workshop on Algorithm Engineering*, *Lecture Notes in Computer Science* **2141**, Springer-Verlag, Berlin/Heidelberg, Germany (2001), 39–50.
[21] T. Kofler, Robust iterators in ET++, *Structured Programming* **14**, 2 (1993), 62–85.
[22] K. Mehlhorn and S. Näher, *LEDA: A Platform for Combinatorial and Geometric Computing*, Cambridge University Press (1999).
[23] N. Sarnak and R. E. Tarjan, Planar point location using persistent search trees, *Communications of the ACM* **29**, 7 (1986), 669–679.
[24] B. Simonsen, Refactoring the CPH STL: Designing an independent and generic iterator, CPH STL Report 2008-6, Department of Computing, University of Copenhagen (2008).

Appendix

Below the table of contents of the appendix:

The entry solution

The snapshot iterator

Persistent search tree

Appendix A. Source code

Appendix A.1 iv_dynamic_array_entry.h++

```
/*
  A definition of the "Entry" class for dynamic array which should be
      customizing the
  container (it's given as template parameter).

  It also defines the behaviour of the iterator

  Author: Bo Simonsen, May 2008, September 2008

*/

#include <cassert>

namespace cphstl
{
  template <typename V, typename A, typename E>
  class holder {
  public:
    typedef iv_dynamic_array<V, A, E> container_type;
    container_type* ptr;
  };


  template <typename V, typename A>
  class iv_dynamic_array_entry {
  public:
      typedef std::size_t size_type;
      typedef V value_type;
      typedef holder<V, A, iv_dynamic_array_entry> holder_type;
      typedef std::pair<holder_type*, size_type> position_type;
      typedef typename A::template rebind<size_type>::other
          size_type_allocator;
      typedef typename A::template rebind<iv_dynamic_array_entry>::
          other entry_allocator;
      typedef std::ptrdiff_t difference_type;

    protected:
      value_type v;
      position_type p;
      friend class iv_dynamic_array< V, A, iv_dynamic_array_entry<V,A>
           >;

    public:
      ~iv_dynamic_array_entry() {
      }
      iv_dynamic_array_entry(value_type const& _v, position_type _p) {
        (*this).v = _v;
        (*this).p = _p;
      }
      value_type& operator*() {
        return v;
      }
      void update(size_type n) {
        p.second = n;
      }
      value_type& content() {
        return v;
      }
      value_type const& content() const {
        return v;
```

```
      }

      iv_dynamic_array_entry* advance(const difference_type n) const {
        return (*(*p.first).ptr).get_entry(p.second + n);
      }

      difference_type distance(iv_dynamic_array_entry const* a) const
          {
        return p.second - (*a).p.second;
      }

      bool equal(iv_dynamic_array_entry const* a) const {
        return (p.first == (*a).p.first) && (p.second == (*a).p.second
            );
      }
    };

}
```

Appendix A.2 iv_dynamic_array.h++

```
/*
  A dynamic array is a container that doubles its capacity when it
  becomes full, and halves its capacity when it becomes only one-forth
  full.

  Authors: Tina A. G. Andersen, Jyrki Katajainen, Ulrik Schou
      Joergensen and Claus Ullerlund, Bo Simonsen,
  April 2008


  Important : This class cannot be used standalone. This class is a
      _realisation_ to be used with the _bridge-class_ vector(../../
      Vector).

  The design pattern is called a "bridge pattern":

  client <--> bridge <--> realisation

  The advantage of this, is you can change the realisation without
      confusing the client.

*/

#ifndef __CPHSTL_DYNAMIC_ARRAY__
#define __CPHSTL_DYNAMIC_ARRAY__

#include <memory> // defines std::allocator
#include <stdexcept> // defines std::out_of_range
#include <cstddef> // defines std::size_t and std::ptrdiff_t
#include <utility> // defines std::pair
#include <cassert>
#include <iostream>
namespace cphstl {

  template <typename V, typename A>
  class iv_dynamic_array_entry;

  template <typename V, typename A, typename E>
  class holder;

  template <
    typename V,
    typename A = std::allocator<V>,
```

```
  typename E = iv_dynamic_array_entry<V, A>
>
class iv_dynamic_array {

public:
  // types
  typedef V value_type;
  typedef A allocator_type;
  typedef V* pointer;
  typedef E entry;
  typedef V const* const_pointer;
  typedef std::ptrdiff_t difference_type;
  typedef std::size_t size_type;
  typedef typename E::holder_type holder_type;
protected:
  typedef typename entry::position_type position_type;
  typedef entry** array;
public:
  typedef entry* concrete_iterator;
  typedef const entry* concrete_const_iterator;
  // structors

  explicit iv_dynamic_array(A const& = A());
  explicit iv_dynamic_array(size_type, V const& = V(), A const& = A
      ());
  iv_dynamic_array(iv_dynamic_array<V, A, E> const&);
  virtual ~iv_dynamic_array();

  // iterators

  concrete_iterator begin();
  concrete_const_iterator begin() const;

  // accessors

  A get_allocator() const;
  size_type size() const;
  size_type max_size() const;
  size_type capacity() const;
  V const& operator[](size_type) const;

  iv_dynamic_array<V,A>& operator=(const iv_dynamic_array<V,A>&);

  // modifiers

  V& operator[](size_type);
  void reserve(size_type);
  concrete_iterator insert(concrete_iterator, V const&);
  void insert(concrete_iterator, size_type, V const&);

  template <typename I>
  void insert(concrete_iterator, I, I);

  concrete_iterator erase(concrete_iterator);
  concrete_iterator erase(concrete_iterator, concrete_iterator);
  void swap(iv_dynamic_array<V, A, E>&);

  entry* get_entry(size_type num);

protected:

  virtual void _erase(size_type);
  virtual entry* _insert(V const&, size_type);
```

```
    typedef typename A::template rebind<entry*>::other
        entry_ptr_allocator_type;
    typedef typename A::template rebind<entry>::other
        entry_allocator_type;
    typedef typename A::template rebind<holder_type>::other
        holder_allocator_type;

    size_type current_capacity;
    size_type current_size;
    holder_type* hld;
    array array_p;
    entry_ptr_allocator_type array_allocator;
    entry_allocator_type entry_allocator;
    holder_allocator_type holder_allocator;
  };
}

#include "iv_dynamic_array_entry.h++"
#include "iv_dynamic_array.c++" //implements cphstl::iv_dynamic_array

#endif
```

Appendix A.3 iv_dynamic_array.c++

```
/*
  An implementation of cphstl::iv_dynamic_array (doubling array)

  Authors: Tina A. G. Andersen, Jyrki Katajainen, Mads D. Kristensen,
  Ulrik Schou Joergensen and Claus Ullerlund, April 2008
  Bo Simonsen, May 2008, September 2008

  Important: This class cannot be used standalone. This class is a
  realization to be used with the bridge-class vector.

  The design pattern used is called the "bridge pattern":

  client <--> bridge <--> realization

  The advantage of this is you can change the realization without
  confusing the client.
*/

#include <cassert> // defines assert
#include <algorithm> // defines std::swap
#include <cstdio>

namespace cphstl {

  // constructor: create an empty array

  template <typename V, typename A, typename E>
  iv_dynamic_array<V, A, E>::iv_dynamic_array(A const& a)
    : current_capacity(1), current_size(0), array_allocator(a),
        holder_allocator(a), entry_allocator(a) {
    (*this).hld = (*this).holder_allocator.allocate(1);
    (*(*this).hld).ptr = this;

    (*this).array_p = (*this).array_allocator.allocate((*this).
        current_capacity);
    (*this).array_p[0] = (*this)._insert(V(), 0);
  }
  // copy constructor

  template <typename V, typename A, typename E>
```

```
iv_dynamic_array<V, A, E>::iv_dynamic_array(iv_dynamic_array const&
    x)
  : current_capacity(x.current_capacity), current_size(x.
      current_size),
    array_allocator(x.array_allocator), holder_allocator(x.
        holder_allocator),
    entry_allocator(x.entry_allocator) {
  (*this).hld = (*this).holder_allocator.allocate(1);
  (*(*this).hld).ptr = this;

  (*this).array_p = (*this).array_allocator.allocate((*this).
      current_capacity);

  size_type i;

  for (i = 0; i < (*this).current_size; ++i) {
    (*this).array_p[i] = (*this)._insert(*(*x.array_p[i]), i);
  }

  // sentinel entry
  (*this).array_p[i] = (*this)._insert(V(), i);

}

// destructor

template <typename V, typename A, typename E>
iv_dynamic_array<V, A, E>::~iv_dynamic_array() {
  size_type i;
  for (i = 0; i < (*this).current_size; ++i) {
    (*this)._erase(i);
  }
  (*this)._erase(i);
  (*this).array_allocator.deallocate((*this).array_p, (*this).
      current_capacity);
  (*this).holder_allocator.deallocate((*this).hld, 1);
}

// = operator overloading

template <typename V, typename A, typename E>
iv_dynamic_array<V,A>&
iv_dynamic_array<V, A, E>::operator=(const iv_dynamic_array<V,A> &x)
    {

  size_type i;

  for(i = 0; i < (*this).current_size; ++i) {
    (*this)._erase(i);
  }

  // sentinel
  (*this).entry_allocator.deallocate((*this).array_p[i], 1);
  (*this).array_allocator.deallocate((*this).array_p, (*this).
      current_capacity);
  (*this).holder_allocator.deallocate((*this).hld, 1);

  (*this).entry_allocator = x.entry_allocator;
  (*this).array_allocator = x.array_allocator;
  (*this).holder_allocator = x.holder_allocator;

  // copy variables from the other containers
  (*this).current_capacity = x.current_capacity;
  (*this).current_size     = x.current_size;
```

```
  (*this).array_p = array_allocator.allocate((*this).
      current_capacity);

  // allocate the holder
  (*this).hld = holder_allocator.allocate(1);
  (*(*this).hld).ptr = this;

  for(i = 0; i < (*this).current_size; ++i) {
    (*this).array_p[i] = (*this)._insert(*(*x.array_p[i]), i);
  }

  // sentinel
  (*this).array_p[i] = (*this)._insert(V(), i);

  return (*this);
}

// begin: return a random-access iterator referring to the first
    element

template <typename V, typename A, typename E>
typename iv_dynamic_array<V, A, E>::concrete_iterator
iv_dynamic_array<V, A, E>::begin() {
  assert((*this).array_p[0] != 0);
  return concrete_iterator((*this).array_p[0]);
}

template <typename V, typename A, typename E>
typename iv_dynamic_array<V, A, E>::concrete_const_iterator
iv_dynamic_array<V, A, E>::begin() const {
  assert((*this).array_p[0] != 0);
  return concrete_const_iterator((*this).array_p[0]);
}

// get_allocator: return the allocator

template <typename V, typename A, typename E>
A
iv_dynamic_array<V, A, E>::get_allocator() const {
  return A((*this).array_allocator);
}

// size: return the number of elements stored

template <typename V, typename A, typename E>
typename iv_dynamic_array<V, A, E>::size_type
iv_dynamic_array<V, A, E>::size() const {
  return (*this).current_size;
}

// max_size: return the maximum number of elements possible

template <typename V, typename A, typename E>
typename iv_dynamic_array<V, A, E>::size_type
iv_dynamic_array<V, A, E>::max_size() const {
  return (*this).current_capacity + array_allocator.max_size();
}

// capacity: return the maximum number of elements without
    reallocation

template <typename V, typename A, typename E>
typename iv_dynamic_array<V, A, E>::size_type
iv_dynamic_array<V, A, E>::capacity() const {
  return (*this).current_capacity;
```

```
}

// operator[]: return a const reference to the element with index s

template <typename V, typename A, typename E>
V const&
iv_dynamic_array<V, A, E>::operator[](size_type s) const {
  return *(*(*this).array_p[s]);
}

// operator[]: return a reference to the element with index s

template <typename V, typename A, typename E>
V&
iv_dynamic_array<V, A, E>::operator[](size_type s) {
  return *(*(*this).array_p[s]);
}

// reserve: enlarge the capacity, if not large enough

template <typename V, typename A, typename E>
void
iv_dynamic_array<V, A, E>::reserve(size_type s) {
  if (s <= (*this).current_capacity){
    return;
  }
  assert((*this).current_size < (*this).current_capacity+1);
  // Allocate new space for the elements.
  array new_array_p = (*this).array_allocator.allocate(s);

  // Move the objects from the old array into the new one. While we
      do
  // this we also destroy the old objects.

  size_type i;

  for(i=0; i <= (*this).current_size; ++i) {
    new_array_p[i] = (*this).array_p[i];
  }

  // Deallocate the old space and swap the pointers.
  (*this).array_allocator.deallocate((*this).array_p, (*this).
      current_capacity);
  (*this).array_p = new_array_p;
  (*this).current_capacity = s;

  assert((*this).current_size < (*this).current_capacity + 1);
}

// insert: insert a copy of v before position p; return its position

template <typename V, typename A, typename E>
typename iv_dynamic_array<V, A, E>::concrete_iterator
iv_dynamic_array<V, A, E>::insert(concrete_iterator pos, value_type
    const& v) {

  concrete_iterator retval;
  position_type p = (*pos).p;

  assert(p.first == (*this).hld);
  assert((*this).current_size < (*this).current_capacity+1);

  // This has been split up into two parts, one where resizing is
  // needed and one that doesn't need resizing. This way less
  // element moving is done when a resize is needed.
```

```
  if ((*this).current_size == (*this).current_capacity - 1) {

    size_type new_capacity = (*this).current_capacity * 2;
    // Increase the capacity of the array.
    array new_array_p = array_allocator.allocate(new_capacity);

    size_type i, ii;

    // First we copy the elements in front of pos into the new array
        .
    for(i = 0; i < p.second; ++i) {
      new_array_p[i] = (*this).array_p[i];
    }
    // Then we insert the new element and copy the rest of the
    // elements into the new array.

    ii = i;

    while(i <= current_size) {
      new_array_p[i+1] = (*this).array_p[i];
      (*new_array_p[i+1]).update(i+1);
      ++i;
    }

    retval = new_array_p[ii] = (*this)._insert(v, ii);

    // The last thing to do is to deallocate the old array
    // and swap the pointers.
    (*this).array_allocator.deallocate((*this).array_p, (*this).
        current_capacity);
    (*this).current_capacity = new_capacity;
    (*this).array_p = new_array_p;
  }
  else {

    size_type i;

    // In this case we move all the elements from pos and forward
    // one position to the right.
    for(i = (*this).current_size+1; i > p.second; --i) {
      (*this).array_p[i] = (*this).array_p[i-1];
      (*(*this).array_p[i]).update(i);
    }

    retval = (*this).array_p[i] = (*this)._insert(v, i);
  }

  (*this).current_size += 1;
  assert((*this).current_size <= (*this).current_capacity);

  // we will return the entry which represents the new element
  return concrete_iterator(retval);
}

// insert: insert s copies of v before position p; return nothing

template <typename V, typename A, typename E>
void
iv_dynamic_array<V, A, E>::insert(concrete_iterator pos, size_type s
    , value_type const& v) {

  position_type p = (*pos).p;

  assert(p.first == (*this).hld);
  assert((*this).current_size <= (*this).current_capacity);
```

```
  //if we need to resize : calculate new target and resize properly
  if ((*this).current_size + s >= (*this).current_capacity) {
    size_type new_capacity = (*this).current_capacity;

    while((*this).current_size + s >= new_capacity){
      new_capacity*=2;
    }

    //set new target and capacity;
    array new_array_p = (*this).array_allocator.allocate(
        new_capacity);

    size_type i;

    // First we copy the elements in front of pos into the new array
        .
    for (i = 0; i < p.second; ++i) {
      new_array_p[i] = (*this).array_p[i];
    }

    size_type j;
    //Now we insert s elements in front
    for(j = i; j-i < s; ++j){
      new_array_p[j] = (*this)._insert(v, j);
    }
    //copy the rest in and put them in front

    while(i <= current_size) {
      new_array_p[j] = (*this).array_p[i];
      (*new_array_p[j]).update(j);
      ++i;
      ++j;
    }

    (*this).array_allocator.deallocate((*this).array_p, (*this).
        current_capacity);
    (*this).current_capacity = new_capacity;
    (*this).array_p = new_array_p;
  }
  else {
    //If we dont need to increase capity;
    //First copy the old element forward to their new position
    for(size_type i = (*this).current_size + s; i >= p.second + s;
        --i) {
      (*this).array_p[i] = (*this).array_p[i-s];
      (*(*this).array_p[i]).update(i);
    }

    for(size_type i = p.second; i < p.second + s; ++i) {
      (*this).array_p[i] = (*this)._insert(v, i);
    }
  }
  (*this).current_size += s;

  assert((*this).current_size <= (*this).current_capacity);
}

// insert: insert a copy of elements in the range [q,r) before
    position p
// this version is O(n), we can't do better with this data structure
    .
// exactly O(n) + 2|array[p..q]| = O(n)

template <typename V, typename A, typename E>
```

```
template <typename I>
void
iv_dynamic_array<V, A, E>::insert(concrete_iterator p, I q, I r) {

  // Find the number of elements to insert
  I iter = q;
  size_type count = 0;
  while(iter != r) {
    ++count;
    ++iter;
  }

  position_type pos = (*p).p;

  // We let this insert function do the work for us
  (*this).insert(p, count, V());

  iter = q;
  size_type i = pos.second;

  // Destroy the old elements and insert the new ones
  while(iter != r) {
    //assert((*this)[i] == V());
    // using the [] operator
    (*this)[i].~V();
    (*this)[i] = *iter;
    ++iter;
    ++i;
  }
}

// erase: remove the element at position p; return the position of
    the next element

template <typename V, typename A, typename E>
typename iv_dynamic_array<V, A, E>::concrete_iterator
iv_dynamic_array<V, A, E>::erase(concrete_iterator pos) {

  position_type p = (*pos).p;

  // We now have some additional safety with respect to iterators,
  // such that only iterators which points to this container can
  // be used
  assert(p.first == (*this).hld);

  // The container's internal invariant holds
  assert((*this).current_size <= (*this).current_capacity);

  (*this)._erase(p.second);

  size_type i;

  for(i = p.second; i < current_size; ++i) {
    (*this).array_p[i] = (*this).array_p[i+1];
    (*(*this).array_p[i]).update(i);
  }

  --(*this).current_size;

  assert((*this).current_size <= (*this).current_capacity);

  // Let's say p.second is 1, and the array has only one element,
  // we need this one, since the sentinel should then be returned
  // and not array_p[1]
  if(p.second < current_size) {
```

```
    return concrete_iterator((*this).array_p[p.second]);
  }

  return concrete_iterator((*this).array_p[0]);
}

// erase: remove all elements in the range [p,q); return q

template <typename V, typename A, typename E>
typename iv_dynamic_array<V, A, E>::concrete_iterator
iv_dynamic_array<V, A, E>::erase(concrete_iterator pos_p,
    concrete_iterator pos_q) {
  position_type p = (*pos_p).p, q = (*pos_q).p;
  size_type range = q.second - p.second;

  assert(p.first == (*this).hld && q.first == (*this).hld);
  assert(range >= 0);

  for(size_type i = p.second; i < q.second; ++i) {
    (*this)._erase(i);
  }

  size_type j;

  for(j = p.second; j <= ((*this).current_size - range); ++j) {
    (*this).array_p[j] = (*this).array_p[j+range];
    (*(*this).array_p[j]).update(j);
  }

  (*this).current_size -= range;
  return concrete_iterator(pos_q);
}

// swap: swap the data of the calling dynamic array and r

template <typename V, typename A, typename E>
void
iv_dynamic_array<V, A, E>::swap(iv_dynamic_array<V, A, E>& r) {
  assert(array_allocator == r.array_allocator);
  std::swap((*this).array_p, r.array_p);
  std::swap((*(*this).hld).ptr, (*r.hld).ptr);
  std::swap((*this).hld, r.hld);
  std::swap((*this).current_size, r.current_size);
  std::swap((*this).current_capacity, r.current_capacity);
}

// get_entry - only used by the iterator_entry class to get the
// entry, no encapsulation is needed, since the user of this
// container has no direct access to the container instance

template <typename V, typename A, typename E>
typename iv_dynamic_array<V, A, E>::entry*
iv_dynamic_array<V, A, E>::get_entry(size_type num) {
  return (*this).array_p[num];
}

// _erase

template <typename V, typename A, typename E>
void iv_dynamic_array<V, A, E>::_erase(size_type num) {
  (*(*this).array_p[num]).~entry();
  (*this).entry_allocator.deallocate((*this).array_p[num], 1);
}

template <typename V, typename A, typename E>
```

```
  typename iv_dynamic_array<V, A, E>::entry*
  iv_dynamic_array<V, A, E>::_insert(value_type const& v, size_type
      num) {
    entry* tmp = (*this).entry_allocator.allocate(1);
    new (tmp) entry(v, position_type((*this).hld, num));
    return tmp;
  }
}
```

Appendix A.4 test_validity.c++

```
#include "iv_dynamic_array_rc.h++"
#include "ps_vector.h++"
#include "extended_entry_iterator.h++"
#include "snapshot_iterator.h++"

typedef int V;
typedef std::allocator<int> A;
typedef cphstl::iv_dynamic_array_rc<V, A, cphstl::
    iv_dynamic_array_entry_rc<V,A> > KERNEL;
typedef cphstl::extended_entry_iterator<KERNEL, cphstl::
    iv_dynamic_array_entry_rc<V,A>, false> ITERATOR;
typedef cphstl::extended_entry_iterator<KERNEL, cphstl::
    iv_dynamic_array_entry_rc<V,A>, true> CONST_ITERATOR;

typedef cphstl::ps_vector<V, A, KERNEL, ITERATOR, CONST_ITERATOR> cont
    ;
typedef cont::iterator cont_it;

int main() {

  cont v;

  cont_it it1 = v.insert(v.begin(), 5);
  cont_it it2 = v.insert(it1, 7);
  cont_it it3 = v.insert(v.begin(), 9);
  cont_it it4 = v.begin();
  it4 += v.size();

  std::cout << "Test" << std::endl;

  std::cout << "It:␣" << *it1 << std::endl;
  assert(*it1 == 5);
  assert(*it2 == 7);
  assert(*it3 == 9);

  typedef cphstl::snapshot_iterator<cont_it> si;
  si s(v.begin(), v.end());
  si::iterator ss = s.begin();

  while(ss != s.end()) {
    std::cout << "ss:␣" << *(*ss) << std::endl;
    v.erase(*ss);
    ++ss;
  }
  ss = s.begin();
  while(ss != s.end()) {
    std::cout << "ss:␣" << *(*ss) << std::endl;
    ++ss;
  }
  std::cout << *it1 << std::endl;
  std::cout << *it2 << std::endl;
  std::cout << *it3 << std::endl;

  it1 = v.insert(v.begin(), 27);
```

```
  v.erase(it1);

  cont v2;
  v2.ps_insert(v2.begin(), it1);

  std::cout << "Test:␣" << *it1 << std::endl;
  assert(v2[0] == *it1);
  assert(v.size() == 0);
}
```

Appendix A.5 snapshot_iterator.h++

```
/*
  Implementation of cphstl::snapshot_iterator

  Author: Bo Simonsen, October 2008
*/

#include <cassert> // defines assert macro
#include <cstddef> // defines std::size_t and std::ptrdiff_t
#include "entry_iterator.h++"

namespace cphstl {

  template <typename V, typename A>
  class snapshot_iterator_entry;

  template <typename I, typename A = std::allocator<I>, typename E =
      snapshot_iterator_entry<I, A> >
  class snapshot_iterator {
    public:
      typedef I iterator_type;
      typedef typename A::template rebind<E>::other allocator_type;
      typedef std::size_t size_type;
      typedef I value_type;
      typedef E entry;

      typedef cphstl::entry_iterator<snapshot_iterator<I, A>, E, false
          > iterator;
      typedef cphstl::entry_iterator<snapshot_iterator<I, A>, E, true>
           const_iterator;
      typedef E concrete_iterator;
      typedef E concrete_const_iterator;

      /* parameterized constructor */
      snapshot_iterator(iterator_type const& first, iterator_type
          const& last) {
        I tmp = first;
        size_type count = 0;
        while(tmp != last) {
          ++count;
          ++tmp;
        }

        // we will include the sentinel
        ++count;

        (*this).snapshot = allocator.allocate(count);

        size_type i = 0;
        tmp = first;
        while(tmp != last) {
          new (&(*this).snapshot[i]) E(tmp);
          ++tmp;
          ++i;
```

```
      }

      new (&(*this).snapshot[i]) E(tmp);

      (*this).count = i;
      (*this).position = 0;
    }

    /* destructor */
    ~snapshot_iterator() {
      std::cout << "+Snapshot␣destruct" << std::endl;
      for(size_type i=0; i <= (*this).count; ++i) {
        (*(&(*this).snapshot[i])).~E();
        std::cout << i << std::endl;
      }
      allocator.deallocate((*this).snapshot, (*this).count);
      std::cout << "-Snapshot␣destruct" << std::endl;
    }

    /* copy constructor */
    snapshot_iterator(snapshot_iterator const& it) {

      std::cout << "Copy␣constructor" << std::endl;
      (*this).count = it.count;

      (*this).snapshot = allocator.allocate((*this).count);

      for(size_type i = 0; i < it.count; ++i) {
        new (&(*this).snapshot[i]) E(it.snapshot[i]);
      }

      (*this).position = 0;
    }

    iterator begin() {
      return iterator(&(*this).snapshot[0]);
    }

    iterator end() {
      return iterator(&(*this).snapshot[count]);
    }

    iterator_type& operator[](size_type i) {
      return (*this).snapshot[i];
    }

  private:
    E* snapshot;
    allocator_type allocator;
    size_type count;
    size_type position;

};

template <typename V, typename A>
class snapshot_iterator_entry {
public:
  typedef V value_type;
  typedef std::ptrdiff_t difference_type;

  value_type v;

  snapshot_iterator_entry(value_type& _v) {
    v = _v;
  }
```

```
    snapshot_iterator_entry* advance(difference_type const n) {
      std::cout << "Advance" << n << std::endl;
      return this + n;
    }
    value_type& content() {
      return v;
    }
    difference_type distance(snapshot_iterator_entry const* a) const {
      return a - this;
    }
    bool equal(snapshot_iterator_entry const* a) const {
      return this == a;
    }
  };



}
```

Appendix A.6 extended_entry_iterator.h++

```
/*
  Implementation of cphstl::extended_entry_iterator
  Supports operations attach/detach in order to notify the entry,
  when a iterator is constructed/destructed.

  This iterator types applications includes reference counting, where
      we
  need to inform the handle, that a iterator is now pointing at it.

  Author: Bo Simonsen, October 2008
*/

#include "entry_iterator.h++"

namespace cphstl {
  template <typename R, typename E, bool is_const = false>
  class extended_entry_iterator : public entry_iterator<R, E, is_const
      > {
  public:

    extended_entry_iterator() : entry_iterator<R, E, is_const>() {
    }
    extended_entry_iterator(extended_entry_iterator<R, E, false> const
        & a)
    : entry_iterator<R, E, false>(a.position) {
      (*(*this).position).attach();
    }
    extended_entry_iterator(extended_entry_iterator<R, E, true> const&
         a)
    : entry_iterator<R, E, true>(a.position) {
      (*(*this).position).attach();
    }
    extended_entry_iterator& operator=(extended_entry_iterator const&
        a) {
      if((*this).position != 0) {
        (*(*this).position).detach();
      }
      (*this).position = a.position;
      (*(*this).position).attach();
      return *this;
    }

    ~extended_entry_iterator() {
```

```
      (*(*this).position).detach();
    }

    extended_entry_iterator(typename entry_iterator<R,E,is_const>::
        node_pointer const& _position)
    : entry_iterator<R, E, is_const>(_position) {
      (*(*this).position).attach();
    }

    extended_entry_iterator operator+(typename entry_iterator<R,E,
        is_const>::difference_type n) const {
      extended_entry_iterator<R, E, is_const> temporary = *this;
      temporary.position = (*temporary.position).advance(n);
      return temporary;
    }
    extended_entry_iterator operator-(typename entry_iterator<R,E,
        is_const>::difference_type n) const {
      extended_entry_iterator<R, E, is_const> temporary = *this;
      temporary.position = (*temporary.position).advance(-n);
      return temporary;
    }

    // These methods need to be overloaded because the iterator class
        need to
    // be explicitly defined inside the methods.

    extended_entry_iterator& operator++() {
      (*this).position = (*(*this).position).advance(1);
      return *this;
    }
    extended_entry_iterator operator++(int) {
      extended_entry_iterator<R, E, is_const> temporary = *this;
      (*this).position = (*(*this).position).advance(1);
      return temporary;
    }
    extended_entry_iterator& operator--() {
      (*this).position = (*(*this).position).advance(1);
      return *this;
    }
    extended_entry_iterator operator--(int) const {
      extended_entry_iterator<R, E, is_const> temporary = *this;
      (*this).position = (*(*this).position).advance(1);
      return temporary;
    }

  };
}
```

Appendix A.7 iv_dynamic_array_rc.h++

```
/*
  This extention adds supports reference counting, which should be
      used with the iv_dynamic_entry_rc
  class.

  Authors: Bo Simonsen, November 2008
*/

#include "iv_dynamic_array.h++"

template <typename V, typename A>
class iv_dynamic_array_entry_rc;

namespace cphstl {
  template <
```

```
  typename V,
  typename A = std::allocator<V>,
  typename E = iv_dynamic_array_entry_rc<V, A>
>
class iv_dynamic_array_rc : public iv_dynamic_array<V, A, E> {
public:

  explicit iv_dynamic_array_rc(A const& a = A()) : iv_dynamic_array<
      V,A,E>(a) {}
  explicit iv_dynamic_array_rc(typename iv_dynamic_array<V, A, E>::
      size_type s, V const& v = V(),
      A const& a = A()) : iv_dynamic_array<V, A, E>(s, v, a) {}
  iv_dynamic_array_rc(iv_dynamic_array<V, A, E> const& o) :
      iv_dynamic_array<V, A, E>(o) {}
  typedef V value_type;
  typedef E entry;
  typedef entry* concrete_iterator;
  typedef typename entry::position_type position_type;
  typedef std::ptrdiff_t difference_type;
  typedef std::size_t size_type;
  typedef entry** array;
protected:
  void _erase(size_type num) {
    if((*(*this).array_p[num]).references == 0) {
      (*(*this).array_p[num]).~E();
      (*this).entry_allocator.deallocate((*this).array_p[num], 1);
      std::cout << "Delete␣now" << std::endl;
    } else {
      (*(*this).array_p[num]).deleted = true;
      std::cout << "Delayed␣delete" << std::endl;
    }
  }
public:
  void ps_insert(entry* pos, entry* element) {
    concrete_iterator retval;
    position_type p = (*pos).p;

    assert(p.first == (*this).hld);
    assert((*this).current_size < (*this).current_capacity+1);

    // This has been split up into two parts, one where resizing is
    // needed and one that doesn't need resizing. This way less
    // element moving is done when a resize is needed.
    if ((*this).current_size == (*this).current_capacity - 1) {

      size_type new_capacity = (*this).current_capacity * 2;
      // Increase the capacity of the array.
      array new_array_p = (*this).array_allocator.allocate(
          new_capacity);

      size_type i;

      // First we copy the elements in front of pos into the new
          array.
      for(i = 0; i < p.second; ++i) {
        new_array_p[i] = (*this).array_p[i];
      }
      // Then we insert the new element and copy the rest of the
      // elements into the new array.

      retval = new_array_p[i] = element;

      (*element).p.first = (*this).hld;
      (*element).p.second = i;
      (*element).deleted = false;
```

```
        while(i <= (*this).current_size) {
          new_array_p[i+1] = (*this).array_p[i];
          (*new_array_p[i+1]).update(i+1);
          ++i;
        }

        // The last thing to do is to deallocate the old array
        // and swap the pointers.
        (*this).array_allocator.deallocate((*this).array_p, (*this).
            current_capacity);
        (*this).current_capacity = new_capacity;
        (*this).array_p = new_array_p;
      }
      else {

        size_type i;

        // In this case we move all the elements from pos and forward
        // one position to the right.
        for(i = (*this).current_size+1; i > p.second; --i) {
          (*this).array_p[i] = (*this).array_p[i-1];
          (*(*this).array_p[i]).update(i);
        }

        retval = (*this).array_p[i] = element;

        (*element).p.first = (*this).hld;
        (*element).p.second = i;
        (*element).deleted = false;
      }

      (*this).current_size += 1;
      assert((*this).current_size <= (*this).current_capacity);

      // we will return the entry which represents the new element
      //return concrete_iterator(retval);
    }
  };
}


#include "iv_dynamic_array_entry_rc.h++"
```

Appendix A.8 iv_dynamic_array_entry_rc.h++

```
namespace cphstl {

  template<typename V, typename A, typename E>
  class persistent_dynamic_array;

  template <typename V, typename A>
  class iv_dynamic_array_entry_rc {
  public:
      typedef std::size_t size_type;
      typedef V value_type;
      typedef holder<V, A, iv_dynamic_array_entry_rc> holder_type;
      typedef std::pair<holder_type*, size_type> position_type;
      typedef typename A::template rebind<size_type>::other
          size_type_allocator;
      typedef typename A::template rebind<iv_dynamic_array_entry_rc>::
          other entry_allocator;
      typedef std::ptrdiff_t difference_type;

    protected: /* should be protected */
```

```
  value_type v;
  position_type p;
  size_type* references;
  bool deleted;
  friend class persistent_dynamic_array<V, A,
      iv_dynamic_array_entry_rc<V, A> >;
  friend class iv_dynamic_array_rc< V, A,
      iv_dynamic_array_entry_rc<V,A> >;
  friend class iv_dynamic_array< V, A, iv_dynamic_array_entry_rc<V
      ,A> >;

public:
  ~iv_dynamic_array_entry_rc() {
  }
  iv_dynamic_array_entry_rc(value_type const& _v, position_type _p
      ) {
    (*this).v = _v;
    (*this).p = _p;
    size_type_allocator sta;
    (*this).references = sta.allocate(1);
    *((*this).references) = 0;
    (*this).deleted = false;
  }
  value_type& operator*() {
    return v;
  }
  void update(size_type n) {
    p.second = n;
  }
  value_type& content() {
    return v;
  }
  value_type const& content() const {
    return v;
  }
  void attach() const {
    ++(*(*this).references);
  }

  void detach() const {
    assert(*(*this).references > 0);
    --(*(*this).references);

    if(*(*this).references == 0 && (*this).deleted) {
      (*this).~iv_dynamic_array_entry_rc();
      entry_allocator a;
      a.deallocate((iv_dynamic_array_entry_rc*) this, 1);
    }
  }

  iv_dynamic_array_entry_rc* advance(const typename
      iv_dynamic_array<V,A>::difference_type n) const {
    iv_dynamic_array_entry_rc* tmp = (*(*(*this).p.first).ptr).
        get_entry((*this).p.second + n);
    (*this).detach();
    (*tmp).attach();
    return tmp;
  }

  difference_type distance(iv_dynamic_array_entry_rc const* a)
      const {
    return p.second - (*a).p.second;
  }

  bool equal(iv_dynamic_array_entry_rc const* a) const {
```

```
        return (p.first == (*a).p.first) && (p.second == (*a).p.second
            );
      }
    };

}
```

Appendix A.9 ps_vector.h++

```
#include "vector.h++"

/*
   Bridge class for the pointer semantics extention to vector. Which
       means this is possible:

      cont v1;
      ...
      it1 = v.insert(v1.begin(), 27);
      v1.erase(it1);
      cont v2;
      v2.ps_insert(v2.begin(), it1);

   Without additional memory allocation, or iterators gets invalidated
       .

   Author: Bo Simonsen, November 2008
*/

namespace cphstl {
  template <
    typename V,
    typename A = std::allocator<V>,
    typename R = std::vector<V, A>,
    typename I = typename R::iterator,
    typename J = typename R::const_iterator
  >
  class ps_vector : public vector<V, A, R, I, J> {
  public:
    typedef I iterator;
    typedef J const_iterator;
    void ps_insert(iterator pos, iterator element) {
      (*this).kernel.ps_insert(pos, element);
    }

  };
}
```

Appendix A.10 avl.h++

```
/*

  Implementation of a persistent AVL tree.

  Author: Bo Simonsen, November 2008

*/

#include <memory>
#include <functional>
#include <iostream>
#include <cassert>
#include <stack>
#include <deque>
#include <string> // defines std::string
```

```
#include "avl_node.h++"
#include "set_helper.c++"

namespace cphstl {
  template <typename K,
            typename V = K,
            typename F = cphstl::unnamed::identity<V>,
            typename C = std::less<K>,
            typename A = std::allocator<V>,
            typename N = persistent_avl_node<V> >
  class persistent_avl_tree {
  public:
    typedef N node_type;
    typedef V value_type;
    typedef typename A::template rebind<node_type>::other
        allocator_type;
    typedef C comparator_type;
    typedef std::pair<node_type*, std::deque<node_type*> >
        concrete_iterator;
    typedef std::pair<node_type const*, std::deque<node_type const*> >
         const_concrete_iterator;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    typedef value_type* pointer;
    typedef value_type const* const_pointer;
    typedef value_type& reference;
    typedef value_type const& const_reference;
  private:
    allocator_type allocator;
    comparator_type comparator;
    enum {revisions = 20};
    node_type* root[revisions];
    size_type node_count[revisions];
    unsigned int current_revision;

  protected:

    /* internal structure helpers */
    int height(node_type* node, int count) const;
    int balance(node_type* node) const;
    node_type* rebalance(node_type* parent, node_type* node);
    node_type* rotate_right(node_type* q, node_type* p);
    node_type* rotate_left(node_type* p, node_type* q);

    /* Insert helper method */
    concrete_iterator _insert(node_type* parent, node_type* node,
        value_type const& v);

    /* Erase helper methods */
    void _erase(node_type* parent, node_type* node);
    void _erase(node_type* node, node_type* parent, value_type const&
        v);

    /* destructor helper */
    void destroy_tree(node_type*, size_type, std::stack<node_type*>&);

  public:
    ~persistent_avl_tree();
    persistent_avl_tree(persistent_avl_tree const& x);
    explicit persistent_avl_tree(C const& c, A const& a);

    allocator_type get_allocator() const;
    comparator_type key_comp() const;
```

```
    comparator_type value_comp() const;

    void swap(persistent_avl_tree& x);

    std::pair<concrete_iterator, bool> insert(value_type const& v);

    template <typename I>
    void insert(I x, I y);

    size_type erase(value_type const& v);

    concrete_iterator begin(int revision);
    concrete_iterator end(int revision);
    const_concrete_iterator begin(int revision) const;
    const_concrete_iterator end(int revision) const;

    size_type size(int revision) const;
    size_type max_size() const;

    int revision();

    std::string dump_tree(node_type* node, int d = 0) const;
    operator std::string() const;
  };

}
#include "avl.c++"
```

Appendix A.11 avl.c++

```
/*

   A persistent version of the AVL tree, should be used with the
       persistent_set bridge class
   This version is based on path copying.

   Limitations:

     - No support for insert(iterator, value), erase(iterator) yet
     - No support for multiset

   Author: Bo Simonsen, November 2008

*/

namespace cphstl {
  template <typename K, typename V, typename F, typename C, typename A
      , typename N>
  int
  persistent_avl_tree<K, V, F, C, A, N>::height(node_type* node, int
      count) const {
    if(node == 0)
      return count;

    int hl = (*this).height((*node).left, count+1);
    int hr = (*this).height((*node).right, count+1);

    if(hr > hl)
      return hr;

    return hl;
  }

  template <typename K, typename V, typename F, typename C, typename A
      , typename N>
```

```
int
persistent_avl_tree<K, V, F, C, A, N>::balance(node_type* node)
    const {
  return (*this).height((*node).right, 0) - (*this).height((*node).
      left, 0);
}

template <typename K, typename V, typename F, typename C, typename A
    , typename N>
typename persistent_avl_tree<K, V, F, C, A, N>::node_type*
persistent_avl_tree<K, V, F, C, A, N>::rebalance(node_type* parent,
    node_type* node) {

  node_type* tmp = 0;

  if((*this).balance(node) > 1) {
    /* right-right */
    if((*this).balance((*node).right) >= 1) {
      if((*(*node).right).revision != current_revision) {
        node_type* tnoderight = allocator.allocate(1);
        new (tnoderight) node_type(*(*node).right, current_revision)
            ;
        (*node).right = tnoderight;
      }
      tmp = (*this).rotate_left(node, (*node).right);
    }
    /* right-left */
    else {
      if((*(*node).right).revision != current_revision) {
        node_type* tnoderight = allocator.allocate(1);
        new (tnoderight) node_type(*(*node).right, current_revision)
            ;
        (*node).right = tnoderight;
      }

      if((*(*(*node).right).left).revision != current_revision) {
        node_type* tnoderightleft = allocator.allocate(1);
        new (tnoderightleft) node_type(*(*(*node).right).left,
            current_revision);
        (*(*node).right).left = tnoderightleft;
      }

      (*node).right = (*this).rotate_right((*node).right, (*(*node).
          right).left);
      tmp = (*this).rotate_left(node, (*node).right);
    }

  }
  else if((*this).balance(node) < -1) {
    /* left-left */
    if((*this).balance((*node).left) <= -1) {
      if((*(*node).left).revision != current_revision) {
        node_type* tnodeleft = allocator.allocate(1);
        new (tnodeleft) node_type(*(*node).left, current_revision);
        (*node).left = tnodeleft;
      }

      tmp = (*this).rotate_right(node, (*node).left);
    }
    /* left-right */
    else {
      if((*(*node).left).revision != current_revision) {
        node_type* tnodeleft = allocator.allocate(1);
        new (tnodeleft) node_type(*(*node).left, current_revision);
        (*node).left = tnodeleft;
```

```
      }

      if((*(*(*node).left).right).revision != current_revision) {
        node_type* tnodeleftright = allocator.allocate(1);
        new (tnodeleftright) node_type(*(*(*node).left).right,
            current_revision);
        (*(*node).left).right = tnodeleftright;
      }

      (*node).left = (*this).rotate_left((*node).left, (*(*node).
          left).right);
      tmp = (*this).rotate_right(node, (*node).left);
    }
  }

  /* if a rotation occur */
  if(tmp != 0) {

    if(parent != 0) {
      if((*parent).left == node) {
        (*parent).left = tmp;
      }
      else if((*parent).right == node) {
        (*parent).right = tmp;
      }
      else {
        assert(0);
      }
    }
    else {
      root[current_revision] = tmp;
    }

  }

  return tmp;
}

template <typename K, typename V, typename F, typename C, typename A
    , typename N>
typename persistent_avl_tree<K, V, F, C, A, N>::node_type*
persistent_avl_tree<K, V, F, C, A, N>::rotate_right(node_type* q,
    node_type* p) {
  assert(p != 0 && q != 0);

  node_type* a = (*p).left;
  node_type* b = (*p).right;
  node_type* c = (*q).right;

  (*p).right = q;
  (*p).left = a;
  (*q).left = b;
  (*q).right = c;

  return p;
}

template <typename K, typename V, typename F, typename C, typename A
    , typename N>
typename persistent_avl_tree<K, V, F, C, A, N>::node_type*
persistent_avl_tree<K, V, F, C, A, N>::rotate_left(node_type* p,
    node_type* q) {
  assert(p != 0 && q != 0);

  node_type* a = (*p).left;
```

```
  node_type* b = (*q).left;
  node_type* c = (*q).right;

  (*q).left = p;
  (*p).left = a;
  (*p).right = b;
  (*q).right = c;

  return q;

}

template <typename K, typename V, typename F, typename C, typename A
    , typename N>
typename persistent_avl_tree<K, V, F, C, A, N>::concrete_iterator
persistent_avl_tree<K, V, F, C, A, N>::_insert(node_type* parent,
    node_type* node, value_type const& v) {
  concrete_iterator retval;

  if(node == 0) {
    if((*parent).sentinel || comparator(v, (*parent).data)) {
      (*parent).left = allocator.allocate(1);
      node = (*parent).left;
    } else {
      (*parent).right = allocator.allocate(1);
      node = (*parent).right;
    }
    new (node) N(v, current_revision);
    return concrete_iterator(node, std::deque<node_type*>());
  } else {
    if((*node).sentinel || comparator(v, (*node).data)) {
      if((*node).left) {
        node_type* tnodeleft = allocator.allocate(1);
        new (tnodeleft) node_type(*(*node).left, current_revision);
        (*node).left = tnodeleft;
      }
      retval = _insert(node, (*node).left, v);
    }
    else {
      if((*node).right) {
        node_type* tnoderight = allocator.allocate(1);
        new (tnoderight) node_type(*(*node).right, current_revision)
            ;
        (*node).right = tnoderight;
      }
      retval = _insert(node, (*node).right, v);
    }
    retval.second.push_front(rebalance(parent, node));
    return retval;
  }
}

template <typename K, typename V, typename F, typename C, typename A
    , typename N>
void
persistent_avl_tree<K, V, F, C, A, N>::_erase(node_type* parent,
    node_type* node) {
  std::stack<node_type*> s;

  /* O.K. */
  if((*node).left == 0 && (*node).right == 0) {
    if((*parent).left == node) {
      (*parent).left = 0;
    }
    else {
```

```
    (*parent).right = 0;
  }
}
else if((*node).left == 0) {
  if((*parent).left == node) {
    (*parent).left = (*node).right;
  }
  else {
    (*parent).right = (*node).right;
  }
}
/* O.K. */
else if((*node).right == 0) {
  if((*parent).left == node) {
    (*parent).left = (*node).left;
  }
  else {
    (*parent).right = (*node).left;
  }
}
else {
  node_type* tnode = 0;

  /*
   *            X  <--- to delete
   *          /   \
   *         Y   /_\
   *       /  \
   *      /_\  ..
   *           Z  <-- root candidate
   *
   */

  if((*(*node).left).right != 0) {

    node_type* parent_tmp = node;
    node_type* tmp;

    tmp = allocator.allocate(1);
    new (tmp) node_type(*(*node).left, current_revision);
    (*node).left = tmp;

    s.push(tmp);

    while((*tmp).right != 0) {
      parent_tmp = tmp;
      node_type* _tmp = allocator.allocate(1);
      new (_tmp) node_type(*(*tmp).right, current_revision);
      tmp = (*tmp).right = _tmp;
      s.push(tmp);
    }

    tnode = allocator.allocate(1);
    new (tnode) node_type(*tmp, current_revision);

    (*tnode).left = (*node).left;
    (*tnode).right = (*node).right;

    (*parent_tmp).right = (*tmp).left;
  }

  /*
   *       X  <-- to delete
   *     /   \
   *    Y <+ /_\
```

```
       *   /   |
       *  Z    +-- root candidate
       *
       */

      else {
        tnode = allocator.allocate(1);
        new (tnode) node_type(*(*node).left, current_revision);
        (*tnode).right = (*node).right;
      }

      /* Where to attach the newly allocated node */
      if(parent != 0) {
        if((*parent).left == node) {
          (*parent).left = tnode;
        } else {
          (*parent).right = tnode;
        }

      }
      else {
        root[current_revision] = tnode;
      }

      rebalance(parent, tnode);

      /* we should rebalance parent pointers on the stack */
      if(!s.empty()) {

        node_type* tmp = s.top();
        s.pop();

        while(!s.empty()) {
          std::cout << "Once" << std::endl;
          node_type* tmp_ = s.top();
          s.pop();
          rebalance(tmp_, tmp);
          tmp = tmp_;
        }
      }
    }

}

template <typename K, typename V, typename F, typename C, typename A
    , typename N>
void
persistent_avl_tree<K, V, F, C, A, N>::_erase(node_type* node,
    node_type* parent, value_type const& v) {
  if(!comparator((*node).data, v) && !comparator(v, (*node).data)) {
    _erase(parent, node);
    /* we allocated one node to much */
    allocator.deallocate(node, 1);
    return;
  }

  if(comparator(v, (*node).data)) {
    node_type* tnodeleft = allocator.allocate(1);
    new (tnodeleft) node_type(*(*node).left, current_revision);
    (*node).left = tnodeleft;
    _erase((*node).left, node, v);
  }
  else {
    node_type* tnoderight = allocator.allocate(1);
    new (tnoderight) node_type(*(*node).right, current_revision);
```

```
    (*node).right = tnoderight;
    _erase((*node).right, node, v);
  }

  /* The tree could be become unbalanced from the deleted node to
      parent */
  rebalance(parent, node);
}

template <typename K, typename V, typename F, typename C, typename A
    , typename N>
persistent_avl_tree<K, V, F, C, A, N>::persistent_avl_tree(C const&
    c, A const& a) : allocator(a), comparator(c) {
  root[0] = allocator.allocate(1);
  /* sentinel */
  new (root[0]) node_type(V(), 0);
  (*root[0]).sentinel = true;

  node_count[0] = 0;
  current_revision = 0;
}

template <typename K, typename V, typename F, typename C, typename A
    , typename N>
persistent_avl_tree<K, V, F, C, A, N>::persistent_avl_tree(
    persistent_avl_tree<K, V, F, C, A, N> const& x) {
  /* no implemented yet */
}


template <typename K, typename V, typename F, typename C, typename A
    , typename N>
void
persistent_avl_tree<K, V, F, C, A, N>::destroy_tree(node_type* n,
    size_type revision, std::stack<node_type*>& s) {
  if(n == 0) {
    return;
  }

  if((*n).revision == revision) {
    (*this).destroy_tree((*n).left, revision, s);
    s.push(n);
    (*this).destroy_tree((*n).right, revision, s);
  }


}

template <typename K, typename V, typename F, typename C, typename A
    , typename N>
persistent_avl_tree<K, V, F, C, A, N>::~persistent_avl_tree() {
  std::stack<node_type*> s;

  for(int i=0; i <= current_revision; ++i) {
    (*this).destroy_tree(root[i], i, s);
  }

  while(!s.empty()) {
    node_type* n = s.top();
    (*n).~node_type();
    allocator.deallocate(n, 1);
    s.pop();
  }
}
```

```
template <typename K, typename V, typename F, typename C, typename A
    , typename N>
std::pair<typename persistent_avl_tree<K, V, F, C, A, N>::
    concrete_iterator, bool>
persistent_avl_tree<K, V, F, C, A, N>::insert(value_type const& v) {
  try {
    ++current_revision;
    root[current_revision] = allocator.allocate(1);
    new (root[current_revision]) node_type(*root[current_revision
        -1], current_revision);
    node_count[current_revision] = node_count[current_revision-1] +
        1;
    return std::pair<concrete_iterator, bool>(_insert(NULL, root[
        current_revision], v), true);
  }
  catch(...) {
    --current_revision;
    /* we should clean up memory */
    throw;
  }
}

template <typename K, typename V, typename F, typename C, typename A
    , typename N>
template <typename I>
void
persistent_avl_tree<K, V, F, C, A, N>::insert(I x, I y) {
  while(x != y) {
    (*this).insert(*x);
    ++x;
  }
}

template <typename K, typename V, typename F, typename C, typename A
    , typename N>
typename persistent_avl_tree<K, V, F, C, A, N>::size_type
persistent_avl_tree<K, V, F, C, A, N>::erase(value_type const& v) {
  try {
    ++current_revision;
    root[current_revision] = allocator.allocate(1);
    new (root[current_revision]) node_type(*root[current_revision
        -1], current_revision);
    _erase(root[current_revision], 0, v);
    node_count[current_revision] = node_count[current_revision-1] -
        1;
    /* only works for set */
    return 1;
  }
  catch (...) {
    --current_revision;
  }
}

template <typename K, typename V, typename F, typename C, typename A
    , typename N>
typename persistent_avl_tree<K, V, F, C, A, N>::concrete_iterator
persistent_avl_tree<K, V, F, C, A, N>::begin(int revision) {
  std::deque<node_type*> s;

  node_type* node = root[revision];
  s.push_front(node);
  while((*node).left) {
    node = (*node).left;
    s.push_front(node);
```

```
  }
  s.pop_front();

  return concrete_iterator(node, s);
}
template <typename K, typename V, typename F, typename C, typename A
    , typename N>
typename persistent_avl_tree<K, V, F, C, A, N>::concrete_iterator
persistent_avl_tree<K, V, F, C, A, N>::end(int revision) {
  std::deque<node_type*> s;

  node_type* node = root[revision];
  s.push_front(node);
  while((*node).right) {
    node = (*node).right;
    s.push_front(node);
  }
  s.pop_front();

  return concrete_iterator(node, s);
}

template <typename K, typename V, typename F, typename C, typename A
    , typename N>
typename persistent_avl_tree<K, V, F, C, A, N>::size_type
persistent_avl_tree<K, V, F, C, A, N>::size(int revision) const {
  return node_count[revision];
}

template <typename K, typename V, typename F, typename C, typename A
    , typename N>
typename persistent_avl_tree<K, V, F, C, A, N>::size_type
persistent_avl_tree<K, V, F, C, A, N>::max_size() const {
  return allocator.max_size();
}

template <typename K, typename V, typename F, typename C, typename A
    , typename N>
typename persistent_avl_tree<K, V, F, C, A, N>::allocator_type
persistent_avl_tree<K, V, F, C, A, N>::get_allocator() const {
  return allocator_type();
}

template <typename K, typename V, typename F, typename C, typename A
    , typename N>
typename persistent_avl_tree<K, V, F, C, A, N>::comparator_type
persistent_avl_tree<K, V, F, C, A, N>::key_comp() const {
  return comparator_type();
}

template <typename K, typename V, typename F, typename C, typename A
    , typename N>
typename persistent_avl_tree<K, V, F, C, A, N>::comparator_type
persistent_avl_tree<K, V, F, C, A, N>::value_comp() const {
  return comparator_type();
}



template <typename K, typename V, typename F, typename C, typename A
    , typename N>
int
persistent_avl_tree<K, V, F, C, A, N>::revision() {
  return current_revision;
}
```

```
  template <typename K, typename V, typename F, typename C, typename A
      , typename N>
  std::string
  persistent_avl_tree<K, V, F, C, A, N>::dump_tree(node_type* node,
      int d) const {
    std::string ret, data, depth, bal;
    std::stringstream ss, ss_, ss__;

    if(node == 0)
      return std::string();

    ret += dump_tree((*node).left, d+1);

    ss << (*node).data;
    ss >> data;
    ss_ << d;
    ss_ >> depth;
    ss__ << (*this).balance(node);
    ss__ >> bal;

    ret += std::string("D: ") + depth + std::string(" Elem: ") + data
        + std::string(" Balance ") +
           bal + std::string("\n");
    ret += dump_tree((*node).right, d+1);

    return ret;
  }

  template <typename K, typename V, typename F, typename C, typename A
      , typename N>
  persistent_avl_tree<K, V, F, C, A, N>::operator std::string() const
      {
    return dump_tree(root[current_revision]);
  }

  template <typename K, typename V, typename F, typename C, typename A
      , typename N>
  void
  persistent_avl_tree<K, V, F, C, A, N>::swap(persistent_avl_tree<K, V
      , F, C, A, N>& x) {
    std::swap(root, x.root);
    std::swap(node_count, x.node_count);
    std::swap(current_revision, x.current_revision);
    std::swap(allocator, x.allocator);
    std::swap(comparator, x.comparator);
  }
}
```

Appendix A.12 avl_node.h++

```
namespace cphstl {

template <typename V>
class persistent_avl_node {
public:
  typedef V value_type;
  typedef size_t size_type;

  persistent_avl_node* left;
  persistent_avl_node* right;
  value_type data;
  size_type revision;
  bool sentinel;
```

```
persistent_avl_node() {}

persistent_avl_node(value_type const& _data, size_type const& rev) {
  (*this).left = 0;
  (*this).right = 0;
  (*this).data = _data;
  (*this).revision = rev;
  (*this).sentinel = false;
}

persistent_avl_node(persistent_avl_node& a, size_type const& rev) {
  (*this).left = a.left;
  (*this).right = a.right;
  (*this).data = a.data;
  (*this).sentinel = a.sentinel;
  (*this).revision = rev;
}

value_type& content() {
  return data;
}
const value_type& content() const {
  return data;
}

persistent_avl_node* successor(std::deque<persistent_avl_node*>& s)
    {

  persistent_avl_node* x = this;

  if((*x).right) {
    s.push_front(x);
    x = (*x).right;
    while((*x).left) {
      s.push_front(x);
      x = (*x).left;
    }
    return x;
  }
  else {
    persistent_avl_node* y = s.front();
    s.pop_front();
    while(s.size() > 0 && x == (*y).right) {
      x = y;
      y = s.front();
      s.pop_front();
    }
    return y;
  }
}

persistent_avl_node* predecessor(std::deque<persistent_avl_node*>& s
    ) {
  persistent_avl_node* x = this;

  if((*x).left) {
    s.push_front(x);
    x = (*x).left;
    while((*x).right) {
      s.push_front(x);
      x = (*x).right;
    }
    return x;
  }
  else {
```

```
      persistent_avl_node* y = s.front();
      s.pop_front();
      while(s.size() > 0 && x == (*y).left) {
        x = y;
        y = s.front();
        s.pop_front();
      }
      return y;
    }
  }
};
}
```

Appendix A.13 stl_persistent_set.h++

```
/*
  Implementation of cphstl::persistent_set

  The persistent set class is an extention of the set bridge class.
  With partial persistent data structures, can we get iterators for
  any revision, at any time, which is what this bridge class supports
  using the persistent_set_adapter which is instatiated when the []
      operator
  is used, such that we can do this:

    persistent_set<....> s;
    ...
    // revision 2 and 5 start iterator similar for end.
    it = s[2].begin();
    it = s[5].begin();
    // current / lastest revision
    it = s.begin();

  Author: Bo Simonsen, November 2008
*/



#include "stl_set.h++"

namespace cphstl {

  template <typename R, typename I, typename J>
  class persistent_set_adapter {
  public:
    typedef std::size_t size_type;
    typedef R realization_type;
    typedef I iterator;
    typedef J const_iterator;
  private:
    realization_type* link;
    size_type revision;
  public:
    /* pointer to avoid copy construction */
    explicit persistent_set_adapter(realization_type* r, size_type rev
        ) {
      link = r;
      revision = rev;
    }
    size_type size() {
      return (*link).size(revision);
    }
    iterator begin() {
      return (*link).begin(revision);
    }
```

```
    iterator end() {
      return (*link).end(revision);
    }
    const_iterator begin() const {
      return (*link).begin(revision);
    }
    const_iterator end() const {
      return (*link).end(revision);
    }

  };

  template <
    typename V,
    typename C = std::less<V>,
    typename A = std::allocator<V>,
    typename R = std::set<V, C, A>,
    typename I = typename R::iterator,
    typename J = typename R::const_iterator
  >
  class persistent_set : public set<V, C, A, R, I, J> {
  public:
    typedef set<V, C, A, R, I, J> superclass;
    typedef std::size_t size_type;
    typedef I iterator;
    typedef J const_iterator;

    size_type size() {
      return (*this).kernel.size((*this).kernel.revision());
    }

    iterator begin() {
      return (*this).kernel.begin((*this).kernel.revision());
    }
    iterator end() {
      return (*this).kernel.end((*this).kernel.revision());
    }
    const_iterator begin() const {
      return (*this).kernel.begin((*this).kernel.revision());
    }
    const_iterator end() const {
      return (*this).kernel.end((*this).kernel.revision());
    }
    persistent_set_adapter<R, I, J> operator[](size_type revision) {
      return persistent_set_adapter<R, I, J>(&((*this).kernel),
          revision);
    }

  };
}
```

Appendix A.14 test_avl.c++

```
#include "stack_iterator.h++"
#include "avl.h++"
#include "stl_persistent_set.h++"

int main() {
  typedef int V;
  typedef std::less<V> C;
  typedef std::allocator<V> A;
  typedef cphstl::persistent_avl_tree<V> R;
  typedef cphstl::persistent_set<V, C, A, R, cphstl::stack_iterator<R
      ::node_type, false>,
          cphstl::stack_iterator<R::node_type, true> > abstraction;
```

```
  abstraction s;
  typedef abstraction::iterator si;
  s.insert(1); //
  s.insert(5); //
  s.insert(3); //
  s.insert(15);
  s.insert(7);
  s.insert(9); //
  s.insert(10);
  s.insert(35);
  s.insert(2); //
  s.insert(25);
  s.erase(7);
  s.erase(5);
  s.erase(15);
  s.erase(3);
  s.erase(9);
  si it;
  for(it = s.begin(); it != s.end(); ++it) {
    std::cout << "It:␣" << *it << std::endl;
    std::cout << it << std::endl;

  }
  std::cout << *it << std::endl;
  std::cout << it << std::endl;

  std::cout << "Backwards" << std::endl;

  while(it != s.begin()) {
    std::cout << "It:␣" << *it << std::endl;
    std::cout << it << std::endl;
    it--;
  }

  std::cout << "Size:␣" << s.size() << std::endl;
  std::cout << "Size␣(rev:␣2):␣" << s[2].size() << std::endl;
  std::cout << "Size␣(rev:␣12):␣" << s[12].size() << std::endl;

  std::cout << "Rev␣2:␣" << std::endl;
  for(it = s[3].begin(); it != s[3].end(); ++it) {

    std::cout << "It:␣" << *it << std::endl;
  }

  std::cout << std::string(s);
}
```

Appendix A.15 stack_iterator.h++

```
/*
  The idea of combining iterators and const iterators into the same
  class is taken from [Matt Austern. Defining iterators and const
  iterators. C/C++ User's Journal 19,1 (2001), 74-79].

  Author: Jyrki Katajainen, Bo Simonsen, May 2006, April 2008

  General design idea is proposed by J. Katajainen and B. Simonsen
*/

#ifndef __CPHSTL_STACK_ITERATOR__
#define __CPHSTL_STACK_ITERATOR__

#include <iterator> // defines std::bidirectional_iterator_tag
#include <cstddef> // defines std::ptrdiff_t
```

```
#include <iostream> // defines std::ostream
#include <string> // defines std::string
#include <stack>

#include "type.h++" // defines cphstl::if_then_else

namespace cphstl {

  /* Forward declarations of bridge classes */
  template <typename V, typename C, typename A, typename R, typename I
      , typename J>
  class persistent_set;
  template <typename R, typename I, typename J>
  class persistent_set_adapter;

  template <typename N, bool is_const = false>
  class stack_iterator {

  public:

    // types

    typedef std::bidirectional_iterator_tag iterator_category;
    typedef typename N::value_type value_type;
    typedef std::ptrdiff_t difference_type;

    typedef typename if_then_else<is_const, value_type const*,
        value_type*>::type pointer;
    typedef typename if_then_else<is_const, value_type const&,
        value_type&>::type reference;

    typedef stack_iterator<N, !is_const> complement;
  private:

    // types
    typedef typename if_then_else<is_const, N const*, N*>::type
        node_pointer;
    typedef std::pair<node_pointer, std::deque<node_pointer> >
        concrete_iterator;

  public:

    // friends
    friend class stack_iterator<N, !is_const>;

    template <typename M, bool both>
    friend std::ostream& operator<<(std::ostream&, stack_iterator<M,
        both> const&);

    template <typename V, typename C, typename A, typename R, typename
         I, typename J>
    friend class cphstl::persistent_set;
    template <typename R, typename I, typename J>
    friend class cphstl::persistent_set_adapter;

    template <typename T1, typename T2>
    friend class std::pair;

    // structors

    stack_iterator();
    stack_iterator(stack_iterator<N, false> const&);
    stack_iterator& operator=(stack_iterator const&);
    ~stack_iterator();
```

```
    // operators

    reference operator*() const;
    pointer operator->() const;
    stack_iterator& operator++();
    stack_iterator operator++(int);
    stack_iterator& operator--();
    stack_iterator operator--(int);

    template <bool both>
    bool operator==(stack_iterator<N, both> const&) const;

    template <bool both>
    bool operator!=(stack_iterator<N, both> const&) const;

  private:
    // converters to be used by the friends

    stack_iterator(concrete_iterator); // node_pointer --> iterator
    operator concrete_iterator() const; // iterator --> node_pointer
    operator std::string() const; // iterator --> string

    concrete_iterator link;

  };

}

#include "stack_iterator.c++" // implements cphstl::stack_iterator

#endif
```

Appendix A.16 stack_iterator.c++

```
/*
  Implementation of cphstl::stack_iterator

  Author: Jyrki Katajainen, Bo Simonsen, April 2008

  General design idea is proposed by J. Katajainen and B. Simonsen
*/

#include <sstream> // defines std::stringstream

namespace cphstl {

  // default constructor

  template <typename N, bool is_const>
  stack_iterator<N, is_const>::stack_iterator()
    : link(concrete_iterator()) {
  }

  // copy constructor

  template <typename N, bool is_const>
  stack_iterator<N, is_const>::stack_iterator(stack_iterator<N, false>
       const& a)
    : link(a.link) {
  }

  // assignment

  template <typename N, bool is_const>
  stack_iterator<N, is_const>&
```

```
stack_iterator<N, is_const>::operator=(stack_iterator<N, is_const>
    const& a) {
  link = a.link;
  return *this;
}

// destructor

template <typename N, bool is_const>
stack_iterator<N, is_const>::~stack_iterator() {
}

// operator*

template <typename N, bool is_const>
typename stack_iterator<N, is_const>::reference
stack_iterator<N, is_const>::operator*() const {
  return reference((*link.first).content());
}

// operator->

template <typename N, bool is_const>
typename stack_iterator<N, is_const>::pointer
stack_iterator<N, is_const>::operator->() const {
  return pointer(&(*link.first).content());
}

// operator++; pre-increment

template <typename N, bool is_const>
stack_iterator<N, is_const>&
stack_iterator<N, is_const>::operator++() {
  (*this).link.first = (*(*this).link.first).successor((*this).link.
      second);
  return *this;
}

 // operator++; post-increment

template <typename N, bool is_const>
stack_iterator<N, is_const>
stack_iterator<N, is_const>::operator++(int) {
  stack_iterator<N, is_const> temporary(*this);
  ++(*this);
  return temporary;
}

// operator--; pre-increment

template <typename N, bool is_const>
stack_iterator<N, is_const>&
stack_iterator<N, is_const>::operator--() {
  (*this).link.first = (*(*this).link.first).predecessor((*this).
      link.second);
  return *this;
}

// operator--; post-increment

template <typename N, bool is_const>
stack_iterator<N, is_const>
stack_iterator<N, is_const>::operator--(int) {
  stack_iterator<N, is_const> temporary(*this);
  --(*this);
```

```
  return temporary;
}

// operator==

template <typename N, bool is_const>
template <bool both>
bool
stack_iterator<N, is_const>::operator==(stack_iterator<N, both>
    const& a) const {
  return link.first == a.link.first;
}

// operator!=

template <typename N, bool is_const>
template <bool both>
bool
stack_iterator<N, is_const>::operator!=(stack_iterator<N, both>
    const& a) const {
  return !((*this) == a);
}

// parametrized constructor (node -> iterator)

template <typename N, bool both>
stack_iterator<N, both>::stack_iterator(concrete_iterator p)
  : link(p) {
}

// conversion operators (iterator -> node)

template <typename N, bool is_const>
stack_iterator<N, is_const>::operator concrete_iterator() const {
  return link;
}

// conversion operator (makes it possible to print out an iterator)

template <typename N, bool is_const>
stack_iterator<N, is_const>::operator std::string() const {
  std::stringstream ss, ss_;
  std::string address, size;
  ss << (int)(char*)((*this).link.first);
  ss >> address;
  ss_ << (int)(char*) link.second.size();
  ss_ >> size;

  if (is_const == false) {
    return std::string("iterator: node at ") + address + std::string
        (" stack size: ") + size;
  }
  else {
    return std::string("const_iterator: node at ") + address + std::
        string(" stack size: ") + size;
  }

}

// representation

template <typename N, bool both>
std::ostream&
operator<<(std::ostream& s, stack_iterator<N, both> const& i) {
  s  << std::string(i);
```

```
    return s;
  }

}
```

Appendix A.17 pers_dynamic_array.h++

```
/*
  This extention adds supports reference counting, which should be
      used with the iv_dynamic_entry_rc
  class.

  Authors: Bo Simonsen, November 2008
*/

#include "stack_iterator.h++"
#include "avl.h++"
#include "stl_persistent_set.h++"
#include "iv_dynamic_array.h++"

namespace cphstl {
  template <typename E>
  class my_cmp {
  public:
    bool operator()(E* const& t1, E* const& t2) {
      return (*t1).distance(t2) < 0;
    }
  };

  /* bad solution to copy paste, but since types are defined
      explicitly (and
   * they need too) there is not much else to do.. */

  template <typename E, typename V>
  class my_node {
  public:
    /* The value type which the iterator should return */
    typedef V value_type;
    /* The actual type we hold */
    typedef E entry_type;
    typedef size_t size_type;

    my_node* left;
    my_node* right;
    entry_type data;
    size_type revision;
    bool sentinel;

    my_node() {}

    ~my_node() {
    }

    my_node(entry_type const& _data, size_type const& rev) {
      (*this).left = 0;
      (*this).right = 0;
      (*this).data = _data;
      (*this).revision = rev;
      (*this).sentinel = false;
    }

    my_node(my_node& a, size_type const& rev) {
      (*this).left = a.left;
      (*this).right = a.right;
      (*this).data = a.data;
```

```
    (*this).revision = rev;
    (*this).sentinel = a.sentinel;
  }

  value_type& content() {
    return (*(*this).data).content();
  }
  const value_type& content() const {
    return (*(*this).data).content();
  }

  my_node* successor(std::deque<my_node*>& s) {

    my_node* x = this;

    if((*x).right) {
      s.push_front(x);
      x = (*x).right;
      while((*x).left) {
              s.push_front(x);
        x = (*x).left;
      }
      return x;
    }
    else {
      my_node* y = s.front();
      s.pop_front();
      while(s.size() > 0 && x == (*y).right) {
        x = y;
        y = s.front();
        s.pop_front();
      }
      return y;
    }
  }

  my_node* predecessor(std::deque<my_node*>& s) {
    my_node* x = this;

    if((*x).left) {
      s.push_front(x);
      x = (*x).left;
      while((*x).right) {
              s.push_front(x);
        x = (*x).right;
      }
      return x;
    }
    else {
      my_node* y = s.front();
      s.pop_front();
      while(s.size() > 0 && x == (*y).left) {
        x = y;
        y = s.front();
        s.pop_front();
      }
      return y;
    }
  }
};


template <
  typename V,
  typename A = std::allocator<V>,
```

```
    typename E = iv_dynamic_array_entry<V, A>
  >
  class persistent_dynamic_array : public iv_dynamic_array<V, A, E> {
  public:

    explicit persistent_dynamic_array(A const& a = A()) :
        iv_dynamic_array<V,A,E>(a) {}
    explicit persistent_dynamic_array(typename iv_dynamic_array<V, A,
        E>::size_type s, V const& v = V(),
        A const& a = A()) : iv_dynamic_array<V, A, E>(s, v, a) {}
    persistent_dynamic_array(persistent_dynamic_array<V, A, E> const&
        o) : iv_dynamic_array<V, A, E>(o) {}
    typedef V value_type;
    typedef E entry;
    typedef entry* concrete_iterator;
    typedef typename entry::position_type position_type;
    typedef std::ptrdiff_t difference_type;
    typedef std::size_t size_type;
    typedef entry** array;

  private:
    typedef cphstl::persistent_avl_tree<E*,
                                        E*,
                                        cphstl::unnamed::identity<E*>,
                                        my_cmp<E>,
                                        A,
                                        my_node<E*, value_type>
                                        > persistent_tree;
  public:
    typedef cphstl::stack_iterator<typename persistent_tree::node_type
        , false> persistent_iterator;
    typedef cphstl::stack_iterator<typename persistent_tree::node_type
        , true> persistent_const_iterator;
  private:
    /* persistent tree */
   typedef cphstl::persistent_set<E*,
                                  my_cmp<E>,
                                  A,
                                  persistent_tree,
                                  persistent_iterator,
                                  persistent_const_iterator
                                 > pset;
    pset s;
  protected:
    void _erase(size_type num) {
      s.erase((*this).array_p[num]);
    }
    entry* _insert(value_type const& v, size_type num) {
      entry* tmp = (*this).entry_allocator.allocate(1);
      new (tmp) entry(v, position_type((*this).hld, num));
      s.insert(tmp);
      return tmp;
    }
  public:
    /* The purpose of this class is to provide the snapshot method */
    std::pair<persistent_iterator, persistent_iterator> snapshot() {
      return std::pair<persistent_iterator, persistent_iterator>(s.
          begin(), s.end());
    }
  };
}
```

Appendix A.18 persistent_vector.h++

```
#include "vector.h++"
```

```
namespace cphstl {
  template <
    typename V,
    typename A = std::allocator<V>,
    typename R = std::vector<V, A>,
    typename I = typename R::iterator,
    typename J = typename R::const_iterator
  >
  class persistent_vector : public vector<V, A, R, I, J> {
  public:
    typedef typename R::persistent_iterator persistent_iterator;
    std::pair<persistent_iterator, persistent_iterator> snapshot() {
      return (*this).kernel.snapshot();
    }

  };
}
```

Appendix A.19 test_persistent.c++

```
#include "pers_dynamic_array.h++"
#include "persistent_vector.h++"
#include "extended_entry_iterator.h++"
#include "entry_iterator.h++"

typedef int V;
typedef std::allocator<int> A;
typedef cphstl::persistent_dynamic_array<V, A, cphstl::
    iv_dynamic_array_entry<V,A> > KERNEL;
typedef cphstl::entry_iterator<KERNEL, cphstl::iv_dynamic_array_entry<
    V,A>, false> ITERATOR;
typedef cphstl::entry_iterator<KERNEL, cphstl::iv_dynamic_array_entry<
    V,A>, true> CONST_ITERATOR;

typedef cphstl::persistent_vector<V, A, KERNEL, ITERATOR,
    CONST_ITERATOR> cont;
typedef cont::iterator cont_it;
typedef cont::persistent_iterator cont_pit;

int main() {

  cont v;

  cont_it it1 = v.insert(v.begin(), 5);
  cont_it it2 = v.insert(v.begin(), 7);
  cont_it it3 = v.insert(v.begin(), 9);

  std::pair<cont_pit, cont_pit> p = v.snapshot();

  v.erase(it2);

  for(cont_pit it = p.first; it != p.second; ++it) {
    std::cout << "Elem:␣" << *it << std::endl;
  }

}
```