A framework for implementing associative containers

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

Abstract. The STL provides containers which implement data structures. These containers include: `vector` which is a dynamic array, `list` which is a linked list, `set` which stores an ordered collection of elements, and `map` which is similar to set but each element is associated with a value. In this paper we study the implementations of `set` and `map`. The usual implementations of these containers are balanced search trees. There are several variants of balanced binary trees, but all variants are implemented in the same way except for a few routines; these routines are restoring balance after insertion and deletion. In this paper we demonstrate how we decoupled the implementation specific part of the code, and obtained a framework for implementing binary search trees.

Keywords. frameworks, binary search trees, generic programming

1. Introduction

Recently, the Performance Engineering Laboratory has conducted research in the area of software architecture for a generic program library. We specified the architecture in order to structure the library and to obtain better maintainability by reducing the amount of redundant code [14]. In particular, we have focused on the construction of iterators [16, 17]. Every container has to provide iterators, before the refactoring, every iterator class was declared explicitly meaning that it defined all the operations described in the C++ standard [11].

We observed a recurring pattern for construction of container classes: every container uses a storage class or a *storage policy*, which is given to the container as a template argument. We took advantage of this construction by letting the storage policy implement member functions which defined the behaviour of concrete iterators. After a careful analysis we concluded that only two abstract iterator classes (one for bidirectional iterators and one for random access iterators) were sufficient to create iterators for almost every container.

We also defined container classes by application of the bridge design pattern [8], which made it possible to have several realizators for the same

CPH STL Report 2009-3, January 2009, Revised April 2009.

container, for example, associative containers (`map` and `set` and their multi variants) can be implemented by both red-black trees and AVL trees. For this particular example we observed that we can go even further with respect to obtaining more shared code. The only difference between these data structures is the mechanism for restoring balance. This mechanism is used when the tree becomes unbalanced because of modifying operations. To decouple the mechanism which restores the balance, we created balancing policies which should be given as a template argument to the container similar to the storage policies. After each insert and erase operation the balancing policy is invoked to restore balance. This idea is described in [3].

This new policy-based design implies a framework, since it has no functionality without the appropriate policies[1]. We denote an instance of the framework with the corresponding policies a *configuration*. In Figure 1, a sample configuration of the framework is shown; this configuration represents an AVL tree.

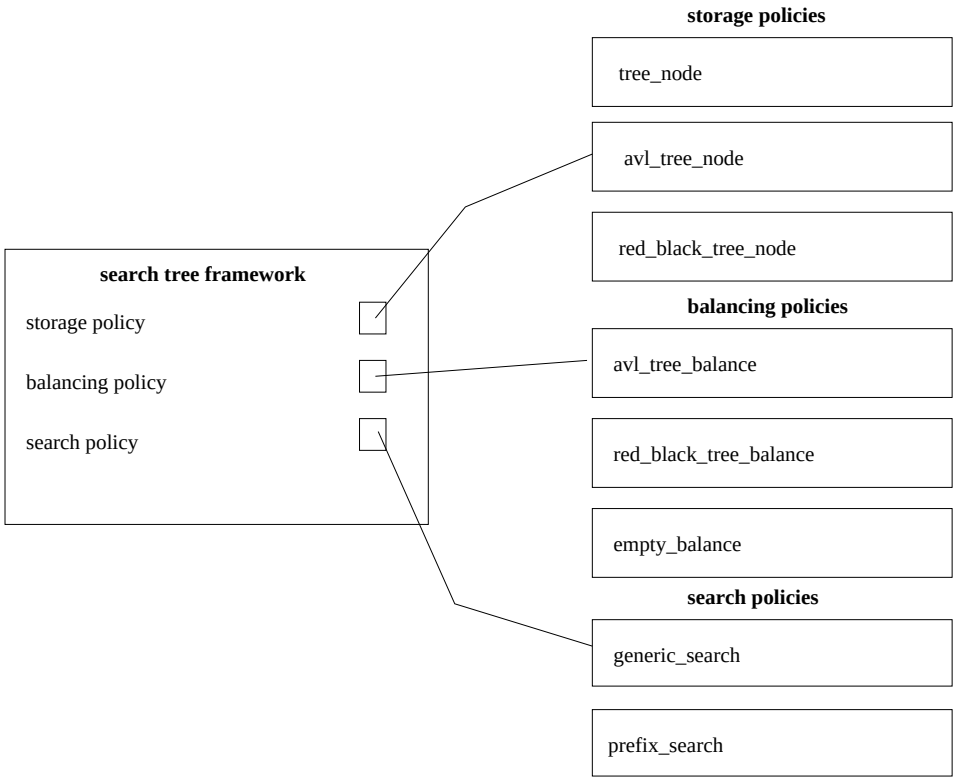


Figure 1. An example configuration of the framework.

An alternative approach to our policy-based design could be to use the

[1] When the framework is given an empty balancing policy, the framework implements a basic binary search tree.

generic template method [7], where a skeleton of the implementation is defined in the base class, and implementation specific details are left to subclasses. This approach would be preferable if the variants of the data structures did differ more, for example, a policy-based design would be difficult to use, to implement a shared base for both vector and safe vector. Vector is based on a plain array containing the values and the safe vector is based on a pointer array where the values are moved to objects [17]. Here the generic template method would be more appropriate.

This paper is organized into the following sections: In Section 2 we describe the design of the container classes and improvements of code reuse, in Section 3 we describe the design of the framework, in Section 4 we describe the configuration policies, and finally we consider some other ways to obtain ordered containers in Section 5. The code for the entire framework can be found in Appendix.

2. Container classes

The container classes involved in this design are the following:

`set<V, C, A, R>`: Stores an ordered and unique (only one element per key) collection of elements.

`multiset<V, C, A, R>`: Stores an ordered collection of elements, but allows several elements with the same key.

`map<K, V, C, A, R>`: Stores unique keys (of type `K`) with their associated values (of type `V`).

`multimap<K, V, C, A, R>`: Stores several elements with the same key and associated values.

The container classes are an application of the bridge design pattern. The main purpose of the bridge design pattern is to decouple the abstraction from the implementation. However there is also a second purpose in this context: all four container classes should be realized using the same realizator. Therefore our framework is just one class. This means that the container classes and the realizator classes do not have the same interface. The interface of the realizator class should at least contain the following template parameters `realizator<K, V, F, C, A, bool is_bag, ...>`. The parameters are: the type of the key `K`, the type of the value `V`[2], the type of comparator `C`, and the type of the allocator `A`. The real difference between the interface of the realizator and the container classes is the template parameters `F` and `is_bag`. `F` is defined to be the function $F_{map} : V \to K$ which should be invoked every time elements are compared according to their keys. For `set`, F is defined the identity function $F_{set} : V \to V$. The template parameter `is_bag` is a Boolean value which is used to inform the realizator whether it should store unique keys or not.

[2] for `set V=K` and for `map V=std::pair<K, mapped_type>` where `mapped_type` is the `V` given to `map`.

The class members for the container classes are specified in the C++ standard, and one can see there is a lot of shared members, for example, the significant difference between `set` and `multiset` is the return types of the insert functions. Therefore, we have created an inheritance hierarchy for these container classes, in order to obtain code reusability and better maintainability. The hierarchy is shown in Figure 2. In the `set_base` class all

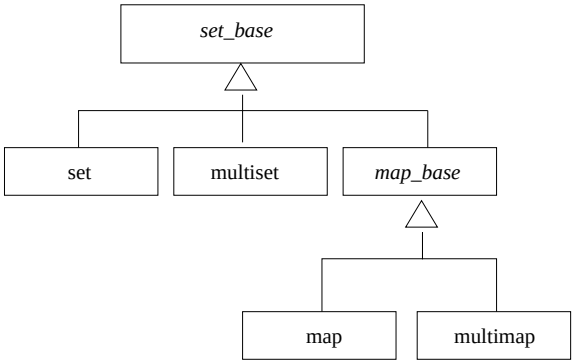


Figure 2. Inheritance hierarchy for ordered containers

members which are the same for all 4 container classes are defined, and in `map_base` all members which are the same for both `map` and `multimap` are defined. When the new C++0x standard is accepted all containers have to provide some extra methods (e.g. `cbegin()` and `cend()` [4]) which means that we should only edit the `set_base` class. What could be even better: create a base class for all containers which defined, for example, `begin`, `end`, which all container classes should provide.

3. Framework

In this section we describe some of the non-trivial issues which are related to all implementations of balanced binary trees. They are therefore implemented in the framework only. As a base for the framework the Safe AVL tree implementation was used, the state of this implementation was reasonable. The challenges were:

– To provide different space efficiency. According to [14] we should provide at least nodes of size: 6-, 5-, 4-words. For 6-word nodes, we have a linked list inside the tree, which we do not have for 4-word nodes. We need to implement possibilities for both 4- and 6-word nodes with just one realizator class, and the user should, by selecting the appropriate storage policy, obtain the desired space efficiency. Using the linked list will require 5 words of pointers (3 words without the linked list) and additional 1 word of book-keeping information. We will also show how we can obtain 5- and 3-word nodes. Some research [12] on space effi-

ciency has been done but that solution is not designed to be used in a framework.
- To decouple the balancing and searching routines from the Safe AVL tree and move it to policies. Furthermore, the balancing routines should also be decoupled from other balanced binary trees in the CPH STL.
- To handle duplicates for multiset. If the duplicates are stored in the tree, the tree will get unnecessary higher, we need another approach.

3.1 Space efficiency and tree structure

According to the C++ standard, iterator operations should be done in amortized constant time. The usual way of performing the iterator operations `it++` and `it--` are to find the successor and predecessor node respectively. The algorithms are given in [5], but they both take $O(\lg n)$ worst-case time. However, in the CPH STL project [6] we require constant time in the worst case. To obtain this we maintain a linked list of nodes. When performing insertion and deletion we ensure that the linked list contains the nodes in sorted order. The extra operations do not change the time complexity asymptotically. However, it requires two additional pointers for each node which we denote the `predecessor` and `successor`. Now, the iterator operations `it--` and `it++` will use the corresponding pointer. Now, the operations takes constant worst-case time. We use a sentinel node to store the pointer to the root. The sentinel node is convenient since `end()` should return one element past the end. Since the user should be able to use the framework with or without the linked list the behaviour of the sentinel node differs. With the linked list the sentinel node H should contain:

predecessor$[H]$: The right most elements. Equivalent to `--c.end()`.

successor$[H]$: The left most element. Equivalent to `c.begin()`.

parent$[H]$: The root node.

The linked list is circular and the following invariants hold: `--c.begin() == c.end()` and `++c.end() == c.begin()`. Without the linked list the sentinel node H should contain:

right$[H]$: The right-most element. Equivalent to `--c.end()`.

left$[H]$: The left-most element. Equivalent to `c.begin()`.

parent$[H]$: The root node.

An example of trees with and without the internal linked list can be seen in Figure 3. As mentioned we should also provide 3- and 5-word nodes. These variants can be obtained using bit-packing. Because of word alignment the last two bits of every pointer are not used. Most variants of balanced binary trees store some kind of book-keeping information. In several cases this book-keeping information is in a small interval, e.g. for red-black trees it is a Boolean value if the node is black or not. This means that we can use the bit-packing technique to pack this bit into a pointer.

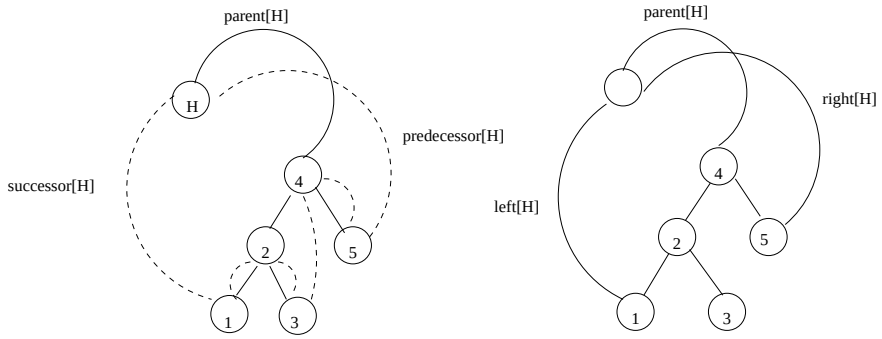


Figure 3. On the left the 6-word nodes, the dashed lines represents the linked list and the other lines represents the tree structure. On the right the 4-word nodes.

3.2 Decoupling

The interface for the tree framework is the following:

```
template <
    typename K,
    typename V = K,
    typename F = cphstl::unnamed::identity<V>,
    typename C = std::less<K>,
    typename A = std::allocator<V>,
    bool is_bag = false,
    typename N = cphstl::red_black_tree_node<V>,
    typename B = cphstl::red_black_tree_balance_policy<
        red_black_tree_node<V> >,
    typename S = cphstl::tree_search<K, V, F, comparator_proxy<C>, N
        , B, is_multiset>
>
class tree;
```

The relevant template parameters are: `N` the storage policy, `B` the balancing policy, and `S` the searching policy. The framework will require a well-defined interface for the policy classes in order to use them properly.

First, let us consider the storage policy. This class is responsible to provide member functions to access the pointers which define the structure of the tree namely the left-, right- and parent pointers. Furthermore, it should also provide member functions for accessing the value stored in the node. For each pointer there should be a member function for storing the pointer and for retrieving the pointer. The value is set using the parameterized constructor and it can be retrieved in both mutable and immutable variants using member functions. The node class should also provide member functions for storing and retrieving predecessor and successor pointers as required by the iterator construction. For the 3- and 4-word nodes the storing member functions should simply be empty and the retrieving member functions should use the methods described in [5] to find the successor and predecessor nodes. The interface which the node class should provide is described in Table 1. The interface is not the same as described in [14], where only one member function per member data is needed to access the

member data. These member functions return references, but this will not work easily when doing bit-packing. The constant `N::has_list` is used by the framework ensure that the correct operations are performed for either 6-word nodes or 4-word nodes. An example for insert is:

```
if (N::has_list) {
  /* add to the linked list */
}
else {
  /* check if the newly inserted node is a candidate for leftmost or
      rightmost */
}
```

The rotation member functions take a node pointer as parameters, since when the root node is rotated, we need to update the pointer to the root node, which is stored as the parent of the sentinel node. Therefore the rotation member function is called with the pointer to the sentinel node as argument, so we can immediately update the pointer. There are two predefined node classes, which contains this interface: One which provides a basic 6-word node; this class is called `tree_node`. The second predefined node class provides a basic 4-word, which means there are no predecessor or successor pointers; this class is called `tree_se_node`. The purpose of these classes is that the final nodes should inherit from these classes, and extend the base classes with book-keeping information and other things needed by the balancing and searching routines. To decide which predefined class a storage policy should inherit from, we provide the `tree_node_selector` class which selects one of the two node classes given a Boolean value `se`. The Boolean value is true for space efficient nodes (4-word nodes), and it is false for 6-word nodes. Now, an implementor of a new node class can use these classes with the following skeleton:

```
template <typename V, bool se = false>
class my_node : public tree_node_selector<V, se, my_node<V, se> >::
    node_type  {
};
```

We use the CRTP idiom [13] to ensure that the base class will use the `my_node` type for all the pointers in the base class (recall the pointers are defined in the base class). The usage of this idiom avoids type casting.

The balance policy is reduced to only three methods, one method for restoring balance after insertion, one method for restoring balance after deletion, and finally one method for restoring balance after search. The last method is not needed by all balanced trees, but it is needed for e.g. splay trees. The interface for the balance policy class is shown in Table 2.

The search policy has been reduced to only two methods, The `find_node` method is used to implement `lower_bound` and `upper_bound` using a corresponding functor [13]. The methods `find` and `count` are implemented only in the scope of the container classes. The erase method is also using `lower_bound` to find the node to be deleted. The `insert(V const&)` needs a different variant than the `find_node` member function, since we are searching for an empty slot for the node to be inserted and not searching for

Table 1. The interface for the storage policy, denoted the node class `N`

Template parameter	**Description**
`V`	The type of the value stored in the node
`se`	A Boolean value, which is true for space efficient nodes
Prototype	**Description**
`N::value_type`	Required by the iterator, used by the dereferencing operator
`N::has_list`	Boolean value, is true if the node is a 6-word node
`N()`	Sets all pointers to 0
`N(V const&)`	Stores the value type given as input
`N* left() const`	Returns a pointer the left node
`N* right() const`	Returns a pointer the right node
`N* parent() const`	Returns a pointer the parent node
`void left(N*)`	Stores the input pointer as left node
`void right(N*)`	Stores the input pointer as right node
`void parent(N*)`	Stores the input pointer as parent node
`N* successor() const`	Returns a pointer to the successor node
`N* predecessor() const`	Returns a pointer to the predecessor node
`void successor(N*)`	Stores the input pointer as successor node
`void predecessor(N*)`	Stores the input pointer as predecessor node
`void copy_aux(N*)`	Copy auxiliary information (balance information)
`void rotate_left(N*)`	Perform a left rotation
`void rotate_right(N*)`	Perform a right rotation
`bool is_left() const`	If this node is the left child of the parent node return true
`bool is_right() const`	If this node is the right child of the parent node return true
`N* grandparent()`	Return `(*parent()).parent()`
`void add_to_list(N*, N*)`	Add the node to the linked list given predecessor and successor
`void remove_from_list()`	Remove the node from the linked list
`V const& content() const`	Returns an immutable reference to the value stored in the node
`V& content()`	Returns a mutable reference to the value stored in the node (only required by map)

Table 2. The interface for the balance policy

Template parameter	**Description**
`N`	The type of the node
Prototype	**Description**
`insert_fixup(node_type* z, node_type* header)`	Restores balance from the newly inserted node `z`.
`erase_fixup(node_type* z, node_type* y, node_type* header)`	`z` is the deepest node, `y` is the node to be deleted. Restores balance before deletion of `y`, but after pointer-relink.
`touch(node_type* z, node_type* header)`	Restores balance after successful retrieval of node `z`.

an existing node. This member function is called `insert_find_node`. The interface for the search policy is shown in Table 3.

Table 3. The interface for the search policy

Template parameter	**Description**
`K`	The type of the key
`V`	The type of the value
`F`	The functor $F : V \to K$
`C`	The comparator
`N`	The type of the node
`B`	The balancing routine
`is_bag`	Is the container a multi variant?
Prototype	**Description**
`template <typename Z> N* find_node(K const& k, Z const& z, N* header, C const& c)`	Use `k` to satisfy the condition given by functor `z` by searching the tree from `(*header).parent()`. The comparator `c` is given to the functor `z`. Returns the node which was the last satisfaction of the functor `z`.
`Q insert_find_node (V const& v, N* header, C const& c)`	The same principle as `find_node`, but we will return successor, predecessor and the parent node, the new node should be attached to (we denote it `y`). This means `Q` has the following structure `std::pair<N*, std::pair<N*, N*> >(y, std::pair<N*, N*>(pred, succ))`

3.3 Copy construction

To make a copy of the data structure using $O(n)$ worst-case time, becomes hard while building both the linked list and the tree structure. To make a copy of the data structure we can trivially traverse over the elements of the source container, and use the insert member function to insert the elements into the destination container. But this takes $\sum_{i=1}^{n} O(\lg i) = \Theta(n \lg n)$ worst-case time which violates the requirements specified in the C++ standard. Knowing that we can traverse the container in sorted order using $O(n)$ time is helpful here. We can now build an array which contains the elements in sorted order. By selecting the median we can use the divide-and-conquer paradigm to build a perfect balanced binary tree (height $h \leq \lg n + 1$) in $\Theta(n)$ worst-case time. At the starting point the median element is selected to be the root node, and the left and right subtrees are built by calling the method recursively. In Figure 4 the principle of this algorithm is shown.

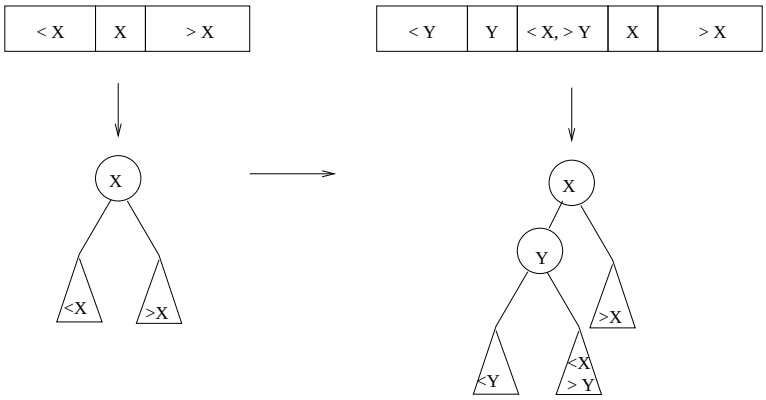


Figure 4. The principle of the algorithm BUILD-TREE

We denote the algorithm BUILD-TREE and it is shown in pseudo-code in Algorithm 1. The algorithm will return the root node of the newly built tree. The auxiliary data mentioned in the algorithm is information related to the data structure e.g. for the AVL tree this information would be the balance. The procedure of building a copy of tree T is as follows:

1. Build an array of copies of the nodes in T ($\Theta(n)$ worst-case time).
2. Use the BUILD-TREE algorithm to construct the tree.
3. Link the nodes together by traversing the array ($\Theta(n)$ worst-case time).

As shown above the time complexity of the entire procedure depends on the BUILD-TREE algorithm.

Proposition 1. BUILD-TREE *constructs a balanced binary tree in* $\Theta(n)$ *worst-case time.*

Proof. We can construct the following recurrence equation: $T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + O(1) = 2T(\frac{n}{2}) + O(1)$. This can be solved by the Master theorem,

Algorithm 1 BUILD-TREE(a, s, n)

Require: An array of pointers to nodes a of size n, s start position. The nodes in the array are in sorted order.

Ensure: x is the root of the copied subtree.

```
1: if n = 0 then
2:    return nil
3: end if
4: median = ⌊n/2⌋
5: x = a[median]
6: left[x] = BUILD-TREE(a, 0, median)
7: if left[x] ≠ nil then
8:    parent[left[x]] = x
9: end if
10: right[x] = BUILD-TREE(a, median + 1, n − median − 1)
11: if right[x] ≠ nil then
12:    parent[right[x]] = x
13: end if
14: {Add auxiliary data}
15: return x
```

using the first case: $O(1) \in O(n^{\lg_2 2-1})$ where $\epsilon = 1, a = 2, b = 2$. This reduces to: $T(n) \in \Theta(n)$, which completes the proof. □

Because of proposition 1, we can now conclude that we construct a copy of the binary tree using $\Theta(n)$ time and space.

There is one problem with this approach, we construct a perfect balanced binary tree, since it is not an identical copy, it may be problematic to compute the auxiliary information. We need a better approach, which constructs an identical copy of the tree in order to keep the auxiliary information consistent. This approach is simpler, since we will make the copy using in-order traversal, but store the copied nodes in a array, which will be in sorted order because of the in-order traversal. Finally we link the nodes together using a traverse of the array.

The new algorithm is shown in Algorithm 2. We still use $\Theta(n)$ worst-case time, but like the first approach we need auxiliary space in the term of $\Theta(n)$ pointers. It has not been possible to find a solution which could perform copy construction without auxiliary space. When the framework is used without the linked list, we will omit the step of linking the nodes together, instead we will find the minimum and maximum node.

3.4 Multiset issues

As mentioned we want to improve the tree structure for multiset. Usually we keep the duplicates in the tree, but we can keep them outside the tree i.e. in the linked list, such that the tree gets minimized, and thereby faster

Algorithm 2 NEW-BUILD-TREE(r, a, c)

Require: The root of a tree r, an array of pointers (by reference) to nodes a, and c the node pointer (by reference).

Ensure: The array a contains c elements in sorted order. *new_node* is the root of the built subtree.

```
1: if r = nil then
2:    return nil
3: end if
4: new_left = NEW-BUILD-TREE(left[r], a, c)
5: {allocate a new copy of r, and denote it new_node}
6: a[c] = new_node
7: c = c + 1
8: left[new_node] = new_left
9: right[new_node] = NEW-BUILD-TREE(right[r], a, c)
10: if left[new_node] ≠ nil then
11:    parent[left[new_node]] = new_node
12: end if
13: if right[new_node] ≠ nil then
14:    parent[right[new_node]] = new_node
15: end if
16: return new_node
```

to search. This idea is shown in Figure 5. There is no reason for keeping them in the tree, because the usual way to access duplicates is to search the tree for the first and last element, using `lower_bound` and `upper_bound` respectively, and then using the iterator given by `lower_bound` to iterate until the iterator given by `upper_bound` is reached. This method will only work if the user is using the appropriate node which supports the linked list, else the duplicate nodes will be stored in the tree.

We need several changes in the implementation: When inserting we know according to the predecessor and successor values whether the node to be inserted is a duplicate or not. If it is a duplicate, we will simply add it to the list in the end. Now, the elements are in correct order according to the point of time they were inserted. We identify duplicates by checking if the parent pointer is nil. If we did not use the last duplicate there will be a problem. Let us consider a tree storing $5, 5, 7$, and we want to insert 6. If we did not use the last duplicate, the order would become $5, 6, 5, 7$. We can get the last duplicate efficiently by using an extra pointer. The node stored in the tree should use this pointer to point at the last duplicate. This gives us insertion of duplicates in $O(1)$ worst-case time. But we would rather spend a little extra time to avoid the extra space. We will use the `upper_bound` member function to get the next unique element and use the `predecessor` member function to get the last duplicate. This uses $O(\lg n)$ additional time but we will still use $O(\lg n)$ overall time for the entire inser-

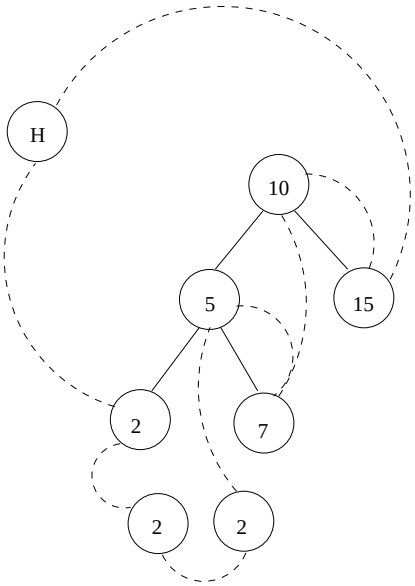


Figure 5. Duplicates in multiset, the linked list is represented by the dashed edges, and the others are the usual tree structure.

tion procedure. We cannot afford $O(\lg n)$ time when inserting an element by locator (`insert(iterator, value_type)`). According to the standard this should take $O(1)$ amortized time, but our balance operations will still require $O(\lg n)$ time for some variants of balanced search trees. Deletion of duplicates is a bit more tricky to handle. If we delete a node which is stored in the tree, we will swap in the next duplicate, we will relink the pointers in order to maintain iterator validity. If we are deleting a duplicate, which is not stored in the tree, we will simply remove it from the list. This gives us deletion of duplicates in $O(1)$ worst-case time.

There is yet another problem with our multiset solution which is related to the code. As mentioned there is some differences between `set` and `multiset` which we need to handle. One of them is that the return types differ for insert. The reason is that for set, we need to know whether the element is inserted or not (if it is already in the set, we do not insert it), but for multiset all elements will be inserted. In the table below the prototypes are shown. We can get different return types of the insert method by using, for

Prototype in set	**Prototype in multiset**
`std::pair<iterator, bool> insert(V const&);`	`iterator insert(V const&);`

example, compile-time selections. This is done by using the `if_then_else` trait as described in [13]. However this does not solve the problem entirely, but it is still a part of the solution. Let us look at this scenario:

```
template <typename K, typename V, typename F, typename C, typename A,
    typename N, bool is_multiset>
class realizator {
  std::pair<iterator, bool> insert_set(V const& v);
  iterator insert_multiset(V const& v);

  typename cphstl::if_then_else<is_multiset, iterator, std::pair<
      iterator, bool> >::type
  insert(V const& v) {
    if(is_multiset) {
      return (*this).insert_multiset(v);
    } else {
      return (*this).insert_set(v);
    }
  }
};
```

This will not work, because code for both insert functions are generated. Because the member function's return types differ, we will get a compile-time error. What we desire is that code for just one member function is generated (either `insert_multiset` or `insert_set`). The generalization of this problem is to make a partial specialization of a member function, but this is not possible in the current C++ standard. In C++ we can perform type dispatching, which means that we can define several functions with the same name but different types. The compiler will select the one which matches the type parameters. Here the input types are identical but the return types differ. But we can define our own type using this trait:

```
template <bool v>
class bool2type {
public:
  enum { value = v };
};
```

We will use this trait in our new insert function such that the appropriate method is called:

```
template <typename K, typename V, typename F, typename C, typename A,
    typename N, bool is_bag>
class avl_tree {
  std::pair<N*, bool> insert_dispatch(const V& v, cphstl::bool2type<
      false>);
  N* insert_dispatch(const V& v, cphstl::bool2type<true>);

  typename cphstl::if_then_else<is_multiset, N*, std::pair<N*, bool>
      >::type insert(V const& v) {
    return (*this).insert_dispatch(v, cphstl::bool2type<is_multiset>()
        );
  }
};
```

3.5 Design issues

The original code for the Safe-AVL component did provide strong exception safety. This property has been maintained. The implementor of the storage, balancing and search policies should not take exception safety into account. The storage policies modify and retrieve the member data in the node classes. An exception can occur (for example, on object creation),

but it is handled by the framework. The balancing policies are only performing pointer modifications which cannot cause exceptions. Finally, the operations performed within the search policies can throw exceptions but no modifications to the container data are performed.

An issue related to exception safety has been found: Swap must not throw an exception, but when the comparator and allocator are swapped, an exception can occur because the copy constructor of the comparator and allocator can throw an exception. A straightforward solution to this problem is to keep the comparator and allocator as pointers. To ensure code reuse, the proxy design pattern [8] has been used, to design comparator proxy and allocator proxy classes. These classes provide the same methods as the comparator and allocator, but they keep the allocator and comparator as pointers. The types of these are given by template argument. Furthermore the `swap` methods have been overloaded to ensure transparency, such that the comparator and allocator proxy can be swapped in the usual way, but what really happens is that the pointers are swapped. The declaration of the comparator and allocator should now be:

```
comparator_proxy<key_compare> comparator;
allocator_proxy<allocator_type> allocator;
```

These classes are reusable such that they can be used for any safe container in the CPH STL.

4. Balancing and storage policies

4.1 AVL trees

The AVL tree was invented by G. Adel'son-Velskii and E. Landis in 1962 [1]. The AVL tree maintains a tree of max height $1.44 \lg n$ and therefore all basic (insert, delete, find) operations are guaranteed to take $O(\lg n)$ worst-case time, since the balance is restored in $O(\lg n)$ worst-case time. Balance is restored by rotations, using a bottom-up approach. The decision of whether a rotation should be performed is based on the balance of each node. The balance is computed using the height of the two subtrees. Let us denote the height function $h(x)$ of a node x. Then the balance is $b(x) = h(\mathit{right}[x]) - h(\mathit{left}[x])$. The invariant of a balanced AVL tree is that $b(x) \in [-1; 1]$ for any node x in the tree. The balance routine is restoring balance bottom-up. When it reaches a node which does not maintain this invariant, it performs rotations to restore the invariant. The balance routine is invoked after insertion and deletion operations. The balancing policy is denoted `avl_tree_balance_policy` and the storage policy for is denoted `avl_tree_node`. We also provide a bit-packing node. There is only three possibilities for the balance, either it is -1, 0, or 1. The balance is packed into the pointers as follows:

- If $b(x) = -1$: The last bit of the left pointer is set to 1.
- If $b(x) = 1$: The last bit of the right pointer is set to 1.
- If $b(x) = 0$: Both bits are not 0.

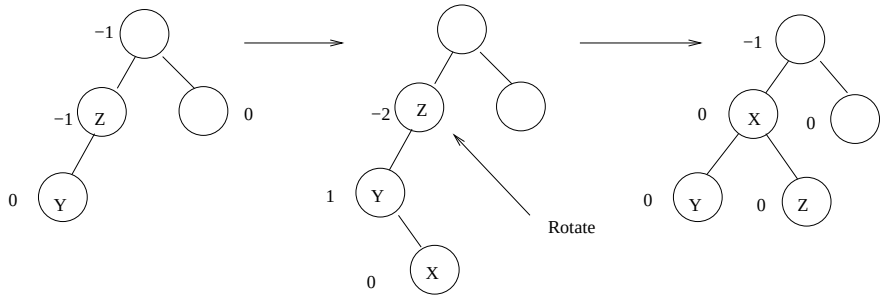


Figure 6. An example insertion on the AVL-tree.

This storage policy is called `packed_avl_tree_node`. Because of word alignment (any address is devidable by 4 with no remainder) we should be able to use the last 2 bits of every pointer, so one pointer would be sufficient for packing the balance. We have not succeeded in using both bits, only the last one. This is the reason for our rather complicated bit-packing algorithm.

4.2 Red-black trees

Red-black tree was invented by R. Bayer in 1972 [9]. The red-black tree maintains a height of maximum $2 \lg n$, and therefore all basic operations are guaranteed to take $O(\lg n)$ worst-case time, since balance is restored in $O(\lg n)$ worst-case time. Each node has a colour and the balance is restored after modifying operations using the colour. The tree structure maintains the following invariants:

- A node is either red or black.
- The root and all leaves are black.
- Both children of every red node are black.
- Every simple path from a node to a descendant leaf contains the same number of black nodes.

The balance routines for both insert and erase ensure that these invariants are true by either recolouring or performing rotations. Figure 4.2a gives an example of how the invariant is restored without rotations but simply by recolouring and Figure 4.2b gives an example of how the invariant is restored by rotations and recolouring.

The code for the balancing policy is drawn from the Safe red-black tree implementation and the balancing policy class is denoted `red_black_tree_balance_policy`. The storage which includes the colour is called `red_black_tree_node`. We also provide a bit-packing variant of the node, where the colour is packed into the parent pointer. This node class is called `packed_red_black_tree_node`.

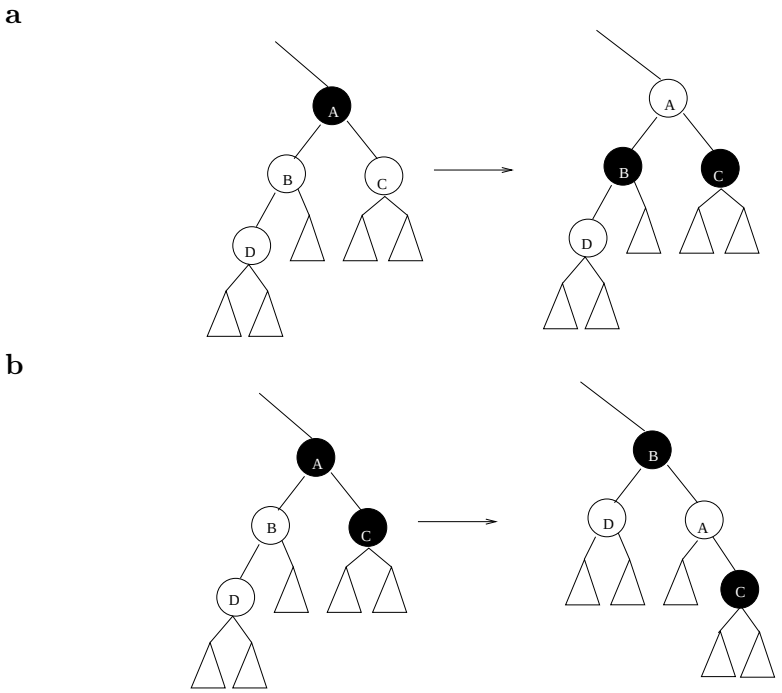


Figure 7. An example of the invariant being restored: **a** without rotations **b** with rotations. The white nodes represent red nodes

4.3 Splay trees

Splay trees were invented in 1985 by D. Sleator and R. Tarjan [18]. All operations are guaranteed to be take amortized $O(\lg n)$ time. The data structure keeps the previous accessed node (by either find or insert) as the root node. This means that this data structure is appropriate when some nodes are more used than others. Balance is restored bottom-up using three different steps. Notice that no book-keeping information is necessary. Let X denote a node, Y the parent of X, and Z the parent of Y.

1. **Zig step:** Z is the sentinel, Y is the root, and X is a child of the root. Perform a rotation on Y.
2. **Zig-zig step:** X is the left child of Y and Y is the left child of Z (symmetrically for right). Perform right rotations (symmetrically left) on Y and Z. This is shown on the left in Figure 8.
3. **Zig-zag step:** X is the left child of Y and Y is the right child of Z (symmetrically for right, left). Perform left rotation on Y and right rotation on Z (symmetrically left, right). This is shown on the right in Figure 8.

The implementation of splay trees was written from scratch inspired by [3]. Splay trees differs from other kinds of balanced binary trees, since we should also invoke the balancing routine after a search (`lower_bound`, `upper_bound`), since the most recent accessed node should be the root.

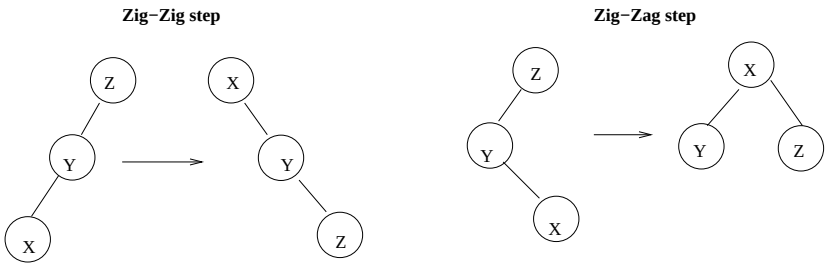


Figure 8. Splay steps to restore balance.

Therefore the `touch` member function in the balancing policy is implemented, and it calls like the `insert_fixup` and `erase_fixup` the `splay` routine. The balancing policy is called `splay_tree_balance_policy`, and the node class is called `splay_tree_node`. The node class does not extend the base class, because no book-keeping information is required. The reason for the existence of the class is simply naming symmetry.

There is one big problem with splay trees: The worst-case time complexity for all operations is $O(n)$. The worst-case scenario occurs when a sorted sequence is inserted and the data is accessed in sorted order too. The running time is $O(n)$ since the result is a linked list. This causes a runtime-error when the framework is used with this balancing policy and the number of elements is 10^6 elements or more. The reason for the runtime-error is that the tree is destroyed recursively, which gives a stack overflow for that big amount of recursions. This could be solved using an iterative approach which involves a stack, but this would slow down the framework when other balancing policies are used. Instead, the user should not use this data structure when the insertion order is sorted, but properly benefit from using another implementation. Also, a splay tree performs useless balance operations when it is used on a sorted sequence, when accessing, for example, the half of the elements, the tree is almost balanced, but when the entire sequence is accessed, the tree is a linked list again.

4.4 AA trees

AA trees were invented in 1993 by A. Andersson [2]. The AA-tree is a variant of red-black trees[3] therefore the operations are also guaranteed to take $O(\lg n)$ worst-case time. Balance is restored bottom-up in two steps, the *skew* and the *split* step. The skew step performs a right rotation when a left horizontal link occurs in order to eliminate them. The horizontal link representation is shown in Figure 9. The split step performs a left rotation when two consecutive horizontal right links occur. Book-keeping information which is denoted the *level*, which is one for a leaf node, for a left node it is strictly less than its parent node's level, and for a right node

[3] red-black trees are isometric to 2-4 tree, AA-trees are isometric to 2-3 tree

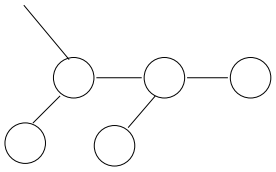


Figure 9. Horizontal link representation.

it is less or equal than its parent node. The level is used in the split routine to determine if a split is necessary. The AA-tree balancing policy class is called `aa_tree_balance_policy`; the code has been drawn from the Safe AA tree implementation. The associated node class is called `aa_tree_node`. The level is bounded to be $o(\lg \lg n)$ bits, however this number is too large for bit-packing, unless we limit the size of the tree. This means that a disadvantage of using AA trees is that we can not reduce the amount of space which is used per node.

4.5 Usage

The default implementation is currently the red-black tree implementation (balancing and node policies) since it performs best according to the benchmarks. However, in the future, if a faster data structure is found, this can easily be changed. A user can use the tree framework with the default implementation by the following code:

```
#include "node_iterator.h++"
#include "stl_set.h++"
#include "tree.h++"
int main() {
  typedef cphstl::set<int, std::less<int>, std::allocator<int>, cphstl
      ::tree<int>,
  cphstl::node_iterator<cphstl::red_black_tree_node<int>, false>,
  cphstl::node_iterator<cphstl::red_black_tree_node<int>, true> > cont
      ;
  cont c;
  ...
}
```

Set has to be given all template arguments since the default realizator of set is `std::set`. Unfortunately the syntax is very complicated, but it gets even worse when a non-default implementation is desired:

```
#include "node_iterator.h++"
#include "stl_set.h++"
#include "tree.h++"
#include "avl_tree_balance.h++"
#include "avl_tree_node.h++"
int main() {
  typedef cphstl::set<int, std::less<int>, std::allocator<int>,
  cphstl::tree<int, int, cphstl::unnamed::identity<int>, std::less<int
      >, std::allocator<int>,
  cphstl::avl_tree_node<int, true>, cphstl::avl_tree_balance_policy<
      cphstl::avl_tree_node<int, true> > >,
  cphstl::node_iterator<cphstl::avl_tree_node<int, true>, false>,
```

```
  cphstl::node_iterator<cphstl::avl_tree_node<int, true>, true> > cont
      ;
  cont c;
  ...
}
```

Since we do not use the default implementation, all template arguments have to be given. The cryptic and large declaration is the only real disadvantage of providing larger flexibility for the user of the library. This issue is discussed in [14], and our proposal is to create a preprocessor which ensures that template parameters are given once, and it should allow the use of default template parameters which does not depend on the order. For the previous example, with our preprocessor it should be possible to just give the type of the value, the balancing-, storage-, and searching policies. The rest should be default.

5. B-Trees

According to [10], B-Trees perform well when they are used to implement `map` and `set`. A B-Tree stores several elements in each node (in an array). These elements are kept sorted and the nodes are linked together. Each element in the child node must be less than the associated key stored in the parent node, to be more precise: Each node keeps an array of pointers to children ($child[]$) and keys ($key[]$), the following invariants should be true for any index y in any node in the tree:

- for $y = 0$: $\forall x \in child[0] : x < key[0]$.
- for $y > 0$: $\forall x \in child[y] : x < key[y] \wedge x > key[y-1]$.
- for $y = n$: $\forall x \in child[y] : x > key[y-1]$.

An example B-Tree is shown in Figure 10. The implementation was not

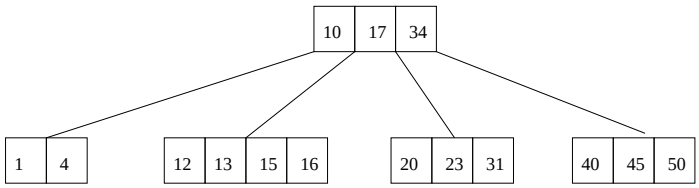


Figure 10. An example of a B-tree.

complete; one of the problems was that the implementation did not work using the container classes (`set`, `map`, etc). The reason was iterator related problems. The original implementation was refactored; the steps of the refactoring were the following:

The storage policy already used `vector` for storing the array of child pointers and the array of keys. Now, it takes the type of the array of keys as a template argument, in order to increase flexibility. The same was desired for the child array, but since it contains an array of

the type itself it was problematic. The CPH STL idiom of variables is accessed using methods were also implemented.

The iterator construction has been problematic. Since the iterator should store both the pointer to the node and the iterator of the key array. As a temporary solution we designed a concrete iterator class which holds the pointer and an iterator and it contains successor and predecessor member functions. A class named `object_iterator` has been created, which uses the concrete iterator for the `successor()` and `predecessor()` member functions.

The realizator has not been significant modified. Several methods have been removed since they were implemented in the container classes, also some new methods have been implemented to satisfy the interface the container class expect. Also an interface similar to the `tree` class has been implemented, but without tree-specific policies.

5.1 Iterator validity

Iterator validity turned out to be a problem. We have earlier conducted research of obtaining an iterator valid dynamic array [17]. The idea is to put the value into a small object denoted an *entry*, which the arrays point to, and allocate these objects explicitly. Iterators will now point to these objects and modification to the array does not affect the iterators, since the address of the entry is not changed. The problem of obtaining iterator validity for B-tree cannot entirely be reduced to the problem of obtaining iterator validity for `vector`. If it was that simple, the B-tree should simply use the iterator valid dynamic array for storing keys, but it is still a part of the solution.

In the earlier work, we did also adapt reference semantics (or pointer semantics). The STL is designed with *value semantics* in mind, which means that all container operations use values as input or iterators as locators. In algorithmic literature data structures are designed with *reference semantics* in mind. This means that all operations take an object of the type that data structure uses for storage. The reason is probably to simplify the complexity analysis. When splitting or merging nodes in the B-tree, which is caused by modifying operations, we need reference semantics in order to keep iterators valid.

There is still problems left: Iterators are still not valid, since when a key changes node (because of merge or split), the node pointer in the iterator should be updated respectively. The solution is to move the node pointer into the entry, this can be done without changing the implementation of the iterator valid dynamic array, since the entry class is given to the dynamic array as a template argument. When split and merge operations occur, the node pointer in the entry will get updated. Now the `concrete_iterator` and `object_iterator` classes are not necessary, since `successor` and `predecessor` can be defined in the entry class and we can use the `node_iterator` class to create iterators.

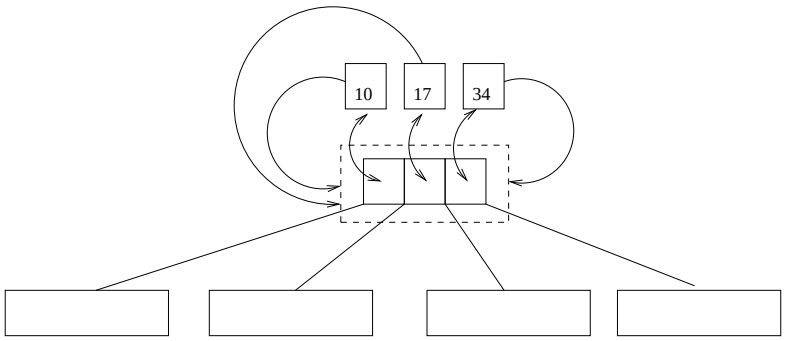


Figure 11. An iterator-valid B-Tree; the dashed box is the node and boxes inside it form the array.

The solution does not add any performance overheads theoretically. Practically there can be performance loss, since one of the advantages of using a B-tree with respect to performance is cache utilization. We can perform fast iteration if the whole node is in the cache. With the new solution where we would have the pointer array in the cache, but the real data (the entries) can be widely distributed in memory. This minimizes the chance that all elements are in the cache. Therefore, we cannot guarantee that this implementation is much faster than the conventional binary search tree for storing a large amount of data.

6. Benchmarks

We have created some benchmarks to show the efficiency of our framework compared to `std::set`. An increasing sequence $(1, ..., n)$ of n elements has been inserted into the container in random order (using `std::random_shuffle`) and accessed again in a random order (but different from the first). We used a random ordered sequence since we would not get any usable results on the splay tree implementation otherwise.

In Figure 6a elements are inserted and afterwards searched for. In Figure 6b elements are inserted and afterwards deleted. In both cases the linked list is used. The same for Figure 6, but here the linked list is not used. In Figures 6b and 6b the bit-packed variant of red-black tree is not shown because there is an error in the implementation which causes an infinite loop.

As expected, the CPH STL implementation of red-black trees is a bit slower than `std::set`. This overhead may be caused by inefficient code in the framework. What is surprising is that AVL tree is almost as fast as red-black tree. There is no significant difference between AVL tree and red-black tree. What is even more surprising is that bit-packing does not imply a performance loss. There is no significant difference between the containers without bit-packing and their corresponding variants with bit-packing. Our

hope was to obtain better performance for splay trees when using a random sequence. It has probably improved performance, but not enough. AA trees are close to red-black trees and AVL trees but still far, this is probably because more rotations are made.

7. Metrics

To show how much code is required to implement any kind of balanced binary tree with the framework, the size of each balancing (Table 4) and storage policy (Table 5) has been measured. As one can see in about 200 lines of code it is possible to implement the most popular kinds of balanced binary trees.

Table 4. Lines Of Code for the balancing policies (without comments)

Balancing policy	**LOC**
AA-tree	113
AVL-tree	114
Red-black-tree	110
Splay-tree	68
Average	101

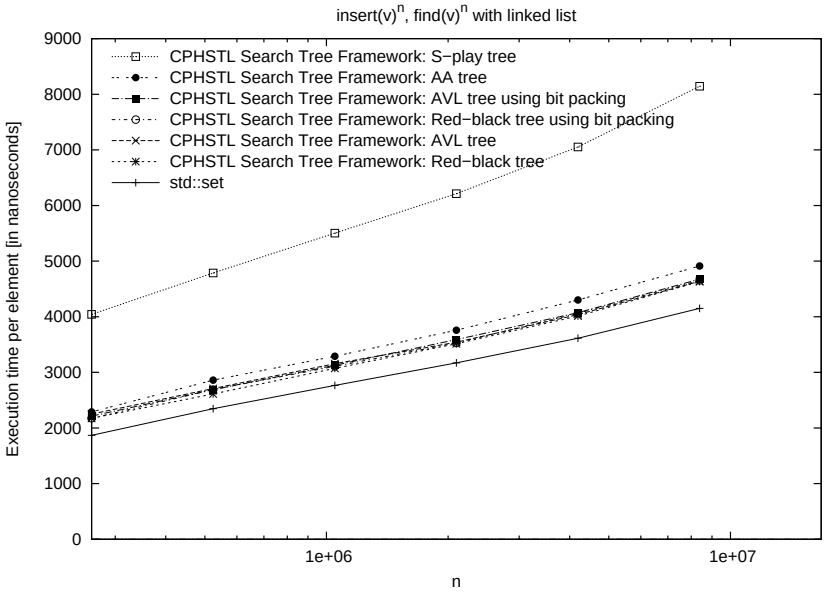
Table 5. Lines Of Code for the storage policies (without comments)

Storage policy	**LOC**
AA-tree node	74
AVL-tree node	74
AVL-tree node with bit-packing	109
Red-black-tree node	94
Red-black-tree node with bit-packing	112
Splay-tree node	35
Average	83

8. Future work

In [3] the search by rank mechanism was implemented. When performing rank search, we search for the ith node in the tree which can be done in $O(\lg n)$ worst-time, but only when the height of each subtree is stored inside each node. Searching is done by comparing these heights when searching the tree from the root down the tree. However their implementation of the tree class does not support insertion or erasal by locators (`erase(iterator, value_type)` and `insert(iterator, value_type)`. This is complicated to implement when

a



b

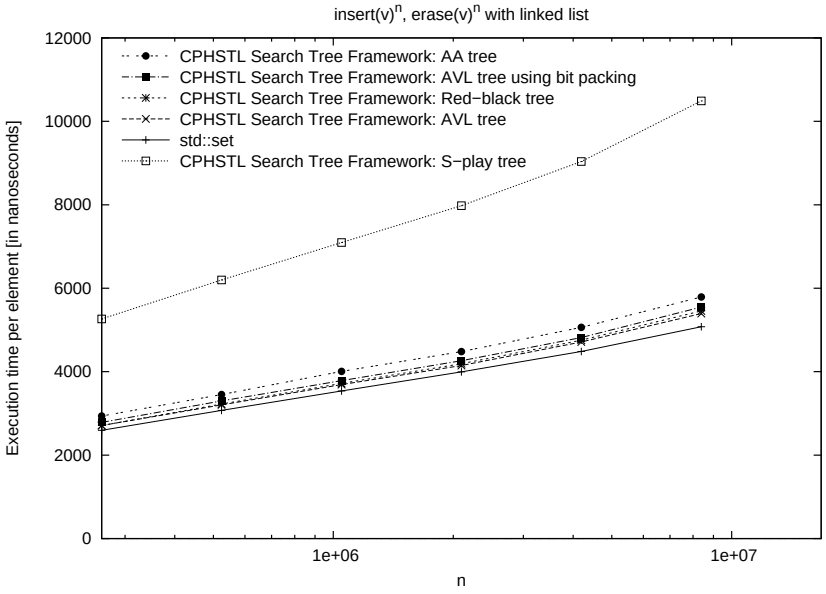
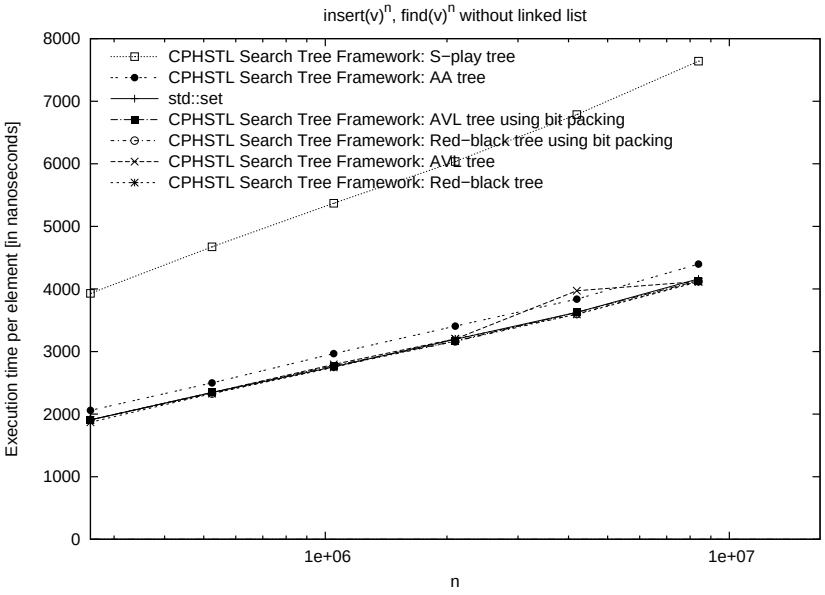


Figure 12. With linked list: **a** Insert and find **b** Insert and erase.

a



b

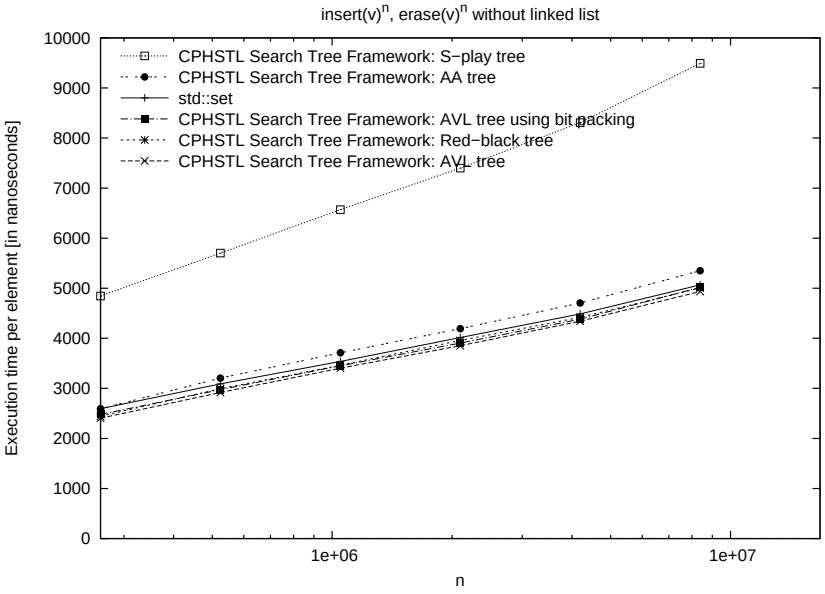


Figure 13. Without linked list: **a** Insert and find **b** Insert and erase.

maintaining the heights, since balancing operations do not necessarily reach the root node (red-black tree performs insertion by locator in $O(1)$ amortized time). Also, when doing erase, since the node to be deleted is "swapped" by its successor, the heights also need to be maintained on the path from the node to be deleted and its successor. With these observations in mind, I decided not to implement searching by rank, since it is too expensive (every rank search still takes $O(\lg n)$ worst-case time). However, we could implement it such that the user explicitly defines that this mechanism should be supported, and thereby he/she agrees that some operations become expensive.

Rank searching also opens possibility of implementing *finger search* [15], where given a finger (locator, iterator) we can advance the iterator d positions in $O(\lg d)$ time. This requires that the nodes are level-wise linked together, but this linking could be done in `insert_fixup` or `erase_fixup` since these methods should proceed to the root node when the tree is used with rank search support. Without the level-wise linked nodes, we can also perform finger search, but it would take $O(\lg n)$ time, this means asymptotically that finger search does not give any advantages (given that we know that the source node is the ith node, we can find node $i+k$ in $O(\lg n)$ time).

In [3], an implementation of optimal string search in a binary tree is given. To prove the power of our framework, it could be interesting to implement this. It could be implemented as a partial specialization of the normal search policy, but a problem related to the storage policy will occur, since the storage policy requires member data for storing the prefix. It would therefore be preferable to implement it as a separate search policy, but still the storage policy should contain both balancing data (for the balancing policy) and searching data (for the searching policy). Therefore the inheritance hierarchy is constructed as shown in Figure 14.

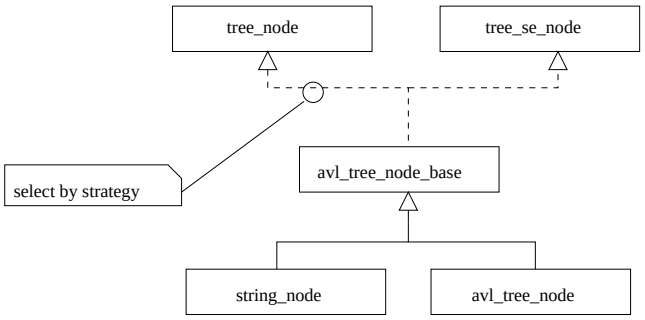


Figure 14. Hierarchy of the storage classes.

9. Concluding remarks

We can make the following concluding remarks:

- The implementation provides a flexible way of implementing several variants of balanced binary trees, with different storage efficiency (see Table 6). It also provides strong exception safety and better asymptotic bounds for duplicates. To summarize the advantages: Flexibility from the users point of view (he/she selects the balancing mechanism), better maintainability (code reuse), and finally safety. The framework also opens new possibilities for optimal algorithms, for example, the optimal string searching algorithm.
- The usability problem has become bigger, the user has to give far too many template arguments if he/she desires another implementation than the default. In future research, this problem will be given attention since it is important for obtaining users for this library.

Table 6. Storage support

Data structure	6-words	5-words	4-words	3-words
AA tree	√		√	
AVL tree	√	√	√	√
Red-black tree	√	√	√	√
Splay tree		√		√

Acknowledgements

I want to thank the persons who contributed to the realizators. These include Frej Soya, Jørgen T. Haahr, Lasse Jon Fuglsang Pedersen, Mads Ruben Burgdorff Kristensen, Anders Sabinsky Tøgern, Bergur Ziska, Jacob de Fine Skibsted, Finn Krog, Claus René Jensen, Stephan Lynge, and Hervé Brønnimann. Finally, I want to thank my supervisor Jyrki Katajainen for guidance, ideas, and motivation.

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. Andersson, Balanced search trees made simple, *Proceedings of the 3rd Workshop on Algorithms and Data Structures*, *Lecture Notes in Computer Science* **709**, Springer-Verlag (1993), 60–71.

[3] M. H. Austern, B. Stroustrup, M. Thorup, and J. Wilkinson, Untangling the balancing and searching of balanced binary search trees, *Software—Practice and Experience* **33**, 13 (2003), 1273–1298.

[4] W. E. Brown, A proposal to improve `const_iterator` use from C++0x containers, Document number N1674, The C++ Standards Committee (2004).

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms 2nd edition*, The MIT Press (2001).

[6] Department of Computer Science, University of Copenhagen, The CPH STL, Website accessible at `http://www.cphstl.dk/` (2000–2009).

[7] A. Duret-Lutz, T. Géraud, and A. Demaille, Design patterns for generic programming in C++, *Proceedings of the 6th Conference on USENIX Conference on Object-Oriented Technologies and Systems*, The USENIX Association (2001), 189–202.
[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, Addison-Wesley Professional (1995).
[9] 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.
[10] J. G. Hansen and A. K. Henriksen, The `(multi)?(map|set)` of the Copenhagen STL, CPH STL Report 2001-6, Department of Computer Science, University of Copenhagen (2001).
[11] International Organization for Standardization, *ISO/IEC 14882:2003: Programming languages — C++* (2003).
[12] C. R. Jensen and F. Krogh, Experimental evaluation of space-efficient search trees, Internal technical report, Department of Computer Science, University of Copenhagen (2008).
[13] N. M. Josuttis and D. Vandevoorde, *C++ Templates*, Pearson Education, Inc. (2003).
[14] J. Katajainen and B. Simonsen, Applying design patterns to specify the architecture of a generic program library (2008, work in progress).
[15] K. Mehlhorn and S. Näher, *LEDA: A Platform for Combinatorial and Geometric Computing*, Cambridge University Press (2000).
[16] B. Simonsen, Refactoring the CPH STL: Designing an independent and generic iterator, CPH STL Report 2008-6, Department of Computer Science, University of Copenhagen (2008).
[17] B. Simonsen, Towards stronger guarantees: Safer iterators, CPH STL Report 2009-1, Department of Computer Science, University of Copenhagen (2009).
[18] D. D. Sleator and R. E. Tarjan, Self-adjusting binary search trees, *Journal of the ACM* **32**, 3 (1985), 652–686.

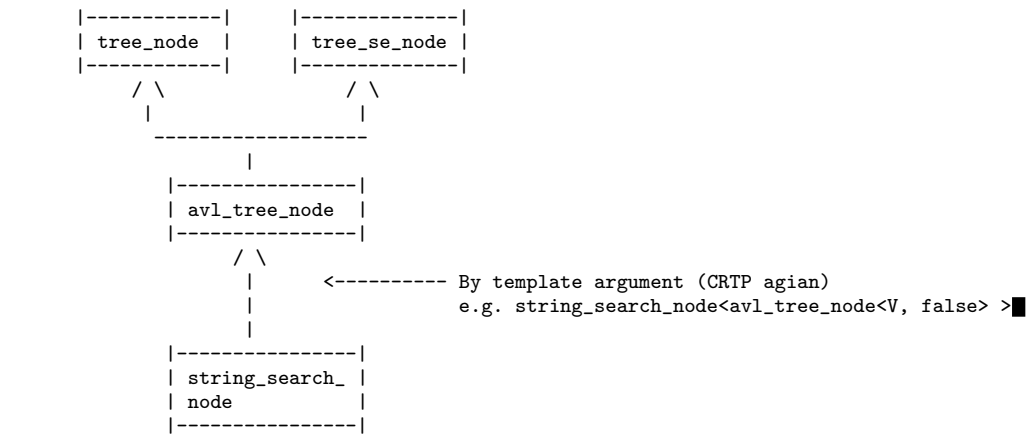
Appendix

My change logs
Search-tree-framework 30
B-Tree 35
Container classes
Map+Set/Code/stl_set_base.h++ 36
Map+Set/Code/stl_set_base.c++ 37
Map+Set/Code/stl_set.h++ 41
Map+Set/Code/stl_set.c++ 43
Map+Set/Code/stl_multiset.h++ 46
Map+Set/Code/stl_multiset.c++ 48
Map+Set/Code/stl_map_base.h++ 50
Map+Set/Code/stl_map_base.c++ 51
Map+Set/Code/stl_map.h++ 53
Map+Set/Code/stl_map.c++ 54
Map+Set/Code/stl_multimap.h++ 57
Map+Set/Code/stl_multimap.c++ 59
Safe Tree Framework
Search-tree-framework/tree.h++ 61
Search-tree-framework/tree.c++ 64
Search-tree-framework/tree_balance.h++ 81
Search-tree-framework/tree_comp.h++ 81
Search-tree-framework/tree_func.h++ 82
Search-tree-framework/tree_node.h++ 83
Search-tree-framework/tree_se_node.h++ 83
Search-tree-framework/tree_node_shared.h++ 89
Search-tree-framework/tree_proxy.h++ 89
Search-tree-framework/tree_search.h++ 91
Balance and storage policies
Search-tree-framework/aa_tree_balance.h++ 93
Search-tree-framework/aa_tree_node.h++ 95
Search-tree-framework/avl_tree_balance.h++ 96
Search-tree-framework/avl_tree_node.h++ 98
Search-tree-framework/packed_avl_tree_node.h++ ... 100
Search-tree-framework/red_black_tree_balance.h++ . 102
Search-tree-framework/red_black_tree_node.h++ 104
Search-tree-framework/packed_red_black_tree_node.h++ 105
Search-tree-framework/splay_tree_balance.h++ 107
Search-tree-framework/splay_tree_node.h++ 109
Test programs
Search-tree-framework/test.c++ 110
Search-tree-framework/iter_test.c++ 114
B-Tree
B-Tree/btree.h++ 115
B-Tree/btree_node.h++ 122
B-Tree/btree_cmp.h++ 123
B-Tree/btree_iterator.h++ 124
B-Tree/main.cpp 126
Iterator/obj_iterator.h++ 127
Iterator/obj_iterator.c++ 129

Appendix A. Notes

Appendix A.1 Search-tree-framework

```
* Thu Jan 29 04:11:19 CET 2009

Worked on AA-trees, the implementation finally works. I took
advantage of the fact that the node base class already provides
rotatation methods, this has made it really simple. 128 lines
of code including comments.

I decided not to implement searching by rank, but rather
describe in the report, why we should not provide this feature.
There is too many problems which slows down other methods
even that the user specified that he/she wanted rank search.

* Wed Jan 21 23:45:18 CET 2009

I tried to get searching by rank working, we could simply provide
an extension called select(int) like described in stroustrup et al.

But there is one big problem with this approach:

We need to maintain the size of the left tree.. This can be done
easily for insert(V const& v), but for insert(iterator, V const& v)
it is not really possible, since we should do it in O(1), we don't really
do this, but we can't guarantee that we our balancing operations
reach the node, such that we can not ensure left_size is updated
consistently. Will we pay that price? With red-black tree we rarely
reach the root when fixing up, but we are forced to do that if we
will provide search-by-rank. Of course we can let the user choose
if he want rank search and there by more expensive erase/insert
operations.

I think I'll make this feature it an option to the user

* Tue Jan 20 00:26:29 CET 2009

I've spend the last week testing the implementation and fixing
bug. The one which really annoyed be was when erasing more than
a million elements, and it was caused by the following declaration

void erase(key_type const& k) {
  ...
  nodes_type arr[node_count];
}

Since the array is "allocated" on compile-time the compiler may
use a fixed constant to determine the size. And that fixed constant
is less than a million.

The solution is to allocate the array on the heap, and it is done
for e.g. copy construction (it was obsolote for erase), but it
is still missing in insert(I, I).

Today, I untangled the searching, the approach by bjarne et al. on
simply using inheritance of the entire tree class did not fall
into my taste. I reduced the searching to just two methods,
one for lower_bound (and there by erase), upper_bound, and one
for insert (for both multiset and set).
These two methods are put into a search strategy which is like the
balancing strategy given as template argument
The advanced search
method which et described by bjarne et al can be implemented now,
with the associated node class. Which is yet another abstraction.
```



The last issue in this project is how to obtain searching by
rank. We could provide random-access iterators, with a guaranteed
complexity for iterator operations within O(lg n).

* Tue Jan 13 00:21:34 CET 2009

I've now implemented by ideas as described in the
previous post. The user can now request a
space efficient version of a node class by typing:

```
avl_tree_node<int, true>
```

And the non-space efficent by leaving out the Boolean
value of setting it to false. When the space efficient
iterator operations is done in O(lg n), and when the
non-space efficient version is used they are done in
O(1). This has also affected time complexity for
relink_for_erase, since it rely on iterator operations,
and for multiset the nodes are present in the tree when
the space efficient version is used. This is basically
the price for using space efficient nodes.

But all this is transparant to the user, he/she should
use the container in the normal way.

I also did some changes to how data inside the nodes is
accessed. The problem of using

```
node_type*& parent();
bool& colour;
```

occurs when doing bit packing, which we should also provide
with respect to space efficiency. We can not rely on references
since the value is packed, therefore I used the following
idiom:

```
node_type* parent();
void parent(node_type*);
```

* Sat Jan 10 23:26:36 CET 2009

In the paper by Katajainen & Simonsen, it is recommended that
every container provide different space efficiency.
For red-black trees it is 3-, 4-, 5- and 6 words, for AVL
atleast 4- and 6 words is desired.

Today I looked into details of how this can be done,
in terms of design of course.

To provide 4 words we leave out the linked list, such
that we use a has_list constant in the node class,
and if the particular node class has support for the
linked list this is defined. Every list operations
then checks if the associated node class defines this,
if it don't the operation is not performed.

* Wed Jan 7 02:45:58 CET 2009

Today, I implemented the bridge classes for set, multiset,
map, and multimap from stracht where each class uses inheritance.

New changes to the standard will be easily implemented e.g. cbegin() is
in the new C++0x standard, and it should be implemented only in
set_base. What we really should have was a container_base, which contained
iterators, ..., since _all_ our containers in CPH STL provide iterators!

* Thu Jan 1 23:59:37 CET 200

Finally got the new multiset solution working (as
described earlier in the notes).

When insertion we need to take care of two things:

% Check if we are inserting a duplicate, we will not
  store it in the tree (the standard says that c.find(x)
  should return the first element of x, so this works
  fine).
% If we are inserting a element which predecessor is
  a duplicate we will search to the last duplicate
  and insert it after that one. This causes a complexity
  overhead which could be fixed by having a pointer
  in the node stored in the three which points to the
  last duplicate element. This would give us insertion
  in constant time (non-amortized) for duplicates.

What is missing right now is to get the copy construction
mechanism to support the idea given above.

* Tue Dec 30 23:47:36 CET 2008

Hmm.. I thought I figured out the solution for the problem
of not comparing the value of the sentinel. I thought the

```
following:

  template <typename V>
  class tree_node {
  public:
    tree_node<V> *_left, *_right, ...;
  };

  template <typename V>
  class avl_tree_node : public tree_node<V> {
  public:
    short _balance;
    V _data;

  };

And the sentinel should be of tree_node and the rest
of avl_tree_node. But it seems harder to get working
than expected. Need to figure out another solution.

* Tue Dec 30 14:17:26 CET 2008

Simplyfied the code a bit..

* Tue Dec 30 10:18:20 CET 2008

Worked on a better multiset solution, the idea is not to keep
duplicates in the tree since it could result in a high tree
and thereby slow access time.

E.g a tree storing 5,5,5,7

      7
     /
    5

However the linked list will look like

  5 <-> 5 <-> 5 <-> 7

It should work, and it is similar to the implementation of AA-trees,
but as far as I remember do they keep an extra list for duplicates,
and this complicates the solution even more when performing
iterator operations.

The node we should store in the tree is the first node,
since s.find(5) should give the first element, and this
is found while traversing the tree.

This means when we are inserting we will append the duplicate node
to the last existing duplicate, I think we need an extra pointer to
the last existing duplicate, else we gain a performance overhead.
We will result in O(n) worst case for insertion, but with an extra
pointer we can do O(1).

(Consider the following example, we found 5, and we will use 2
 advance operations before we reach the node where the new duplicate
 should be inserted)

Recall that the new solution is not involving implementation
specific code (for AVL or red-black or ..).

* Fri Dec 26 00:32:26 CET 2008

Cleaned up the insert_dispatch(position, v) methods, could be nice
```

```
to avoid the redudant code.

Rewrote parts of insert(v) for set, we take advantage of the linked list,
the code is not smaller but more readable indeed. Rewrote some of the
_insert function.

* Thu Dec 25 16:39:22 CET 2008

I experimented with the relink_for_erase method from Frej et al.,
but I could not get it working. It seems like the code is inmature
and not well tested. However, I will try with the approach from
the AVL implementation.

Seems like it is better, but I definitely need to test it even
more.

Implementation specific code has been reduced to what I think is a
minimum:

  Thu Dec 25 16:47 [bo@sigma] [~/CPHSTL/Release/Safe-Tree-Framework] $ cat avl_tree_balance.h++ |wc -l
  121
  Thu Dec 25 16:47 [bo@sigma] [~/CPHSTL/Release/Safe-Tree-Framework] $ cat red_black_tree_balance.h++ |wc -l
  117

We can properly do even better with s-play, which is the next
challenge, since we will need to rebalance during traversal.

I think I will go with the approach by Stroustrup et al.

* Wed Dec 24 12:31:17 CET 2008

I used the Safe AVL implementation as a base for the framework.
The state of this implementation was good, worked with multiset,
and also tested with map.
I decoupled the balance mechanism from the base, such that no
implementation related details are left in the base.

My observations so far:
* Operations related to obtaining strong exception safety is only
  relevant for the base, such that an implementor of a new balancing
  mechanism should not care about that. Stroustrup et al. did not
  make this observation, I think it is quite important.
* The associated data structure i.e. the linked list is handled directly
  in the base, such that every implementation will provide iterator
  operations in O(1). The implementor should still implement successor(),
  predecessor() and content().
* Copy construction can be done efficiently (O(n) non-amortized)
  in a generic way, such that it is not dependent of the implementation.
  (this has been rewritten)
* Complicated stuff like dispatching are also independent of the
  balancing mechanism, it is and should be implemented _once_.
* Because of the associated data structure leftmost(), rightmost() is not
  needed anymore (our linked list is circular).

  leftmost() <-> (*header).successor()
  rightmost() <-> (*header).predecessor()
* Improved overview of the implementation specific details

The rotation mechanism is also moved outside the balancing strategy
to a class class generic_rotation_strategy. This has been done because
it has caused problems in e.g. succint trees.

erase does not work for red-black tree right now, because the
relink_for_erase method is not written in a generic way
but it should be possible. Frej & co, uses the linked list
```

```
for finding the successor. I guess we gotta go with this approach,
the only real problem is that the colour / balance should be
adjusted in erase_fixhup, by giving this function the
new_node too, should fix it.

Also, the base class needs to be cleaned-up and some of the very
small methods has to go.
```

Appendix A.2 B-tree

```
* Sat Jan 10 01:23:00 CET 2009

I ported the solution to the new CPH STL infrastructure, such that it atleast
works with the set bridge class.

This involved:

  * Rewriting of nodes, this involves that it is possible now to give the key
    list, which should be of the vector type as template parameter. It is very
    annoying that it is not possible to give the child_list too, but it involves
    template <typename ...,
              typename CL = std::vector<N> >
    class N {

    };
    Which I can't figure out a solution for..

  * Rewriting of the "kernel", alot of methods could be removed, since they
    are implemented directly in the scope of the bridge classes. Also, I rewrote
    the template parameters such that they are similar to e.g. safe-tree-framework.

  * Wrote the concrete iterator. The node iterator does not work here, since
    the iterator should store a pointer to a node AND a iterator to the key
    vector. I made a new iterator kind called obj_iterator which uses a
    concrete iterator directly to call successor(), ... This means for one
    abstract iterator there is also one concrete iterator.

Of significant problems with this implementation there is:

  * Iterator validity! It is impossible to keep iterators valid even with
    the use of the iterator-valid dynamic array as key_list. This happens
    because split/merge nodes does invalidate the iterators. However,
    with pointer semantics we can solve this, but this would then mean
    that we can not use the standard vector implementation.

    Pointer semantics is desired in the CPH STL, so this would be a good
    idea to make support for pointer semantics in iv-dynamic array
    ( there is already some ), and then leave out the possibility of using
    std::vector for the implementation.

    However this gives us performance issues. According to my project
    on iterator validity the iterator valid dynamic array is significant
    slower than the normal dynamic array, so we should provide two versions
    of the B-tree, one which is unsafe, which means it is fast but does
    not provide iterator validity, and another one which is safe but it
    provides iterator validity.

    On the other hand we may gain performance when merging / splitting
    nodes since we don't use expensive allocator operations (allocate/deallocate).
```

Appendix B. Container classes

Appendix B.1 stl_set_base.h++

```
#ifndef _CPHSTL_STL_BASE_H
#define _CPHSTL_STL_BASE_H

#include <functional> // defines std::less
#include <iterator> // defines std::reverse_iterator
#include <memory> // defines std::allocator
#include <utility> // defines std::pair
#include "set_helper.c++" // defines cphstl::unnamed::{identity and
    key_map}

namespace cphstl {

  template <
    typename K,
    typename V,
    typename C,
    typename A,
    typename R,
    typename I,
    typename J
  >
  class set_base {

  public:
    // types

    typedef K key_type;
    typedef V value_type;
    typedef C key_compare;
    typedef C value_compare;
    typedef A allocator_type;
    typedef V& reference;
    typedef V const& const_reference;
    typedef I iterator;
    typedef J const_iterator;
    typedef std::size_t size_type;
    typedef std::ptrdiff_t difference_type;
    typedef V* pointer;
    typedef V const* const_pointer;
    typedef std::reverse_iterator<iterator> reverse_iterator;
    typedef std::reverse_iterator<const_iterator>
        const_reverse_iterator;

    // structors

    explicit set_base(C const& = C(), A const& = A());

    template <typename X>
    set_base(X, X, C const& = C(), A const& = A());

    set_base(set_base const&);
    ~set_base();
    set_base& operator=(set_base const&);

    // iterators

    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
```

```
    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;

    // accessors

    A get_allocator() const;
    C key_comp() const;
    C value_comp() const;
    bool empty() const;
    size_type size() const;
    size_type max_size() const;
    iterator find(K const&);
    const_iterator find(K const&) const;
    iterator lower_bound(K const&) const;
    iterator upper_bound(K const&) const;
    std::pair<iterator, iterator> equal_range(K const&) const;

    size_type erase(K const&);
    void erase(iterator);
    void erase(iterator, iterator);
    void clear();
    void swap(set_base&);

  protected:

    typedef R realization_type;
    realization_type kernel;

  };

}

#include "stl_set_base.c++"
#endif
```

Appendix B.2 stl_set_base.c++

```
/*
  A set_base is a bridge class that calls the functions available in
      the
  given realization.

  Author: Jyrki Katajainen, May 2007, March 2008
*/

namespace cphstl {

  // explicit constructor

  template <typename K, typename V, typename C, typename A, typename R
      , typename I, typename J>
  set_base<K, V, C, A, R, I, J>::set_base(C const& comparator, A const
      & allocator)
    : kernel(comparator, allocator) {
  }

  // parametrized constructor

  template <typename K, typename V, typename C, typename A, typename R
      , typename I, typename J>
  template <typename X>
  set_base<K, V, C, A, R, I, J>::set_base(X p, X q, C const&
      comparator, A const& allocator)
```

```
  : kernel(comparator, allocator) {
  (*this).kernel.insert(p, q);
}

// copy constructor

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
set_base<K, V, C, A, R, I, J>::set_base(set_base const& other) :
    kernel(other.kernel) {
}

// destructor

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
set_base<K, V, C, A, R, I, J>::~set_base() {
}

// operator=

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
set_base<K, V, C, A, R, I, J>&
set_base<K, V, C, A, R, I, J>::operator=(set_base const& other) {
  (*this).kernel = other.kernel;
  return *this;
}

// begin

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
typename set_base<K, V, C, A, R, I, J>::iterator
set_base<K, V, C, A, R, I, J>::begin() {
  return iterator((*this).kernel.begin());
}

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
typename set_base<K, V, C, A, R, I, J>::const_iterator
set_base<K, V, C, A, R, I, J>::begin() const {
  return const_iterator((*this).kernel.begin());
}

// end

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
typename set_base<K, V, C, A, R, I, J>::iterator
set_base<K, V, C, A, R, I, J>::end() {
  return iterator((*this).kernel.end());
}

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
typename set_base<K, V, C, A, R, I, J>::const_iterator
set_base<K, V, C, A, R, I, J>::end() const {
  return const_iterator((*this).kernel.end());
}

// rbegin

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
```

```
typename set_base<K, V, C, A, R, I, J>::reverse_iterator
set_base<K, V, C, A, R, I, J>::rbegin() {
  return reverse_iterator((*this).end());
}

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
typename set_base<K, V, C, A, R, I, J>::const_reverse_iterator
set_base<K, V, C, A, R, I, J>::rbegin() const {
  return const_reverse_iterator((*this).end());
}

// rend

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
typename set_base<K, V, C, A, R, I, J>::reverse_iterator
set_base<K, V, C, A, R, I, J>::rend() {
  return reverse_iterator((*this).begin());
}

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
typename set_base<K, V, C, A, R, I, J>::const_reverse_iterator
set_base<K, V, C, A, R, I, J>::rend() const {
  return const_reverse_iterator((*this).begin());
}

// get allocator

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
A
set_base<K, V, C, A, R, I, J>::get_allocator() const {
  return (*this).kernel.get_allocator();
}

// get comparator

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
typename set_base<K, V, C, A, R, I, J>::key_compare
set_base<K, V, C, A, R, I, J>::key_comp() const {
  return (*this).kernel.key_comp();
}

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
typename set_base<K, V, C, A, R, I, J>::value_compare
set_base<K, V, C, A, R, I, J>::value_comp() const {
  return (*this).kernel.value_comp();
}

// empty

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
bool
set_base<K, V, C, A, R, I, J>::empty() const {
  return (*this).size() == size_type(0);
}

// size
```

```
template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
typename set_base<K, V, C, A, R, I, J>::size_type
set_base<K, V, C, A, R, I, J>::size() const {
  return (*this).kernel.size();
}

// max_size

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
typename set_base<K, V, C, A, R, I, J>::size_type
set_base<K, V, C, A, R, I, J>::max_size() const {
  return (*this).kernel.max_size();
}

// find

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
typename set_base<K, V, C, A, R, I, J>::iterator
set_base<K, V, C, A, R, I, J>::find(K const& k) {
  iterator p = (*this).lower_bound(k);
  cphstl::unnamed::key_map<K, V> key;
  key_compare comp = (*this).key_comp();

  if(p == (*this).end() || comp(k, key[*p])) {
    return (*this).end();
  }

  return p;
}

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
typename set_base<K, V, C, A, R, I, J>::const_iterator
set_base<K, V, C, A, R, I, J>::find(K const& k) const {
  const_iterator p = (*this).lower_bound(k);
  cphstl::unnamed::key_map<K, V> key;
  key_compare comp = (*this).key_comp();

  if(p == (*this).end() || comp(k, key[*p])) {
    return (*this).end();
  }

  return p;
}

// lower_bound

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
typename set_base<K, V, C, A, R, I, J>::iterator
set_base<K, V, C, A, R, I, J>::lower_bound(K const& k) const {
  return (*this).kernel.lower_bound(k);
}

// upper_bound

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
typename set_base<K, V, C, A, R, I, J>::iterator
set_base<K, V, C, A, R, I, J>::upper_bound(K const& k) const {
  return (*this).kernel.upper_bound(k);
}
```

```
  // equal_range

  template <typename K, typename V, typename C, typename A, typename R
      , typename I, typename J>
  std::pair<typename set_base<K, V, C, A, R, I, J>::iterator,
            typename set_base<K, V, C, A, R, I, J>::iterator>
  set_base<K, V, C, A, R, I, J>::equal_range(K const& k) const {
    return std::make_pair((*this).lower_bound(k), (*this).upper_bound(
        k));
  }

  // single-element erase

  template <typename K, typename V, typename C, typename A, typename R
      , typename I, typename J>
  typename set_base<K, V, C, A, R, I, J>::size_type
  set_base<K, V, C, A, R, I, J>::erase(K const& e) {
    return (*this).kernel.erase(e);
  }

  template <typename K, typename V, typename C, typename A, typename R
      , typename I, typename J>
  void
  set_base<K, V, C, A, R, I, J>::erase(iterator p) {
    (*this).kernel.erase(p);
  }

  // multiple-element erase

  template <typename K, typename V, typename C, typename A, typename R
      , typename I, typename J>
  void
  set_base<K, V, C, A, R, I, J>::erase(iterator p, iterator s) {
    for (iterator r = p; r != s;) {
      iterator q = r;
      ++r;
      (*this).kernel.erase(q);
    }
  }

  // clear

  template <typename K, typename V, typename C, typename A, typename R
      , typename I, typename J>
  void
  set_base<K, V, C, A, R, I, J>::clear() {
    (*this).erase((*this).begin(), (*this).end());
  }

  // swap

  template <typename K, typename V, typename C, typename A, typename R
      , typename I, typename J>
  void
  set_base<K, V, C, A, R, I, J>::swap(set_base<K, V, C, A, R, I, J>& r
      ) {
    (*this).kernel.swap(r.kernel);
  }

}
```

Appendix B.3 stl_set.h++

```
/*
```

```
  This header is taken quite directly from the C++ standard [2003].
  According to the terminology of the standard, a set satisfies all of
  the requirements of a container, of a reversible container, and of
  an associative container; and provides bidirectional iterators to
  the elements stored.

  Author: Jyrki Katajainen, May 2007, March 2008
*/

#ifndef __CPHSTL_SET__
#define __CPHSTL_SET__

#include <set>
#include "stl_set_base.h++"

namespace cphstl {

  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 set : public set_base<V, V, C, A, R, I, J> {

  public:
    // types

    typedef V key_type;
    typedef V value_type;
    typedef C key_compare;
    typedef C value_compare;
    typedef A allocator_type;
    typedef V& reference;
    typedef V const& const_reference;
    typedef I iterator;
    typedef J const_iterator;
    typedef std::size_t size_type;
    typedef std::ptrdiff_t difference_type;
    typedef V* pointer;
    typedef V const* const_pointer;
    typedef std::reverse_iterator<iterator> reverse_iterator;
    typedef std::reverse_iterator<const_iterator>
        const_reverse_iterator;

    // structors

    explicit set(C const& = C(), A const& = A());

    template <typename K>
    set(K, K, C const& = C(), A const& = A());

    set(set const&);
    ~set();

    set& operator=(set const&);

    size_type count(V const&) const;

    // modifiers

    std::pair<iterator, bool> insert(V const&);
    iterator insert(iterator, V const&);
```

```
    template <typename K>
    void insert(K, K);

    operator std::string() const;
  };

  template <typename V, typename C, typename A, typename R, typename I
      , typename J>
  void swap(set<V, C, A, R, I, J>&, set<V, C, A, R, I, J>&);

  template <typename V, typename C, typename A, typename R, typename I
      , typename J>
  bool operator==(set<V, C, A, R, I, J> const&, set<V, C, A, R, I, J>
      const&);

  template <typename V, typename C, typename A, typename R, typename I
      , typename J>
  bool operator<(set<V, C, A, R, I, J> const&, set<V, C, A, R, I, J>
      const&);

  template <typename V, typename C, typename A, typename R, typename I
      , typename J>
  bool operator!=(set<V, C, A, R, I, J> const&, set<V, C, A, R, I, J>
      const&);

  template <typename V, typename C, typename A, typename R, typename I
      , typename J>
  bool operator>(set<V, C, A, R, I, J> const&, set<V, C, A, R, I, J>
      const&);

  template <typename V, typename C, typename A, typename R, typename I
      , typename J>
  bool operator>=(set<V, C, A, R, I, J> const&, set<V, C, A, R, I, J>
      const&);

  template <typename V, typename C, typename A, typename R, typename I
      , typename J>
  bool operator<=(set<V, C, A, R, I, J> const&, set<V, C, A, R, I, J>
      const&);


}

#include "stl_set.c++" // implements cphstl::set

#endif
```

Appendix B.4 stl_set.c++

```
/*
  A set is a bridge class that calls the functions available in the
  given realization.

  Author: Jyrki Katajainen, May 2007, March 2008
*/

namespace cphstl {

  // explicit constructor

  template <typename V, typename C, typename A, typename R, typename I
      , typename J>
  set<V, C, A, R, I, J>::set(C const& comparator, A const& allocator)
    : set_base<V, V, C, A, R, I, J>(comparator, allocator) {
  }
```

```
// parametrized constructor

template <typename V, typename C, typename A, typename R, typename I
    , typename J>
template <typename K>
set<V, C, A, R, I, J>::set(K p, K q, C const& comparator, A const&
    allocator)
  : set_base<V, V, C, A, R, I, J>(p, q, comparator, allocator) {
}

// copy constructor

template <typename V, typename C, typename A, typename R, typename I
    , typename J>
set<V, C, A, R, I, J>::set(set const& other) : set_base<V, V, C, A,
    R, I, J>(other) {
}

// destructor

template <typename V, typename C, typename A, typename R, typename I
    , typename J>
set<V, C, A, R, I, J>::~set() {
}

// operator=

template <typename V, typename C, typename A, typename R, typename I
    , typename J>
set<V, C, A, R, I, J>&
set<V, C, A, R, I, J>::operator=(set const& other) {
  (*this).kernel = other.kernel;
  return *this;
}


// count

template <typename V, typename C, typename A, typename R, typename I
    , typename J>
typename set<V, C, A, R, I, J>::size_type
set<V, C, A, R, I, J>::count(V const& k) const {
  if ((*this).find(k) == (*this).end()) {
    return size_type(0);
  }
  return size_type(1);
}

// single-element insert

template <typename V, typename C, typename A, typename R, typename I
    , typename J>
std::pair<typename set<V, C, A, R, I, J>::iterator, bool>
set<V, C, A, R, I, J>::insert(V const& e) {
  return (*this).kernel.insert(e);
}

template <typename V, typename C, typename A, typename R, typename I
    , typename J>
typename set<V, C, A, R, I, J>::iterator
set<V, C, A, R, I, J>::insert(iterator p, V const& e) {
  return (*this).kernel.insert(p, e); // _unique
}
```

```
// multiple-element insert

template <typename V, typename C, typename A, typename R, typename I
    , typename J>
template <typename K>
void
set<V, C, A, R, I, J>::insert(K p, K q) {
  return (*this).kernel.insert(p, q);
}

template <typename V, typename C, typename A, typename R, typename I
    , typename J>
void
swap(set<V, C, A, R, I, J>& r, set<V, C, A, R, I, J>& s) {
  r.swap(s);
}


// operator==

template <typename V, typename C, typename A, typename R, typename I
    , typename J>
bool
operator==(set<V, C, A, R, I, J> const& r, set<V, C, A, R, I, J>
    const& s) {
  return (r.size() == s.size() && std::equal(r.begin(), r.end(), s.
      begin()));
}

// operator!=

template <typename V, typename C, typename A, typename R, typename I
    , typename J>
bool
operator!=(set<V, C, A, R, I, J> const& r, set<V, C, A, R, I, J>
    const& s) {
  return !(r == s);
}

// operator<

template <typename V, typename C, typename A, typename R, typename I
    , typename J>
bool
operator<(set<V, C, A, R, I, J> const& r, set<V, C, A, R, I, J>
    const& s) {
  return std::lexicographical_compare(r.begin(), r.end(), s.begin(),
      s.end(),
                                      r.key_comp());
}

// operator>

template <typename V, typename C, typename A, typename R, typename I
    , typename J>
bool
operator>(set<V, C, A, R, I, J> const& r, set<V, C, A, R, I, J>
    const& s) {
  return (s < r);
}

// operator<=

template <typename V, typename C, typename A, typename R, typename I
    , typename J>
```

```
  bool
  operator<=(set<V, C, A, R, I, J> const& r, set<V, C, A, R, I, J>
      const& s) {
    return !(s < r);
  }

  // operator>=

  template <typename V, typename C, typename A, typename R, typename I
      , typename J>
  bool
  operator>=(set<V, C, A, R, I, J> const& r, set<V, C, A, R, I, J>
      const& s) {
    return !(r < s);
  }

  template <typename V, typename C, typename A, typename R, typename I
      , typename J>
  set<V, C, A, R, I, J>::operator std::string() const {
    return std::string((*this).kernel);
  }

}
```

Appendix B.5 stl_multiset.h++

```
/*
  This header is taken quite directly from the C++ standard [2003].
  According to the terminology of the standard, a set satisfies all of
  the requirements of a container, of a reversible container, and of
  an associative container; and provides bidirectional iterators to
  the elements stored.

  Author: Jyrki Katajainen, May 2007, March 2008
*/

#ifndef __CPHSTL_MULTISET__
#define __CPHSTL_MULTISET__

#include <set>
#include "stl_set_base.h++"

namespace cphstl {

  template <
    typename V,
    typename C = std::less<V>,
    typename A = std::allocator<V>,
    typename R = std::multiset<V, C, A>,
    typename I = typename R::iterator,
    typename J = typename R::const_iterator
  >
  class multiset : public set_base<V, V, C, A, R, I, J> {

  public:
    // types

    typedef V key_type;
    typedef V value_type;
    typedef C key_compare;
    typedef C value_compare;
    typedef A allocator_type;
    typedef V& reference;
    typedef V const& const_reference;
    typedef I iterator;
```

```
  typedef J const_iterator;
  typedef std::size_t size_type;
  typedef std::ptrdiff_t difference_type;
  typedef V* pointer;
  typedef V const* const_pointer;
  typedef std::reverse_iterator<iterator> reverse_iterator;
  typedef std::reverse_iterator<const_iterator>
      const_reverse_iterator;

  // structors

  explicit multiset(C const& = C(), A const& = A());

  template <typename K>
  multiset(K, K, C const& = C(), A const& = A());

  multiset(multiset const&);
  ~multiset();

  multiset& operator=(multiset const&);

  size_type count(V const&) const;

  // modifiers

  iterator insert(V const&);
  iterator insert(iterator, V const&);
  template <typename K>
  void insert(K, K);

  operator std::string() const;
};

template <typename V, typename C, typename A, typename R, typename I
    , typename J>
void swap(multiset<V, C, A, R, I, J>&, multiset<V, C, A, R, I, J>&);

template <typename V, typename C, typename A, typename R, typename I
    , typename J>
bool operator==(multiset<V, C, A, R, I, J> const&, multiset<V, C, A,
     R, I, J> const&);

template <typename V, typename C, typename A, typename R, typename I
    , typename J>
bool operator<(multiset<V, C, A, R, I, J> const&, multiset<V, C, A,
    R, I, J> const&);

template <typename V, typename C, typename A, typename R, typename I
    , typename J>
bool operator!=(multiset<V, C, A, R, I, J> const&, multiset<V, C, A,
     R, I, J> const&);

template <typename V, typename C, typename A, typename R, typename I
    , typename J>
bool operator>(multiset<V, C, A, R, I, J> const&, multiset<V, C, A,
    R, I, J> const&);

template <typename V, typename C, typename A, typename R, typename I
    , typename J>
bool operator>=(multiset<V, C, A, R, I, J> const&, multiset<V, C, A,
     R, I, J> const&);

template <typename V, typename C, typename A, typename R, typename I
    , typename J>
```

```
  bool operator<=(multiset<V, C, A, R, I, J> const&, multiset<V, C, A,
       R, I, J> const&);


}

#include "stl_multiset.c++" // implements cphstl::multiset

#endif
```

Appendix B.6 stl_multiset.c++

```
/*
  A multiset is a bridge class that calls the functions available in
      the
  given realization.

  Author: Jyrki Katajainen, May 2007, March 2008
*/

namespace cphstl {

  // explicit constructor

  template <typename V, typename C, typename A, typename R, typename I
      , typename J>
  multiset<V, C, A, R, I, J>::multiset(C const& comparator, A const&
      allocator)
    : set_base<V, V, C, A, R, I, J>(comparator, allocator) {
  }

  // parametrized constructor

  template <typename V, typename C, typename A, typename R, typename I
      , typename J>
  template <typename K>
  multiset<V, C, A, R, I, J>::multiset(K p, K q, C const& comparator,
      A const& allocator)
    : set_base<V, V, C, A, R, I, J>(p, q, comparator, allocator) {
  }

  // copy constructor

  template <typename V, typename C, typename A, typename R, typename I
      , typename J>
  multiset<V, C, A, R, I, J>::multiset(multiset const& other) :
      set_base<V, V, C, A, R, I, J>(other) {
  }

  // destructor

  template <typename V, typename C, typename A, typename R, typename I
      , typename J>
  multiset<V, C, A, R, I, J>::~multiset() {
  }

  // operator=

  template <typename V, typename C, typename A, typename R, typename I
      , typename J>
  multiset<V, C, A, R, I, J>&
  multiset<V, C, A, R, I, J>::operator=(multiset const& other) {
    (*this).kernel = other.kernel;
    return *this;
  }
```

```
// count

template <typename V, typename C, typename A, typename R, typename I
    , typename J>
typename multiset<V, C, A, R, I, J>::size_type
multiset<V, C, A, R, I, J>::count(V const& k) const {
  if ((*this).find(k) == (*this).end()) {
    return size_type(0);
  }
  return size_type(std::distance((*this).lower_bound(k),
                                 (*this).upper_bound(k)));
}

// single-element insert

template <typename V, typename C, typename A, typename R, typename I
    , typename J>
typename multiset<V, C, A, R, I, J>::iterator
multiset<V, C, A, R, I, J>::insert(V const& e) {
  return (*this).kernel.insert(e);
}

template <typename V, typename C, typename A, typename R, typename I
    , typename J>
typename multiset<V, C, A, R, I, J>::iterator
multiset<V, C, A, R, I, J>::insert(iterator p, V const& e) {
  return (*this).kernel.insert(p, e); // _unique
}

// multiple-element insert

template <typename V, typename C, typename A, typename R, typename I
    , typename J>
template <typename K>
void
multiset<V, C, A, R, I, J>::insert(K p, K q) {
  return (*this).kernel.insert(p, q);
}

template <typename V, typename C, typename A, typename R, typename I
    , typename J>
void
swap(multiset<V, C, A, R, I, J>& r, multiset<V, C, A, R, I, J>& s) {
  r.swap(s);
}


// operator==

template <typename V, typename C, typename A, typename R, typename I
    , typename J>
bool
operator==(multiset<V, C, A, R, I, J> const& r, multiset<V, C, A, R,
     I, J> const& s) {
  return (r.size() == s.size() && std::equal(r.begin(), r.end(), s.
      begin()));
}

// operator!=

template <typename V, typename C, typename A, typename R, typename I
    , typename J>
bool
```

```
  operator!=(multiset<V, C, A, R, I, J> const& r, multiset<V, C, A, R,
       I, J> const& s) {
    return !(r == s);
  }

  // operator<

  template <typename V, typename C, typename A, typename R, typename I
      , typename J>
  bool
  operator<(multiset<V, C, A, R, I, J> const& r, multiset<V, C, A, R,
      I, J> const& s) {
    return std::lexicographical_compare(r.begin(), r.end(), s.begin(),
        s.end(),
                                        r.key_comp());
  }

  // operator>

  template <typename V, typename C, typename A, typename R, typename I
      , typename J>
  bool
  operator>(multiset<V, C, A, R, I, J> const& r, multiset<V, C, A, R,
      I, J> const& s) {
    return (s < r);
  }

  // operator<=

  template <typename V, typename C, typename A, typename R, typename I
      , typename J>
  bool
  operator<=(multiset<V, C, A, R, I, J> const& r, multiset<V, C, A, R,
       I, J> const& s) {
    return !(s < r);
  }

  // operator>=

  template <typename V, typename C, typename A, typename R, typename I
      , typename J>
  bool
  operator>=(multiset<V, C, A, R, I, J> const& r, multiset<V, C, A, R,
       I, J> const& s) {
    return !(r < s);
  }

  template <typename V, typename C, typename A, typename R, typename I
      , typename J>
  multiset<V, C, A, R, I, J>::operator std::string() const {
    return std::string((*this).kernel);
  }

}
```

Appendix B.7 stl_map_base.h++

```
/*
   Bridge class for cphstl::map, almost directly taken from the C++
       standard.

   Author: Bo Simonsen, December 2008
*/

#include <map>
```

```
#include "stl_set_base.h++"

#ifndef CPHSTL_MAP_BASE_H
#define CPHSTL_MAP_BASE_H

namespace cphstl {
  template <typename K,
            typename V,
            typename C = std::less<K>,
            typename A = std::allocator<std::pair<const K, V> >,
            typename R = std::map<K, V, C, A>,
            typename I = typename R::iterator,
            typename J = typename R::const_iterator
           >

  class map_base : public set_base<K, V, C, A, R, I, J> {
  public:
    typedef I iterator;
    typedef J const_iterator;
    typedef K key_type;

    // structors

    explicit map_base(C const& = C(), A const& = A());

    template <typename X>
    map_base(X, X, C const& = C(), A const& = A());

    map_base(map_base const&);
    ~map_base();

    iterator lower_bound(const  key_type& x);
    const_iterator lower_bound(const  key_type& x) const;
    iterator upper_bound(const  key_type& x);
    const_iterator upper_bound(const  key_type& x) const;

    std::pair<iterator,iterator> equal_range(const key_type& x);
    std::pair<const_iterator,const_iterator> equal_range(const
        key_type& x) const;
  };
}

#include "stl_map_base.c++"
#endif
```

Appendix B.8 stl_map_base.c++

```
namespace cphstl {
  // explicit constructor

  template <typename K, typename V, typename C, typename A, typename R
      , typename I, typename J>
  map_base<K, V, C, A, R, I, J>::map_base(C const& comparator, A const
      & allocator)
    : set_base<K, V, C, A, R, I, J>(comparator, allocator) {
  }

  // parametrized constructor

  template <typename K, typename V, typename C, typename A, typename R
      , typename I, typename J>
  template <typename X>
  map_base<K, V, C, A, R, I, J>::map_base(X p, X q, C const&
      comparator, A const& allocator)
    : set_base<K, V, C, A, R, I, J>(p, q, comparator, allocator) {
```

```
}

// copy constructor

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
map_base<K, V, C, A, R, I, J>::map_base(map_base const& other) :
    set_base<K, V, C, A, R, I, J>(other) {
}

// destructor
template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
map_base<K, V, C, A, R, I, J>::~map_base() {
}

// lower_bound

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
typename map_base<K, V, C, A, R, I, J>::iterator
map_base<K, V, C, A, R, I, J>::lower_bound(K const& k) {
  return (*this).kernel.lower_bound(k);
}

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
typename map_base<K, V, C, A, R, I, J>::const_iterator
map_base<K, V, C, A, R, I, J>::lower_bound(K const& k) const {
  return (*this).kernel.lower_bound(k);
}

// upper_bound

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
typename map_base<K, V, C, A, R, I, J>::iterator
map_base<K, V, C, A, R, I, J>::upper_bound(K const& k) {
  return (*this).kernel.upper_bound(k);
}
template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
typename map_base<K, V, C, A, R, I, J>::const_iterator
map_base<K, V, C, A, R, I, J>::upper_bound(K const& k) const {
  return (*this).kernel.upper_bound(k);
}

// equal_range

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
std::pair<typename map_base<K, V, C, A, R, I, J>::iterator,
          typename map_base<K, V, C, A, R, I, J>::iterator>
map_base<K, V, C, A, R, I, J>::equal_range(K const& k) {
  return std::make_pair((*this).lower_bound(k), (*this).upper_bound(
      k));
}

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
std::pair<typename map_base<K, V, C, A, R, I, J>::const_iterator,
          typename map_base<K, V, C, A, R, I, J>::const_iterator>
map_base<K, V, C, A, R, I, J>::equal_range(K const& k) const {
  return std::make_pair((*this).lower_bound(k), (*this).upper_bound(
      k));
```

```
  }

}
```

Appendix B.9 stl_map.h++

```
/*
   Bridge class for cphstl::map, almost directly taken from the C++
       standard.

   Author: Bo Simonsen, December 2008
*/

#include <map>
#include "stl_map_base.h++"

#ifndef CPHSTL_MAP_H
#define CPHSTL_MAP_H

namespace cphstl {
  template <typename K,
            typename V,
            typename C = std::less<K>,
            typename A = std::allocator<std::pair<const K, V> >,
            typename R = std::map<K, V, C, A>,
            typename I = typename R::iterator,
            typename J = typename R::const_iterator
           >

  class map : public map_base<K, V, C, A, R, I, J> {
  public:
    // types:
    typedef K key_type;
    typedef V mapped_type;
    typedef std::pair<const K, V> value_type;
    typedef C key_compare;
    typedef A allocator_type;
    typedef I iterator;
    typedef J const_iterator;

    typedef typename R::reference reference;
    typedef typename R::const_reference const_reference;
    typedef typename R::pointer pointer;
    typedef typename R::const_pointer const_pointer;

    typedef typename R::value_compare value_compare;

    typedef std::size_t size_type;
    typedef std::ptrdiff_t difference_type;
    typedef std::reverse_iterator<iterator> reverse_iterator;
    typedef std::reverse_iterator<const_iterator>
        const_reverse_iterator;

    explicit map(C const& = C(), A const& = A());

    template <typename X>
    map(X p, X q, C const& c = C(), A const& a = A());
    map(const map<K, V, C, A, R, I, J>& x);

    ~map();

    map<K, V, C, A, R, I, J>& operator=(const map<K, V, C, A, R, I, J
        >& x);

    mapped_type& operator[](const key_type& x);
```

```
    std::pair<iterator, bool> insert(const value_type& x);
    iterator insert(iterator p, const value_type& x);

    template <typename X>
    void insert(X p, X q);

    size_type count(const key_type& x) const;
  };

  template <typename K, typename V, typename C, typename A, typename R
      , typename I, typename J>
     bool operator==(const map<K,V,C,A,R,I,J>& x,
                          const map<K,V,C,A,R,I,J>& y);
  template <typename K, typename V, typename C, typename A, typename R
      , typename I, typename J>
    bool operator< (const map<K,V,C,A,R,I,J>& x,
                          const map<K,V,C,A,R,I,J>& y);
  template <typename K, typename V, typename C, typename A, typename R
      , typename I, typename J>
    bool operator!=(const map<K,V,C,A,R,I,J>& x,
                          const map<K,V,C,A,R,I,J>& y);
  template <typename K, typename V, typename C, typename A, typename R
      , typename I, typename J>
    bool operator> (const map<K,V,C,A,R,I,J>& x,
                          const map<K,V,C,A,R,I,J>& y);
  template <typename K, typename V, typename C, typename A, typename R
      , typename I, typename J>
    bool operator>=(const map<K,V,C,A,R,I,J>& x,
                          const map<K,V,C,A,R,I,J>& y);
  template <typename K, typename V, typename C, typename A, typename R
      , typename I, typename J>
    bool operator<=(const map<K,V,C,A,R,I,J>& x,
                          const map<K,V,C,A,R,I,J>& y);
  template <typename K, typename V, typename C, typename A, typename R
      , typename I, typename J>
    void swap(map<K,V,C,A,R,I,J>& x,
                    map<K,V,C,A,R,I,J>& y);

} // cphstl namespace

#include "stl_map.c++"
#endif
```

Appendix B.10 stl_map.c++

```
/*
   Implementation of the cphstl::map bridge class,
   most origins from the cphstl::set implementation

   Author: Bo Simonsen, December 2008
*/

namespace cphstl {
  // explicit constructor

  template <typename K, typename V, typename C, typename A, typename R
      , typename I, typename J>
  map<K, V, C, A, R, I, J>::map(C const& comparator, A const&
      allocator)
    : map_base<K, V, C, A, R, I, J>(comparator, allocator) {
  }

  // parametrized constructor
```

```
template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
template <typename X>
map<K, V, C, A, R, I, J>::map(X p, X q, C const& comparator, A const
    & allocator)
  : map_base<K, V, C, A, R, I, J>(p, q, comparator, allocator) {
}

// copy constructor

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
map<K, V, C, A, R, I, J>::map(map const& other) : map_base<K, V, C,
    A, R, I, J>(other) {
}

// destructor
template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
map<K, V, C, A, R, I, J>::~map() {
}

// operator=

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
map<K, V, C, A, R, I, J>&
map<K, V, C, A, R, I, J>::operator=(map const& other) {
  (*this).kernel = other.kernel;
  return *this;
}

// operator[]
template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
typename map<K, V, C, A, R, I, J>::mapped_type&
map<K, V, C, A, R, I, J>::operator[](key_type const& k) {
  iterator i = (*this).lower_bound(k);

  if(i == (*this).end() || (*this).key_comp()(k, (*i).first)) {
    i = (*this).insert(i, value_type(k, mapped_type()));
  }

  return (*i).second;
}


// count

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
typename map<K, V, C, A, R, I, J>::size_type
map<K, V, C, A, R, I, J>::count(K const& k) const {
  if ((*this).find(k) == (*this).end()) {
    return size_type(0);
  }
  return size_type(1);
}
// single-element insert

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
std::pair<typename map<K, V, C, A, R, I, J>::iterator, bool>
map<K, V, C, A, R, I, J>::insert(value_type const& e) {
  return (*this).kernel.insert(e);
```

```
}

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
typename map<K, V, C, A, R, I, J>::iterator
map<K, V, C, A, R, I, J>::insert(iterator p, value_type const& e) {
  return (*this).kernel.insert(p, e); // _unique
}

// multiple-element insert

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
template <typename X>
void
map<K, V, C, A, R, I, J>::insert(X p, X q) {
  (*this).kernel.insert(p, q); // _unique
}


template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
void
swap(map<K, V, C, A, R, I, J>& r, map<K, V, C, A, R, I, J>& s) {
  r.swap(s);
}

/* Casting operator for printing out the realisation data structure

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
map<K, V, C, A, R, I, J>::operator std::string() const {
  return std::string((*this).kernel);
} */

// operator==

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
bool
operator==(map<K, V, C, A, R, I, J> const& r, map<K, V, C, A, R, I,
    J> const& s) {
  return (r.size() == s.size() && std::equal(r.begin(), r.end(), s.
      begin()));
}

// operator!=

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
bool
operator!=(map<K, V, C, A, R, I, J> const& r, map<K, V, C, A, R, I,
    J> const& s) {
  return !(r == s);
}

// operator<

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
bool
operator<(map<K, V, C, A, R, I, J> const& r, map<K, V, C, A, R, I, J
    > const& s) {
  return std::lexicographical_compare(r.begin(), r.end(), s.begin(),
       s.end(),
```

```
                                        r.key_comp());
  }

  // operator>

  template <typename K, typename V, typename C, typename A, typename R
      , typename I, typename J>
  bool
  operator>(map<K, V, C, A, R, I, J> const& r, map<K, V, C, A, R, I, J
      > const& s) {
    return (s < r);
  }

  // operator<=

  template <typename K, typename V, typename C, typename A, typename R
      , typename I, typename J>
  bool
  operator<=(map<K, V, C, A, R, I, J> const& r, map<K, V, C, A, R, I,
      J> const& s) {
    return !(s < r);
  }

  // operator>=

  template <typename K, typename V, typename C, typename A, typename R
      , typename I, typename J>
  bool
  operator>=(map<K, V, C, A, R, I, J> const& r, map<K, V, C, A, R, I,
      J> const& s) {
    return !(r < s);
  }


}
```

Appendix B.11 stl_multimap.h++

```
/*
   Bridge class for cphstl::map, almost directly taken from the C++
       standard.

   Author: Bo Simonsen, December 2008
*/

#include <map>
#include "stl_map_base.h++"

#ifndef CPHSTL_MAP_H
#define CPHSTL_MAP_H

namespace cphstl {
  template <typename K,
            typename V,
            typename C = std::less<K>,
            typename A = std::allocator<std::pair<const K, V> >,
            typename R = std::map<K, V, C, A>,
            typename I = typename R::iterator,
            typename J = typename R::const_iterator
           >

  class map : public map_base<K, V, C, A, R, I, J> {
  public:
    // types:
    typedef K key_type;
```

```
  typedef V mapped_type;
  typedef std::pair<const K, V> value_type;
  typedef C key_compare;
  typedef A allocator_type;
  typedef I iterator;
  typedef J const_iterator;

  typedef typename R::reference reference;
  typedef typename R::const_reference const_reference;
  typedef typename R::pointer pointer;
  typedef typename R::const_pointer const_pointer;

  typedef typename R::value_compare value_compare;

  typedef std::size_t size_type;
  typedef std::ptrdiff_t difference_type;
  typedef std::reverse_iterator<iterator> reverse_iterator;
  typedef std::reverse_iterator<const_iterator>
      const_reverse_iterator;

  explicit map(C const& = C(), A const& = A());

  template <typename X>
  map(X p, X q, C const& c = C(), A const& a = A());
  map(const map<K, V, C, A, R, I, J>& x);

  ~map();

  map<K, V, C, A, R, I, J>& operator=(const map<K, V, C, A, R, I, J
      >& x);

  iterator insert(const value_type& x);
  iterator insert(iterator p, const value_type& x);

  template <typename X>
  void insert(X p, X q);

  size_type count(const key_type& x) const;
};

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
   bool operator==(const map<K,V,C,A,R,I,J>& x,
                           const map<K,V,C,A,R,I,J>& y);
template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
  bool operator< (const map<K,V,C,A,R,I,J>& x,
                           const map<K,V,C,A,R,I,J>& y);
template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
  bool operator!=(const map<K,V,C,A,R,I,J>& x,
                           const map<K,V,C,A,R,I,J>& y);
template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
  bool operator> (const map<K,V,C,A,R,I,J>& x,
                           const map<K,V,C,A,R,I,J>& y);
template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
  bool operator>=(const map<K,V,C,A,R,I,J>& x,
                           const map<K,V,C,A,R,I,J>& y);
template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
  bool operator<=(const map<K,V,C,A,R,I,J>& x,
                           const map<K,V,C,A,R,I,J>& y);
```

```
  template <typename K, typename V, typename C, typename A, typename R
      , typename I, typename J>
    void swap(map<K,V,C,A,R,I,J>& x,
                  map<K,V,C,A,R,I,J>& y);

} // cphstl namespace

#include "stl_map.c++"
#endif
```

Appendix B.12 stl_multimap.c++

```
/*
   Implementation of the cphstl::multimap bridge class,
   most origins from the cphstl::set implementation

   Author: Bo Simonsen, December 2008
*/

namespace cphstl {
  // explicit constructor

  template <typename K, typename V, typename C, typename A, typename R
      , typename I, typename J>
  multimap<K, V, C, A, R, I, J>::multimap(C const& comparator, A const
      & allocator)
    : map_base<K, V, C, A, R, I, J>(comparator, allocator) {
  }

  // parametrized constructor

  template <typename K, typename V, typename C, typename A, typename R
      , typename I, typename J>
  template <typename X>
  multimap<K, V, C, A, R, I, J>::multimap(X p, X q, C const&
      comparator, A const& allocator)
    : map_base<K, V, C, A, R, I, J>(p, q, comparator, allocator) {
  }

  // copy constructor

  template <typename K, typename V, typename C, typename A, typename R
      , typename I, typename J>
  multimap<K, V, C, A, R, I, J>::multimap(multimap const& other) :
      map_base<K, V, C, A, R, I, J>(other) {
  }

  // destructor
  template <typename K, typename V, typename C, typename A, typename R
      , typename I, typename J>
  multimap<K, V, C, A, R, I, J>::~multimap() {
  }

  // operator=

  template <typename K, typename V, typename C, typename A, typename R
      , typename I, typename J>
  multimap<K, V, C, A, R, I, J>&
  multimap<K, V, C, A, R, I, J>::operator=(multimap const& other) {
    (*this).kernel = other.kernel;
    return *this;
  }

  // count
```

```
template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
typename multimap<K, V, C, A, R, I, J>::size_type
multimap<K, V, C, A, R, I, J>::count(K const& k) const {
  if ((*this).find(k) == (*this).end()) {
    return size_type(0);
  }
  return size_type(std::distance((*this).lower_bound(k),
                                 (*this).upper_bound(k)));

}
// single-element insert

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
typename multimap<K, V, C, A, R, I, J>::iterator
multimap<K, V, C, A, R, I, J>::insert(value_type const& e) {
  return (*this).kernel.insert(e);
}

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
typename multimap<K, V, C, A, R, I, J>::iterator
multimap<K, V, C, A, R, I, J>::insert(iterator p, value_type const&
    e) {
  return (*this).kernel.insert(p, e); // _unique
}

// multiple-element insert

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
template <typename X>
void
multimap<K, V, C, A, R, I, J>::insert(X p, X q) {
  (*this).kernel.insert(p, q); // _unique
}


template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
void
swap(multimap<K, V, C, A, R, I, J>& r, multimap<K, V, C, A, R, I, J
    >& s) {
  r.swap(s);
}

/* Casting operator for printing out the realisation data structure

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
multimap<K, V, C, A, R, I, J>::operator std::string() const {
  return std::string((*this).kernel);
} */

// operator==

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
bool
operator==(multimap<K, V, C, A, R, I, J> const& r, multimap<K, V, C,
     A, R, I, J> const& s) {
  return (r.size() == s.size() && std::equal(r.begin(), r.end(), s.
      begin()));
}
```

```
  // operator!=

  template <typename K, typename V, typename C, typename A, typename R
      , typename I, typename J>
  bool
  operator!=(multimap<K, V, C, A, R, I, J> const& r, multimap<K, V, C,
       A, R, I, J> const& s) {
    return !(r == s);
  }

  // operator<

  template <typename K, typename V, typename C, typename A, typename R
      , typename I, typename J>
  bool
  operator<(multimap<K, V, C, A, R, I, J> const& r, multimap<K, V, C,
      A, R, I, J> const& s) {
    return std::lexicographical_compare(r.begin(), r.end(), s.begin(),
        s.end(),
                                        r.key_comp());
  }

  // operator>

  template <typename K, typename V, typename C, typename A, typename R
      , typename I, typename J>
  bool
  operator>(multimap<K, V, C, A, R, I, J> const& r, multimap<K, V, C,
      A, R, I, J> const& s) {
    return (s < r);
  }

  // operator<=

  template <typename K, typename V, typename C, typename A, typename R
      , typename I, typename J>
  bool
  operator<=(multimap<K, V, C, A, R, I, J> const& r, multimap<K, V, C,
       A, R, I, J> const& s) {
    return !(s < r);
  }

  // operator>=

  template <typename K, typename V, typename C, typename A, typename R
      , typename I, typename J>
  bool
  operator>=(multimap<K, V, C, A, R, I, J> const& r, multimap<K, V, C,
       A, R, I, J> const& s) {
    return !(r < s);
  }


}
```

Appendix C. Safe Tree Framework

Appendix C.1 tree.h++

```
/*

  Header file for the tree framework
```

```
  Authors: Bo Simonsen, May 2008, December 2008

*/

#ifndef _CPHSTL_TREE_H
#define _CPHSTL_TREE_H

#include <functional>
#include <algorithm>
#include <memory>
#include <iostream>
#include <stack>
#include "assert.h++"
#include "type.h++"
#include "tree_comp.h++"
#include "tree_search.h++"

#include "red_black_tree_node.h++"
#include "red_black_tree_balance.h++"

#include "tree_proxy.h++"

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 = cphstl::red_black_tree_node<V>,
    typename B = cphstl::red_black_tree_balance_policy<
        red_black_tree_node<V> >,
    bool is_multiset = false,
    typename S = cphstl::tree_search<K, V, F, comparator_proxy<C>, N,
        B, is_multiset>
      >
  class tree
  {
  private:
    typedef N node_type;
    typedef F functor_type;
  public:
    typedef K key_type;
    typedef V value_type;
    typedef C key_compare;

    typedef typename cphstl::if_then_else<cphstl::types<K, V>::
        are_same, key_compare, _value_compare<key_type, value_type,
        key_compare> >::type value_compare;
    typedef typename A::template rebind<node_type>::other
        allocator_type;
    typedef typename A::template rebind<node_type*>::other
        array_allocator_type;

    typedef size_t size_type;
    typedef node_type* concrete_iterator;

  public:
    /* Constructors */
    explicit tree(const C&, const A&);
    tree(const tree&);
    ~tree();

    /* Operators */
```

```
  tree& operator=(const tree&);

  /* Size accessors */
  size_type size() const;
  size_type max_size() const;

  /* Policy accessors */
  allocator_type get_allocator() const;
  key_compare key_comp() const;
  value_compare value_comp() const;

  /* Accessing and manipulating methods */
  node_type* lower_bound(const key_type&) const;
  node_type* upper_bound(const key_type&) const;

  node_type* begin() const;
  node_type* end() const;
  //non-const variants missing!

  typename cphstl::if_then_else<is_multiset,
                        node_type*,
                        std::pair<node_type*, bool>
                       >::type
  insert(value_type const&);

  node_type* insert(node_type*, value_type const&);

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

  void erase(node_type*);
  size_type erase(key_type const&);

  void swap(tree&);
  operator std::string() const;

private:
  void destroy_tree(node_type*);
  void init();

  void duplicate_swap(node_type*);
  void normal_erase(node_type*);

  node_type* copy_nodes(node_type*, int);
  void copy(const tree&);
  node_type* build_tree(node_type*, node_type*[], int&);

  node_type* _insert(node_type*, const value_type&, node_type*,
      node_type*, bool);
  node_type* relink_for_erase(node_type*);

  /* Dispatching methods - set */
  std::pair<node_type*, bool> insert_dispatch(const value_type&,
      cphstl::bool2type<false>);
  node_type* insert_dispatch(node_type*, const value_type&, cphstl::
      bool2type<false>);
  template <typename I>
  void insert_dispatch(I, I, cphstl::bool2type<false>);

  /* Dispatching methods - multiset */
  node_type* insert_dispatch(const value_type&, cphstl::bool2type<
      true>);
  node_type* insert_dispatch(node_type*, const value_type&, cphstl::
      bool2type<true>);
  template <typename I>
```

```
    void insert_dispatch(I, I, cphstl::bool2type<true>);

  private:
    comparator_proxy<key_compare> comparator;
    allocator_proxy<allocator_type> allocator;
    allocator_proxy<array_allocator_type> array_allocator;
    node_type* header;
    size_type node_count;

  };
} // namespace cphstl

#include "tree.c++"

#endif // CPHSTL_AVL_HPP
```

Appendix C.2 tree.c++

```
/*

  Implementation of the tree framework

  Original source: the AVL implementation by Stephan Lynge (CPHSTL
      report 2004-1)

  Authors: Bo Simonsen, May 2008, December 2008

  Important details:

  Tree                  Circular linked list inside the tree
  ------------------------------------------------------------

         5
        / \             5 <-> 7 <-> 9 <-> header
       7   9            ^--------------------^

  The linked list is in sorted order to obtain constant time
      complexity for
  iterator operations.

  The sentinel / header node has the following properties

  (*header).parent() = Root node of the tree
  (*header).left() / (*header).right() = Not in use
  (*header).predecessor() = Right most node
  (*header).successor() = Left most node

  The means begin() gives (*header).successor() and end() gives header
      .

  For multiset, the duplicate nodes are only kept in the linked list e
      .g.
  an instance of multiset storing {5, 5, 5, 7}.


  Tree            Linked list
  ------------------------------------------------------------

      5           5 <-> 5 <-> 5* <-> 7 <-> header
     /            ^--------------------------^
    7

  This means when inserting a duplicate we insert it between the last
      existing
```

```
  duplicate and the next unique element. E.g. a duplicate of 5 is
      inserted at
  the *-mark.
*/

namespace cphstl {

  /* (C, A) constructor */

  template <typename K, typename V, typename F, typename C, typename A
      , typename N, typename B, bool is_multiset, typename S>
  tree<K, V, F, C, A, N, B, is_multiset, S>::tree(const C& k, const A&
       a) {

    (*this).comparator = k;
    (*this).allocator = a;

    (*this).header = (*this).allocator.allocate(1);
    (*this).init();
  }

  /* Destructor */

  template <typename K, typename V, typename F, typename C, typename A
      , typename N, typename B, bool is_multiset, typename S>
  tree<K, V, F, C, A, N, B, is_multiset, S>::~tree() {
    (*this).destroy_tree((*header).parent());
    (*this).allocator.deallocate((*this).header, 1);
  }

  /* get_allocator */

  template <typename K, typename V, typename F, typename C, typename A
      , typename N, typename B, bool is_multiset, typename S>
  typename tree<K, V, F, C, A, N, B, is_multiset, S>::allocator_type
  tree<K, V, F, C, A, N, B, is_multiset, S>::get_allocator() const {
    return allocator_type();
  }

  /* key_comp */

  template <typename K, typename V, typename F, typename C, typename A
      , typename N, typename B, bool is_multiset, typename S>
  typename tree<K, V, F, C, A, N, B, is_multiset, S>::key_compare
  tree<K, V, F, C, A, N, B, is_multiset, S>::key_comp() const {
    return key_compare();
  }

  /* value_comp */

  template <typename K, typename V, typename F, typename C, typename A
      , typename N, typename B, bool is_multiset, typename S>
  typename tree<K, V, F, C, A, N, B, is_multiset, S>::value_compare
  tree<K, V, F, C, A, N, B, is_multiset, S>::value_comp() const {
    return value_compare();
  }

  /* Init */

  template <typename K, typename V, typename F, typename C, typename A
      , typename N, typename B, bool is_multiset, typename S>
  void tree<K, V, F, C, A, N, B, is_multiset, S>::init()
  {
    (*(*this).header).parent(0);
```

```
  if(N::has_list) {
    (*(*this).header).right(0);
    (*(*this).header).left(0);
    (*(*this).header).predecessor((*this).header);
    (*(*this).header).successor((*this).header);
  }
  else {
    (*(*this).header).left((*this).header);
    (*(*this).header).right((*this).header);
  }

  (*this).node_count = 0;
}

/* destroy_tree - Will destruct the entire tree
 *
 * Complexity O(n)
 * */
template <typename K, typename V, typename F, typename C, typename A
    , typename N, typename B, bool is_multiset, typename S>
void tree<K, V, F, C, A, N, B, is_multiset, S>::destroy_tree(
    node_type* x) {

  node_type* y = 0;

  while(x != 0) {

    if((*x).right() != 0) {
      (*this).destroy_tree((*x).right());
    }

    y = (*x).left();

    (*x).~N();
    (*this).allocator.deallocate(x, 1);

    x = y;
  }

}

/* Builds a copy of a tree
 *
 * Complexity O(n)
 */
template <typename K, typename V, typename F, typename C, typename A
    , typename N, typename B, bool is_multiset, typename S>
typename tree<K, V, F, C, A, N, B, is_multiset, S>::node_type*
tree<K, V, F, C, A, N, B, is_multiset, S>::build_tree(node_type*
    node, node_type** nodes, int& nodec) {
  if(node == 0)
    return 0;

  node_type* left;
  node_type* new_node;

  left = build_tree((*node).left(), nodes, nodec);

  if(N::has_list) {
    node_type* tmp_node = node;
    /* Recall: first node is in the tree */
    do {
      try {
        new_node = allocator.allocate(1);
        new (new_node) node_type((*tmp_node).content());
```

```
      }
      catch(...) {
        throw;
      }

      nodes[nodec] = new_node;
      ++nodec;

      tmp_node = (*tmp_node).successor();
    } while((*tmp_node).parent() == 0);
  }
  else {
    try {
      new_node = allocator.allocate(1);
      new (new_node) node_type((*node).content());
    }
    catch(...) {
      throw;
    }
  }

  (*new_node).left(left);
  (*new_node).right(build_tree((*node).right(), nodes, nodec));

  if((*new_node).left()) {
    (*(*new_node).left()).parent(new_node);
  }

  if((*new_node).right()) {
    (*(*new_node).right()).parent(new_node);
  }

  return new_node;
}

/* copy
 *
 * Complexity O(n)
 */

template <typename K, typename V, typename F, typename C, typename A
    , typename N, typename B, bool is_multiset, typename S>
void tree<K, V, F, C, A, N, B, is_multiset, S>::copy(const tree& x)
    {

  int new_node_count = x.node_count;

  node_type** nodes = (*this).array_allocator.allocate(
      new_node_count);
  node_type* new_root;
  int i = 0;

  int count = 0;
  try {
    new_root = build_tree((*x.header).parent(), nodes, count);
  }
  catch (...) {
    for(i=0; i < count; i++) {
      (*nodes[i]).~N();
      allocator.deallocate(nodes[i], 1);
    }
    array_allocator.deallocate(nodes, new_node_count);
    throw;
  }
```

```
  if(N::has_list) {
    /* Adjust the associated data structure */
    (*nodes[0]).predecessor((*this).header);
    (*(*this).header).successor(nodes[0]);

    /* We have 2+ elements */
    if(new_node_count > 1) {
      (*nodes[0]).successor(nodes[1]);

      for(i = 1; i < new_node_count - 1; ++i) {
        (*nodes[i]).predecessor(nodes[i-1]);
        (*nodes[i]).successor(nodes[i+1]);
      }

      (*nodes[new_node_count-1]).predecessor(nodes[new_node_count -
          2]);
      (*nodes[new_node_count-1]).successor((*this).header);
      (*(*this).header).predecessor(nodes[new_node_count-1]);
    }
    /* We have 1 element */
    else {
     (*nodes[0]).successor((*this).header);
     (*(*this).header).predecessor(nodes[0]);
    }
  }
  else {
    /* Find minimum */
    node_type* x = new_root;
    while((*x).left()) {
      x = (*x).left();
    }
    (*header).left(x);

    /* Find maximum */
    x = new_root;
    while((*x).right()) {
      x = (*x).right();
    }
    (*header).right(x);
  }

  (*header).parent(new_root);
  (*(*header).parent()).parent((*this).header);
  (*this).node_count = x.node_count;

  (*this).array_allocator.deallocate(nodes, new_node_count);
}

/* Copy constructor
 *
 * Complexity O(n)
 */

template <typename K, typename V, typename F, typename C, typename A
    , typename N, typename B, bool is_multiset, typename S>
tree<K, V, F, C, A, N, B, is_multiset, S>::tree(const tree& x) {

  try {
    (*this).header = allocator.allocate(1);
    (*this).init();
    if((*x.header).parent() != 0) {
      (*this).copy(x);
    }
  }
  catch (...) {
```

```
    if ((*this).header != 0) {
      allocator.deallocate((*this).header, 1);
    }
    throw;
  }

  (*this).allocator = x.allocator;
  (*this).comparator = x.comparator;
}

/* assignment operator
 *
 * Complexity O(n)
 */

template <typename K, typename V, typename F, typename C, typename A
    , typename N, typename B, bool is_multiset, typename S>
tree<K, V, F, C, A, N, B, is_multiset, S>&
tree<K, V, F, C, A, N, B, is_multiset, S>::operator=(const tree& x)
    {
  if (this != &x) {
    node_type* old_header = (*this).header;
    try {
      (*this).header = allocator.allocate(1);
    }
    catch(...) {
      (*this).header = old_header;
      throw;
    }

    try {
      (*this).init();

      if((*x.header).parent() != 0) {
        (*this).copy(x);
      }
    }
    catch(...) {
      (*this).destroy_tree((*header).parent());
      allocator.deallocate((*this).header, 1);
      (*this).header = old_header;
      throw;
    }

    /* It's now safe to free the old tree */
    (*this).destroy_tree((*old_header).parent());
    allocator.deallocate(old_header, 1);

    (*this).allocator = x.allocator;
    (*this).comparator = x.comparator;

  }
  return *this;

}

/* size */

template <typename K, typename V, typename F, typename C, typename A
    , typename N, typename B, bool is_multiset, typename S>
typename tree<K, V, F, C, A, N, B, is_multiset, S>::size_type
tree<K, V, F, C, A, N, B, is_multiset, S>::size() const {
  return (*this).node_count;
}
```

```
/* max_size */

template <typename K, typename V, typename F, typename C, typename A
    , typename N, typename B, bool is_multiset, typename S>
typename tree<K, V, F, C, A, N, B, is_multiset, S>::size_type
tree<K, V, F, C, A, N, B, is_multiset, S>::max_size() const {
  typename allocator_type::template rebind<char>::other
      char_allocator;
  size_type a = char_allocator.max_size(); // available memory in
      bytes
  size_type b = sizeof(node_type); // size of nodes
  size_type c = sizeof(container_type); // size of the tree
  // solve n from (n + 2) * b + c = a
  return (a - 2 * b - c) / b;
}

/* begin */

template <typename K, typename V, typename F, typename C, typename A
    , typename N, typename B, bool is_multiset, typename S>
typename tree<K, V, F, C, A, N, B, is_multiset, S>::node_type*
tree<K, V, F, C, A, N, B, is_multiset, S>::begin() const {
  if(!N::has_list) {
    return (*header).left();
  }
  return (*(*this).header).successor();
}

/* end */

template <typename K, typename V, typename F, typename C, typename A
    , typename N, typename B, bool is_multiset, typename S>
typename tree<K, V, F, C, A, N, B, is_multiset, S>::node_type*
tree<K, V, F, C, A, N, B, is_multiset, S>::end() const {
  return (*this).header;
}

/* insert(v)
 *
 * Complexity O(lg n)
 */

template <typename K, typename V, typename F, typename C, typename A
    , typename N, typename B, bool is_multiset, typename S>
typename cphstl::if_then_else<is_multiset,
                              typename tree<K, V, F, C, A, N, B,
                                  is_multiset, S>::node_type*,
                              std::pair<typename tree<K, V, F, C, A
                                  , N, B, is_multiset, S>::
                                  node_type*, bool>
                             >::type
tree<K, V, F, C, A, N, B, is_multiset, S>::insert(const value_type&
    v) {
  return (*this).insert_dispatch(v, cphstl::bool2type<is_multiset>()
      );
}

/* insert(pos, v)
 *
 * Complexity O(lg n), because of balancing operations.
 * */

template <typename K, typename V, typename F, typename C, typename A
    , typename N, typename B, bool is_multiset, typename S>
typename tree<K, V, F, C, A, N, B, is_multiset, S>::node_type*
```

```
tree<K, V, F, C, A, N, B, is_multiset, S>::insert(node_type*
    position, const value_type& v) {
  return (*this).insert_dispatch(position, v, cphstl::bool2type<
      is_multiset>());
}

/* insert(I, I)
 *
 * Complexity O(k lg n)
 */

template <typename K, typename V, typename F, typename C, typename A
    , typename N, typename B, bool is_multiset, typename S>
template <typename I>
void
tree<K, V, F, C, A, N, B, is_multiset, S>::insert(I x, I y) {
  (*this).insert_dispatch(x, y, cphstl::bool2type<is_multiset>());
}

/* _insert - Where the actual insertion is done,
 * after that rebalancing is done
 *
 * Complexity: Should be O(1), but is not currently
 */

template <typename K, typename V, typename F, typename C, typename A
    , typename N, typename B, bool is_multiset, typename S>
typename tree<K, V, F, C, A, N, B, is_multiset, S>::node_type*
tree<K, V, F, C, A, N, B, is_multiset, S>::_insert(node_type* y,
    const value_type& v, node_type* pred, node_type* succ, bool
    is_duplicate=false) {

  functor_type f;
  bool comp;

  node_type* z = allocator.allocate(1);
  new (z) node_type(v);

  key_type v_key = f(v);

  if(!is_duplicate || !N::has_list) {
    if (y == (*this).header) {
      (*header).parent(z);
    }
    else {
      try {
        key_type y_key = f((*y).content());
        comp = (*this).comparator(v_key, y_key);
      }
      catch(...) {
        (*z).~N();
        allocator.deallocate(z, 1);
        throw;
      }

      if (comp) {
        (*y).left(z);
      }
      else {
        (*y).right(z);
      }
    }
    (*z).parent(y);
  }
```

```
  /* Move to the last predecessor, recall that duplicates
   * can be identified by a null-ed parent node, faster
   * and simpler than using comparator */

  if(N::has_list) {
    if((*(*pred).successor()).parent() == 0) {
      pred = (*(*this).upper_bound(f(v))).predecessor();
    }

    (*z).add_to_list(pred, succ);
  } else {
    functor_type f;

    if((*header).left() == header || (*this).comparator(f(v), f((*(*
        header).left()).content()))) {
      (*header).left(z);
    }
    if((*header).right() == header || (*this).comparator(f((*(*
        header).right()).content()), f(v))) {
      (*header).right(z);
    }
  }

  if(!is_duplicate || !N::has_list) {
    B::insert_fixup(z, header);

  }

  ++(*this).node_count;

  return z;
}

/* Relink for erase will delink the node to delete from the tree,
    but not
 * deallocate it. As we know from Cormen et al we will use these 3
     cases:
 *
 *  & If the left subtree is empty, use the root of the right
     subtree to
 *    substitute the node to delete.
 *  & Vice versa for right subtree
 *  & If both subtrees are in use, find the successor and use this.
 *
 * Because of iterator validity we will not use plain swap but we
     will
 * relink the pointers.
 *
 * Complexity O(1) for set.
 */

template <typename K, typename V, typename F, typename C, typename A
    , typename N, typename B, bool is_multiset, typename S>
typename tree<K, V, F, C, A, N, B, is_multiset, S>::node_type*
tree<K, V, F, C, A, N, B, is_multiset, S>::relink_for_erase(
    node_type* z) {

  node_type* x_parent = 0;

  node_type* y = z;
  node_type* x = 0;
  /* Only one child either left or right */
  if ((*y).left() == 0) {
    x = (*y).right();
  }
```

```
  else if ((*y).right() == 0) {
    x = (*y).left();
  }
  /* Both childs, we will find the successor and "swap" it with the
      deleted node */
  else {
    y = (*z).successor();
    x = (*y).right();
  }

  node_type* new_z_parent = y->parent();

  /* We will "swap" here. We don't really swap since it will violate
       iterator
   * validity, but simply perform relinking. */
  if (y != z) {
    (*(*z).left()).parent(y);
    (*y).left((*z).left());
    if (y != (*z).right()) {
      x_parent = (*y).parent();
      if (x) {
        (*x).parent((*y).parent());
      }
      (*(*y).parent()).left(x);
      (*y).right((*z).right());
      (*(*z).right()).parent(y);
    }
    else {
      x_parent = y;
      new_z_parent = y;
    }
    if ((*header).parent() == z) {
      (*header).parent(y);
    }
    else if ((*z).is_left()) {
      (*(*z).parent()).left(y);
    }
    else {
      (*(*z).parent()).right(y);
    }
    (*y).parent((*z).parent());
    (*z).parent(new_z_parent);
    (*y).copy_aux(z);
  }
  /* We did not need to "swap" */
  else {
    x_parent = (*y).parent();
    if (x) {
      (*x).parent((*y).parent());
    }
    if ((*header).parent() == z) {
      (*header).parent(x);
    }
    else {
      if ((*z).is_left()) {
        (*(*z).parent()).left(x);
      }
      else {
        (*(*z).parent()).right(x);
      }
    }
  }

  return x;
}
```

```
template <typename K, typename V, typename F, typename C, typename A
    , typename N, typename B, bool is_multiset, typename S>
void
tree<K, V, F, C, A, N, B, is_multiset, S>::duplicate_swap(node_type*
     position) {

  node_type* y = (*position).successor();

  if((*position).is_left()) {
    (*(*position).parent()).left(y);
  }
  else if((*position).is_right()) {
    (*(*position).parent()).right(y);
  }
  /* Root */
  else {
    (*header).parent(y);
  }

  (*y).parent((*position).parent());
  (*y).left((*position).left());
  (*y).right((*position).right());

  if((*y).left() != 0) {
    (*(*y).left()).parent(y);
  }
  if((*y).right() != 0) {
    (*(*y).right()).parent(y);
  }

  (*y).copy_aux(position);
}

template <typename K, typename V, typename F, typename C, typename A
    , typename N, typename B, bool is_multiset, typename S>
void
tree<K, V, F, C, A, N, B, is_multiset, S>::normal_erase(node_type*
    position) {

  node_type* x = (*this).relink_for_erase(position);

  if(x != 0) {
    B::erase_fixup(x, position, header);
  }
}
/* erase(position)
 *
 * Complexity: O(lg n) because of fixups
 * Complexity for erasing duplicates: O(1)
 */

template <typename K, typename V, typename F, typename C, typename A
    , typename N, typename B, bool is_multiset, typename S>
void
tree<K, V, F, C, A, N, B, is_multiset, S>::erase(node_type* position
    ) {

  /* For nodes there provides the list:
   *   - IF this is the node which is kept in the tree and we have a
   *   supplicate, let us swap the duplicate in.
   *   - If there is not a duplicate, lets delete it in a normal way
       .
   *   (- If we are deleting a duplicate we will simply remove it
       from the
```

```
   *   list)
   * For nodes there don't provide the list mechanism:
   *   - We will simply treat it as set, which means that every node
   *   is in the tree, such that we will relink for erase
   */

  if(N::has_list) {
    if((*position).parent() != 0) {
      if((*(*position).successor()).parent() == 0) {
        (*this).duplicate_swap(position);
      }
      /* We are not deleting a duplicate */
      else {
        (*this).normal_erase(position);
      }
    }

    (*position).remove_from_list();
  }
  else {
    if((*header).left() == position) {
      (*header).left((*position).successor());
    }
    if((*header).right() == position) {
      (*header).right((*position).predecessor());
    }

    (*this).normal_erase(position);
  }

  (*position).~N();
  (*this).allocator.deallocate(position, 1);

  --(*this).node_count;

}

/* erase(k)
 *
 * Complexity O(lg n + k)
 */
template <typename K, typename V, typename F, typename C, typename A
    , typename N, typename B, bool is_multiset, typename S>
typename tree<K, V, F, C, A, N, B, is_multiset, S>::size_type
tree<K, V, F, C, A, N, B, is_multiset, S>::erase(key_type const& k)
    {

  size_type retval;

  /* The upperbound for how many elements we can erase is actually
      the node_count */
  functor_type f;

  node_type* erase_node = (*this).lower_bound(k);

  if(erase_node == (*this).header || (*this).comparator(k, f((*
      erase_node).content()))) {
    retval = 0;
  }
  else if(!is_multiset) {
    (*this).erase(erase_node);
    retval = 1;
  }
  else if(N::has_list) {
```

```
    /* since we only store the first node of sequence, we will just
        delete
     * until the value does not match anymore */
    size_type count = 0;

    bool stop = false;

    while(!stop) {
      node_type* tmp = (*erase_node).successor();

      /* next node is NOT a duplicate, so we will stop iteration */
      if((*tmp).parent() != 0) {
        stop = true;
      }

      (*this).erase(erase_node);
      count++;
      erase_node = tmp;
    }

    retval = count;
  }
  else {
    int new_node_count = (*this).node_count;
    node_type** for_erase = array_allocator.allocate(new_node_count)
        ;

    int count = 0;

    try {
      do {
        for_erase[count] = erase_node;
        ++count;
        erase_node = (*erase_node).successor();
      } while(erase_node != header &&
              !(*this).comparator(k, f((*erase_node).content())) &&
              !(*this).comparator(f((*erase_node).content()), k));
    }
    catch(...) {
      array_allocator.deallocate(for_erase, new_node_count);
      throw;
    }
    for(int i = 0; i < count; ++i) {
      (*this).erase(for_erase[i]);
    }

    array_allocator.deallocate(for_erase, new_node_count);
    retval = count;
  }

  return retval;
}

/* lower_bound(k)
 *
 * Complexity O(lg n)
 */
template <typename K, typename V, typename F, typename C, typename A
    , typename N, typename B, bool is_multiset, typename S>
typename tree<K, V, F, C, A, N, B, is_multiset, S>::node_type*
tree<K, V, F, C, A, N, B, is_multiset, S>::lower_bound(const
    key_type& k) const
{
  return S::find_node(k, typename S::lower_bound(), (*this).header,
      (*this).comparator);
```

```
}

/* upper_bound(k)
 *
 * Complexity O(lg n)
 */
template <typename K, typename V, typename F, typename C, typename A
    , typename N, typename B, bool is_multiset, typename S>
typename tree<K, V, F, C, A, N, B, is_multiset, S>::node_type*
tree<K, V, F, C, A, N, B, is_multiset, S>::upper_bound(const
    key_type& k) const
{
  return S::find_node(k, typename S::upper_bound(), (*this).header,
      (*this).comparator);
}

/* swap
 *
 * Complexity: O(1)
 */

template <typename K, typename V, typename F, typename C, typename A
    , typename N, typename B, bool is_multiset, typename S>
void
tree<K, V, F, C, A, N, B, is_multiset, S>::swap(tree<K, V, F, C, A,
    N, B, is_multiset, S>& t) {
  std::swap((*this).header, t.header);
  std::swap((*this).node_count, t.node_count);
  std::swap((*this).comparator, t.comparator);
  std::swap((*this).allocator, t.allocator);
}

/* casting operator - it makes it possible to print out the data
    structure
   (with balance information, ..)
 */

template <typename K, typename V, typename F, typename C, typename A
    , typename N, typename B, bool is_multiset, typename S>
tree<K, V, F, C, A, N, B, is_multiset, S>::operator std::string()
    const {
  if((*header).parent() == 0) {
    return std::string("The␣container␣is␣empty!");
  }

  return std::string(*(*header).parent());
}

template <typename K, typename V, typename F, typename C, typename A
    , typename N, typename B, bool is_multiset, typename S>
std::pair<typename tree<K, V, F, C, A, N, B, is_multiset, S>::
    node_type*, bool>
tree<K, V, F, C, A, N, B, is_multiset, S>::insert_dispatch(const
    value_type& v, cphstl::bool2type<false>) {

  std::pair<node_type*,
            std::pair<node_type*, node_type*> > p = S::
                insert_find_node(v, (*this).header, (*this).
                comparator);

  /* If an exception occured, no changes to the data structure has
      occured so
   * far */
```

```
  if(p.second.first != 0 && p.second.second != 0) {
    return std::pair<node_type*,bool>((*this)._insert(p.first, v, p.
        second.first, p.second.second, false), true);
  }
  return std::pair<node_type*,bool>(p.first, false);

}

template <typename K, typename V, typename F, typename C, typename A
    , typename N, typename B, bool is_multiset, typename S>
typename tree<K, V, F, C, A, N, B, is_multiset, S>::node_type*
tree<K, V, F, C, A, N, B, is_multiset, S>::insert_dispatch(const
    value_type& v, cphstl::bool2type<true>) {

  std::pair<node_type*,
            std::pair<node_type*, node_type*> > p = S::
                insert_find_node(v, (*this).header, (*this).
                comparator);

  functor_type f;
  bool is_duplicate = false;

  if(p.second.first != header &&
     !(*this).comparator(f((*p.second.first).content()), f(v)) &&
     !(*this).comparator(f(v), f((*p.second.first).content()))) {
    is_duplicate = true;
  }

  if(p.second.second != header &&
     !(*this).comparator(f((*p.second.second).content()), f(v)) &&
     !(*this).comparator(f(v), f((*p.second.second).content()))) {
    is_duplicate = true;
  }

  return (*this)._insert(p.first, v, p.second.first, p.second.second
      , is_duplicate);
}

template <typename K, typename V, typename F, typename C, typename A
    , typename N, typename B, bool is_multiset, typename S>
typename tree<K, V, F, C, A, N, B, is_multiset, S>::node_type*
tree<K, V, F, C, A, N, B, is_multiset, S>::insert_dispatch(node_type
    * position, const value_type& v, cphstl::bool2type<false>) {

  functor_type f;

  key_type k = f(v);
  key_type position_key = f((*position).content());

  if (position == (*this).begin()) {
    if ((*this).size() > 0 && (*this).comparator(k, position_key)) {
      return (*this)._insert(position, v, (*position).predecessor(),
          position, false);
    }
    else {
      return (*this).insert(v).first;
    }
  } else if (position == (*this).end()) {
    node_type* rightmost = (*position).predecessor();
    key_type rightmost_key = f((*rightmost).content());

    if ((*this).comparator(rightmost_key, k)) {
      return (*this)._insert(rightmost, v, (*position).predecessor()
          , position, false);
    }
```

```
    else {
      return (*this).insert(v).first;
    }
  } else {
    node_type* before = (*position).predecessor();
    key_type before_key = f((*before).content());

    if ((*this).comparator(before_key, k) && (*this).comparator(k,
        position_key)) {
      if ((*before).right() == 0) {
        return (*this)._insert(before, v, before, position, false);
      }
      else {
        return (*this)._insert(position, v, before, position, false)
            ;
      }
    } else {
      return (*this).insert(v).first;
    }
  }
}

template <typename K, typename V, typename F, typename C, typename A
    , typename N, typename B, bool is_multiset, typename S>
typename tree<K, V, F, C, A, N, B, is_multiset, S>::node_type*
tree<K, V, F, C, A, N, B, is_multiset, S>::insert_dispatch(node_type
    * position, const value_type& v, cphstl::bool2type<true>) {

  functor_type f;
  key_type k = f(v);
  key_type position_key = f((*position).content());

  if (position == (*this).begin()) { // begin()
    if (size() > 0 && !(*this).comparator(k, position_key) && !(*
        this).comparator(position_key, k)) {
      return _insert(position, v, (*position).predecessor(),
          position, true);
    }
    else if (size() > 0 && (*this).comparator(k, position_key)) {
      return _insert(position, v, (*position).predecessor(),
          position, false);
    }
    else {
      return (*this).insert(v);
    }
  } else if (position == (*this).end()) { // end()
    node_type* rightmost = (*position).predecessor();
    key_type rightmost_key = f((*rightmost).content());

    if (!(*this).comparator(k, rightmost_key) && !(*this).comparator
        (rightmost_key, k)) {
      return (*this)._insert(rightmost, v, position, (*position).
          successor(), true);
    }
    else if (!(*this).comparator(k, rightmost_key)) {
      return (*this)._insert(rightmost, v, position, (*position).
          successor(), false);
    }
    else {
      return (*this).insert(v);
    }
  } else {
    node_type* before = (*position).predecessor();
    key_type before_key = f((*before).content());
```

```
    bool is_duplicate = false;

    if(!(*this).comparator(before_key, k) && !(*this).comparator(k,
        before_key))
      is_duplicate = true;
    if(!(*this).comparator(position_key, k) && !(*this).comparator(k
        , position_key))
      is_duplicate = true;

    if (!(*this).comparator(k, before_key) && !(*this).comparator(
        position_key, k)) {
      if ((*before).right() == 0) {
        return (*this)._insert(before, v, before, position,
            is_duplicate);
      }
      else {
        return (*this)._insert(position, v, before, position,
            is_duplicate);
      }
    }
    else {
      return (*this).insert(v);
    }
  }
}


template <typename K, typename V, typename F, typename C, typename A
    , typename N, typename B, bool is_multiset, typename S>
template <typename I>
void
tree<K, V, F, C, A, N, B, is_multiset, S>::insert_dispatch(I x, I y,
     cphstl::bool2type<false>) {
  int count = 0;
  for(I it = x; it != y; ++it)
    ++count;

  node_type** undolog = array_allocator.allocate(count);

  count = 0;
  try {
    for(I it = x; it != y; ++it) {
      std::pair<node_type*, bool> pair;
      pair = (*this).insert(*it);
      if(pair.second) {
        undolog[count] = pair.first;
        ++count;
      }
    }
  }
  catch (...) {
    for (int i = 0;i<=count;++i) {
      (*this).erase(undolog[i]);
    }
    (*this).array_allocator.deallocate(undolog, count);
    throw;
  }

  (*this).array_allocator.deallocate(undolog, count);
}

template <typename K, typename V, typename F, typename C, typename A
    , typename N, typename B, bool is_multiset, typename S>
template <typename I>
void
```

```
  tree<K, V, F, C, A, N, B, is_multiset, S>::insert_dispatch(I x, I y,
       cphstl::bool2type<true>) {
    int count = 0;

    for(I it = x; it != y; ++it)
      ++count;

    node_type** undolog = array_allocator.allocate(count);

    count = 0;
    try {
      for(I it = x; it != y; ++it) {
        undolog[count] = (*this).insert(*it);
        ++count;
      }
    }
    catch (...) {
      for (int i = 0;i<=count;++i) {
        (*this).erase(undolog[i]);
      }
      (*this).array_allocator.deallocate(undolog, count);
      throw;
    }

    (*this).array_allocator.deallocate(undolog, count);
  }
}
```

Appendix C.3 tree_balance.h++

```
/*

   Balancing base class, will implement a conventional binary tree

   Author: Bo Simonsen, December 2008
*/

#ifndef _CPHSTL_TREE_BALANCE_H
#define _CPHSTL_TREE_BALANCE_H

namespace cphstl {
  template <typename N >
  class tree_balance_policy {
  public:
    typedef N node_type;

    static void insert_fixup(node_type* z, node_type* header) {
    }
    static void erase_fixup(node_type* x, node_type* y, node_type*
        header) {
    }
    static void touch(node_type* z, node_type* header) {
    }
  };
}
#endif
```

Appendix C.4 tree_comp.h++

```
/*

  Comparator
  Taken from red_black_tree, this should properly be shared, when the
  red black tree implementation is fixed
```

```
  Authors: Bo Simonsen, June 2008

*/

namespace cphstl {
  template <typename K, typename V, typename C>
  class _value_compare
    : public std::binary_function<V, V, bool> {
  public:
    typedef K key_type;
    typedef V value_type;
    typedef C key_compare;

    bool operator()(value_type const& x, value_type const& y) const {
      cphstl::unnamed::key_map<key_type, value_type> key;
      return comparator(key[x], key[y]);
    }
  protected:
    _value_compare(key_compare const& predicate)
      : comparator(predicate) {
    }

    key_compare comparator;
  };

}
```

Appendix C.5 tree_func.h++

```
/*

  Functors

  Authors: Bo Simonsen, June 2008

*/


#ifndef _CPHSTL_TREE_FUNC_H
#define _CPHSTL_TREE_FUNC_H

namespace cphstl {

  /* Functors used for find_node */

  template<typename K,
           typename F,
           typename C,
           typename N>
  class f_lowerbound {
  public:
    typedef N node_type;
    typedef K key_type;
    typedef F functor_type;
    typedef C comparator_type;

    bool operator()(comparator_type const& c, node_type* x, const
        key_type& k) const {
      functor_type f;
      key_type x_key = f(x->content());
      return !c(x_key, k);
    }
  };
```

```
  /* Functors used for find_node */

  template<typename K,
           typename F,
           typename C,
           typename N>
  class f_upperbound {
  public:
    typedef N node_type;
    typedef K key_type;
    typedef F functor_type;
    typedef C comparator_type;

    bool operator()(comparator_type const& c, node_type* x, const
        key_type& k) const {
      functor_type f;
      key_type x_key = f(x->content());
      return c(k, x_key);
    }
  };
}

#endif
```

Appendix C.6 tree_node.h++

```
/*

  Tree node base class

  Authors: Bo Simonsen, June 2008

*/

#include <string>
#include <iostream>

/* Use this using CRTP, like

     template <typename V>
     class my_node : public tree_node<my_node<V> > {

     }
*/

#ifndef _CPHSTL_TREE_NODE
#define _CPHSTL_TREE_NODE
namespace cphstl {
  template <typename V, typename N>
  class tree_node {
  protected:
    typedef N node_type;
    typedef V value_type;

    node_type* _parent;
    node_type* _left;
    node_type* _right;
    node_type* _prev;
    node_type* _next;

    value_type _data;

  public:
    enum {has_list = true};
```

```
tree_node(value_type const& v) {
  (*this)._parent = 0;
  (*this)._left = 0;
  (*this)._right = 0;
  (*this)._prev = 0;
  (*this)._next = 0;
  (*this)._data = v;
}

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

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


node_type* parent() const {
  return (*this)._parent;
}
void parent(node_type* n) {
  (*this)._parent = n;
}
node_type* left() const {
  /*if(this == 0)
    return 0;*/

  return (*this)._left;
}
void left(node_type* n) {
  (*this)._left = n;
}
node_type* right() const {
  /*if(this == 0)
    return 0;*/

  return (*this)._right;
}
void right(node_type* n) {
  (*this)._right = n;
}
void successor(node_type* n) {
  (*this)._next = n;
}
void predecessor(node_type* n) {
  (*this)._prev = n;
}
node_type* successor() const {
  return (*this)._next;
}
node_type* predecessor() const {
  return (*this)._prev;
}
void copy_aux(node_type* n) {
}

bool is_right() const {
  if((*(*this).parent()).right() == this)
    return true;
  return false;
}
bool is_left() const {
  if((*(*this).parent()).left() == this)
    return true;
```

```
  return false;
}
node_type* grandparent() {
  return (*(*this).parent()).parent();
}

void rotate_left(node_type* header) {
  node_type* y = (*this).right();

  (*this).right((*y).left());

  if ((*y).left() !=0) {
    (*(*y).left()).parent((node_type*) this);
  }

  (*y).parent((*this).parent());
  if (this == (*header).parent()) {
    (*header).parent(y);
  }

  else if (this == (*(*this).parent()).left()) {
    (*(*this).parent()).left(y);
  }
  else {
    (*(*this).parent()).right(y);
  }

  (*y).left((node_type*) this);
  (*this).parent(y);
}

void rotate_right(node_type* header) {
  node_type* y = (*this).left();

  (*this).left((*y).right());

  if ((*y).right() !=0) {
    (*(*y).right()).parent((node_type*) this);
  }

  (*y).parent((*this).parent());

  if (this == (*header).parent()) {
    (*header).parent(y);
  }
  else if (this == (*(*this).parent()).right()) {
    (*(*this).parent()).right(y);
  }
  else {
    (*(*this).parent()).left(y);
  }

  (*y).right((node_type*) this);
  (*this).parent(y);
}

void add_to_list(node_type* p, node_type* s) {
  (*this)._prev = p;
  (*this)._next = s;
  (*p).successor((node_type*) this);
  (*s).predecessor((node_type*) this);
}

void remove_from_list() {
  (*_prev).successor(_next);
```

```
      (*_next).predecessor(_prev);
    }

    operator std::string() const {
      std::stringstream ss;
      std::string data_str;

      ss << (*this)._data;
      ss >> data_str;

      std::string ret = "";

      ret += "Node: " + data_str + "\n";

      if((*this)._left != 0) {
        ret += "Left\n" + (std::string) *(*this)._left;
      }

      if((*this)._right != 0) {
        ret += "Right\n" + (std::string) *(*this)._right;
      }

      return ret;
    }

  };
}
#endif
```

Appendix C.7 tree_se_node.h++

```
/*

  Space efficient tree node base class

  Authors: Bo Simonsen, June 2008

*/

#include <string>
#include <iostream>


#ifndef _CPHSTL_TREE_SE_NODE
#define _CPHSTL_TREE_SE_NODE
namespace cphstl {
  template <typename V, typename N>
  class tree_se_node {
  protected:
    typedef N node_type;
    typedef V value_type;

    node_type* _parent;
    node_type* _left;
    node_type* _right;

    value_type _data;

  public:
    enum {has_list = false};

    tree_se_node(const value_type& v) {
      (*this)._parent = 0;
      (*this)._left = 0;
      (*this)._right = 0;
```

```
  (*this)._data = v;
}

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

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


node_type* parent() const {
  return (*this)._parent;
}
void parent(node_type* n) {
  (*this)._parent = n;
}
node_type* left() const {
  if(this == 0)
    return 0;
  return (*this)._left;
}
void left(node_type* n) {
  (*this)._left = n;
}
node_type* right() const {
  if(this == 0)
    return 0;
  return (*this)._right;
}
void right(node_type* n) {
  (*this)._right = n;
}

void successor(node_type* n) {
}
void predecessor(node_type* n) {
}

node_type* successor() const {
  node_type* x = (node_type*) this;
  if((*x).right() != 0) {
    if((*(*x).right()).parent() != x) {
      return (*x).left();
    }
    x = (*x).right();
    while((*x).left() != 0) {
      x = (*x).left();
    }
  }
  else {
    node_type* y = (*x).parent();
    while(x == (*y).right()) {
      x = y;
      y = (*y).parent();
    }
    if((*x).right() != y) {
      x = y;
    }
  }
  return x;
}
node_type* predecessor() const {
  node_type* x = (node_type*) this;
```

```
  if((*x).left() != 0) {
    if((*(*x).left()).parent() != x) {
      return (*x).right();
    }
    x = (*x).left();
    while((*x).right() != 0) {
      x = (*x).right();
    }
  }
  else {
    node_type* y = (*x).parent();
    while(x == (*y).left()) {
      x = y;
      y = (*y).parent();
    }
    if((*x).left() != y) {
      x = y;
    }
  }
  return x;
}

void copy_aux(node_type* n) {
}

bool is_right() const {
  if((*(*this).parent()).right() == this)
    return true;
  return false;
}
bool is_left() const {
  if((*(*this).parent()).left() == this)
    return true;
  return false;
}
node_type* grandparent() {
  return (*(*this).parent()).parent();
}

node_type* sibling() {
  if((*this).is_left()) {
    return (*(*this).parent()).right();
  }

  return (*(*this).parent()).left();
}
void rotate_left(node_type* header) {
  node_type* y = (*this).right();

  (*this).right((*y).left());

  if ((*y).left() !=0) {
    (*(*y).left()).parent((node_type*) this);
  }

  (*y).parent((*this).parent());

  if (this == (*header).parent()) {
    (*header).parent(y);
  }
  else if (this == (*(*this).parent()).left()) {
    (*(*this).parent()).left(y);
  }
  else {
    (*(*this).parent()).right(y);
```

```
      }

      (*y).left((node_type*) this);
      (*this).parent(y);
    }

    void rotate_right(node_type* header) {
      node_type* y = (*this).left();

      (*this).left((*y).right());

      if ((*y).right() !=0) {
        (*(*y).right()).parent((node_type*) this);
      }

      (*y).parent((*this).parent());

      if (this == (*header).parent()) {
        (*header).parent(y);
      }
      else if (this == (*(*this).parent()).right()) {
        (*(*this).parent()).right(y);
      }
      else {
        (*(*this).parent()).left(y);
      }

      (*y).right((node_type*) this);
      (*this).parent(y);
    }

    void add_to_list(node_type* p, node_type* s) {
    }

    void remove_from_list() {
    }
  };
}
#endif
```

Appendix C.8 tree_node_shared.h++

```
#ifndef _CPHSTL_TREE_NODE_SHARED_H
#define _CPHSTL_TREE_NODE_SHARED_H

namespace cphstl {

  template <typename V, bool se, typename N>
  class tree_node_selector {
  public:
    typedef typename cphstl::if_then_else<se,
                                  cphstl::tree_se_node< V, N >,
                                  cphstl::tree_node< V, N > >::type
                                      node_type;
  };

}
#endif
```

Appendix C.9 tree_proxy.h++

```
namespace cphstl {
  template <typename C>
  class comparator_proxy {
```

```
public:
  typedef typename C::first_argument_type first_argument_type;
  typedef typename C::second_argument_type second_argument_type;

  comparator_proxy() {
    (*this).c = new C();
  }
  comparator_proxy(C& c) {
    (*this).c = new C(c);
  }
  comparator_proxy(comparator_proxy& cp) {
    (*this).c = new C(*cp.c);
  }
  comparator_proxy operator=(comparator_proxy const& cp) {
    delete (*this).c;
    (*this).c = new C(*cp.c);
    return (*this);
  }
  comparator_proxy operator=(C const& c) {
    delete (*this).c;
    (*this).c = new C(c);
    return (*this);
  }
  ~comparator_proxy() {
    delete (*this).c;
  }
  bool operator()(const first_argument_type& t1, const
      second_argument_type& t2) const {
    return (*(*this).c)(t1, t2);
  }
  void swap(comparator_proxy& o) {
    std::swap(o.c, (*this).c);
  }
private:
  C* c;
};

template <typename A>
class allocator_proxy {
public:
  typedef typename A::size_type size_type;
  typedef typename A::difference_type difference_type;
  typedef typename A::pointer pointer;
  typedef typename A::const_pointer const_pointer;
  typedef typename A::reference reference;
  typedef typename A::const_reference const_reference;
  typedef typename A::value_type value_type;

  template <class U>
  struct rebind { typedef allocator_proxy<typename A::template
      rebind<U>::other > other; };

  /* Def. constructor */
  allocator_proxy() {
    (*this).a = new A();
  }
  /* Parameterized constructor */
  allocator_proxy(A& a) {
    (*this).a = new A(a);
  }
  /* Destructor */
  ~allocator_proxy() {
    delete (*this).a;
  }
  /* Copy constructor */
```

```
    allocator_proxy(allocator_proxy& ap) {
      (*this).a = new A(*ap.a);
    }
    /* Assigment operator */
    allocator_proxy operator=(allocator_proxy const& ap) {
      delete (*this).a;
      (*this).a = new A(*ap.a);
      return (*this);
    }
    allocator_proxy operator=(A const& a) {
      delete (*this).a;
      (*this).a = new A(a);
      return (*this);
    }

    pointer address(reference ref) const {
      return (*a).address(ref);
    }
    const_pointer address(const_reference const_ref) const {
      return (*a).address(const_ref);
    }
    size_type max_size() const throw() {
      return (*a).max_size();
    }
    pointer allocate(size_type s, const_pointer p = 0) {
      return (*a).allocate(s);
    }
    void construct(pointer p,value_type const& v) {
      (*a).construct(p,v);
    }
    void destroy(pointer p){
      (*a).construct(p);
    }
    void deallocate(pointer p,size_type s) {
      (*a).deallocate(p,s);
    }
    void swap(allocator_proxy& o) {
      std::swap(o.a, (*this).a);
    }
  private:
    A* a;

  };

}

namespace std {
  template <typename A>
  void swap(cphstl::allocator_proxy<A>& t1, cphstl::allocator_proxy<A
      >& t2) {
    t1.swap(t2);
  }
  template <typename C>
  void swap(cphstl::comparator_proxy<C>& t1, cphstl::comparator_proxy<
      C>& t2) {
    t1.swap(t2);
  }



}
```

Appendix C.10 tree_search.h++

```
#include "tree_func.h++"
```

```
namespace cphstl {
  template <typename K,
            typename V,
            typename F,
            typename C,
            typename N,
            typename B,
            bool is_multiset >
  class tree_search {
  private:
    typedef K key_type;
    typedef V value_type;
    typedef N node_type;
    typedef C comparator_type;
    typedef F functor_type;

  public:
    typedef f_lowerbound<K, F, C, N> lower_bound;
    typedef f_upperbound<K, F, C, N> upper_bound;

    template <typename Z>
    static node_type* find_node(key_type const& k, Z const& z,
        node_type* header, comparator_type const& c) {

      node_type* y = header; /* Last node which is not less than k. */
      node_type* x = (*header).parent(); /* Current node. */

      while (x != 0) {
        if (z(c, x, k)) {
          y = x;
          x = (*x).left();
        }
        else {
          x = (*x).right();
        }
      }

      if(y != header) {
        B::touch(y, header);
      }

      return y;
    }

    static std::pair<node_type*, std::pair<node_type*, node_type*> >
    insert_find_node(value_type const& v, node_type* header,
        comparator_type const& c) {
      functor_type f;

      node_type* y = header;
      node_type* x = (*header).parent();

      node_type* pred = header;
      node_type* succ = header;

      bool comp = true;
      key_type k = f(v);

      while (x != 0) {
        y = x;
        key_type x_key = f((*x).content());

        comp = c(k, x_key);
        if(!is_multiset) {
```

```
          bool comp_equal = !comp && !c(x_key, k);

          if(comp_equal) {
            return std::pair<node_type*,
                             std::pair<node_type*, node_type*> >
                               (y, std::pair<node_type*, node_type
                                   *>(0, 0));
          }
        }

        if (comp) {
          if(N::has_list) {
            succ = x;
          }
          x = (*x).left();
        }
        else {
          if(N::has_list) {
            pred = x;
          }
          x = (*x).right();
        }
      }

      return std::pair<node_type*,
                       std::pair<node_type*, node_type*> >
                         (y, std::pair<node_type*, node_type*>(
                             pred, succ));

    }

  };
}
```

Appendix D. Balancing and storage policies

Appendix D.1 aa_tree_balance.h++

```
/*

  AA tree balancer

  Author: Jyrki Katajainen, May 2007,
          Lasse Jon Fuglsang Pedersen, Mads Ruben Burgdorff Kristensen
              ,
          Anders Sabinsky Toegern, June 2007,
          Bergur Ziska, June 2008,
          Bo Simonsen, January 2009.

*/

namespace cphstl {
  template <typename N>
  class aa_tree_balance_policy : public tree_balance_policy<N> {
  public:
    typedef N node_type;
  private:
    // Right rotation and level adjustment
    static void skew(node_type* oldparent, node_type* header) {

      (*oldparent).rotate_right(header);

      if((*oldparent).left() == 0) {
        (*oldparent).level(1);
```

```
      }
      else {
        (*oldparent).level((*(*oldparent).left()).level() + 1);
      }
    }

    // A conditional left rotation on a binary tree.
    static bool split(node_type* oldparent, node_type* header) {
      node_type* newp = (*oldparent).right();

      if (newp != 0 && (*newp).right() != 0 &&
          (*(*newp).right()).level() == (*oldparent).level()) {

        (*oldparent).rotate_left(header);
        (*newp).level((*oldparent).level() + 1);

        return true;
      }
      else {
        return false;
      }
    }
  public:
    static void insert_fixup(node_type* z, node_type* header) {

      (*header).level(0);

      node_type* y = (*z).parent();

      if((*y).level() != 1)
        return;

      while(y != header) {

        int level = 0;
        if((*y).left() == 0)
          level = 1;
        else
          level = (*(*y).left()).level() + 1;

        if((*y).level() != level) {
          skew(y, header);

          if((*y).right() == 0 || (*y).level() != (*(*y).right()).
              level()) {
            y = (*y).parent();
          }
          if(y == header)
            break;
        }
        if(!split((*y).parent(), header))
          break;

        y = (*y).parent();
      }
    }
    static void erase_fixup(node_type* z, node_type* y, node_type*
        header) {
      node_type* tmp = z;
      while (tmp != header) {
        int level = 0;

        if((*tmp).left() == 0)
          level = 1;
        else
```

```
          level = (*(*tmp).left()).level() + 1;

        // One of tmp's childern had it's level reduced
        if ((*tmp).level() > level) {
          (*tmp).level((*tmp).level() - 1);
          if (split (tmp, header)) {
            if (split (tmp, header)) {
              skew ((*(*tmp).parent()).parent(), header);
            }
            break;
          }
          tmp = (*tmp).parent();
        }
        else {
          if((*tmp).right() == 0) {
            level = 1;
          }
          else {
            level = (*(*tmp).right()).level() + 1;
          }

          if ((*tmp).level() <= level) {
            break;
          }
          else {
            skew (tmp, header);
            if ((*tmp).level() > (*(*tmp).parent()).level()) {
              skew (tmp, header);
              split ((*(*tmp).parent()).parent(), header);
              break;
            }
            tmp = (*(*tmp).parent()).parent();
          }
        }
      }
    }
  };
} // namespace cphstl
```

Appendix D.2 aa_tree_node.h++

```
/*

  AA node class

  Authors: Bo Simonsen, January 2008

*/

#include <string>
#include <iostream>
#include <cassert>

#include "tree_node.h++"
#include "tree_se_node.h++"
#include "tree_node_shared.h++"

namespace cphstl {

  template <typename V, bool se, typename N>
  class aa_tree_node_base : public tree_node_selector<V, se, N>::
      node_type {
  private:
    typedef int level_type;
```

```
    typedef typename tree_node_selector<V, se, N >::node_type
        superclass;
  public:
    typedef V value_type;
  private:
    /* We don't want default construction */
    aa_tree_node_base() {
    }
  public:
    aa_tree_node_base(value_type const& v) : superclass(v) {
      (*this)._level = 1;
    }
    level_type level() const {
      return (*this)._level;
    }
    void level(level_type const& l) {
      (*this)._level = l;
    }
    void copy_aux(N* n) {
      (*this).level((*n).level());
    }
    operator std::string() const {
      std::stringstream ss, _ss;
      std::string level_str;
      std::string data_str;

      ss << _level;
      ss >> level_str;

      _ss << (*this)._data;
      _ss >> data_str;

      std::string ret = "";

      ret += "Node:␣" + data_str + "␣level:␣" + level_str + "\n";

      if((*this)._left != 0) {
        ret += "Left\n" + (std::string) *(*this)._left;
      }

      if((*this)._right != 0) {
        ret += "Right\n" + (std::string) *(*this)._right;
      }

      return ret;
    }
  private:
    level_type _level;

  };

  template <typename V, bool se = false>
  class aa_tree_node : public aa_tree_node_base<V, se, aa_tree_node<V,
       se> > {
  private:
    aa_tree_node() {}
  public:
    typedef aa_tree_node_base<V, se, aa_tree_node<V, se> > superclass;
    aa_tree_node(V const& v) : superclass(v) {
    }
  };
}
```

Appendix D.3 avl_tree_balance.h++

```
/*

   Balancing mechanism for the AVL tree

   Author: Stephan Lynge, Herve Broenniman
           Bo Simonsen, December 2008
*/

#include "tree_balance.h++"

#ifndef _CPHSTL_AVL_TREE_BALANCE_H
#define _CPHSTL_AVL_TREE_BALANCE_H

namespace cphstl {
  template <typename N>
  class avl_tree_balance_policy : public tree_balance_policy<N> {
  public:
    typedef N node_type;
  private:
    static short get_side(node_type* x, node_type* header) {
      node_type* p = (*x).parent();
      if( (*header).parent() == x ) {
        return 0;
      }

      if( (*p).left() == x ) {
        return -1;
      }
      if( (*p).right() == x ) {
        return 1;
      }

      return 0; // Should never come her
    }
    static void rebalance(node_type* x, node_type* header, int cmd,
        short side) {
      while( side != 0 ) {
        if( (*x).balance() != cmd * side ) {
          (*x).balance((*x).balance() + cmd * side);
        }
        else {
          side = (*x).balance();
          node_type* y = (*x).left();
          if( (*x).balance() == 1 ) {
            y = (*x).right();
          }
          if( (*y).balance() == -side ) {
            node_type* z = (*y).right();
            if( side == 1 ) {
              z = (*y).left();
            }
            if( (*z).balance() == -side ) {
              (*x).balance(0);
              (*y).balance(side);
              (*z).balance(0);
            }
            else {
              if( (*z).balance() == side ) {
                (*x).balance(-side);
                (*y).balance(0);
                (*z).balance(0);
              }
              else {
                (*x).balance(0);
                (*y).balance(0);
```

```
            }
          }
          if( side == -1 ) {
            (*y).rotate_left(header);
          }
          else {
            (*y).rotate_right(header);
          }
        }
        else {
          if( (*y).balance() == side ) {
            (*x).balance(0);
            (*y).balance(0);
          }
          else {
            if( (*y).balance() == 0 ) {
              (*x).balance(side);
              (*y).balance(-side);
            }
          }
        }
        if( side == -1 ) {
          y = (*x).left();
          (*x).rotate_right(header);
        }
        else {
          y = (*x).right();
          (*x).rotate_left(header);
        }
        x = y;
      }
      if( ( cmd == 1 && (*x).balance() == 0 ) || ( cmd == -1 && (*x)
          .balance() != 0 ) ) {
        break;
      }

      side = get_side( x, header);
      x = (*x).parent();
    }
  }

public:
  static void insert_fixup(node_type* z, node_type* header) {
    short link_side = get_side( z, header);
    rebalance((*z).parent(), header, 1, link_side);
  }
  static void erase_fixup(node_type* z, node_type* y, node_type*
      header) {
    short link_side = get_side(y, header);

    node_type* w = (*y).parent();

    if((*header).parent() != w) {
      rebalance(w, header, -1, link_side);
    }
  }
};
}
#endif
```

Appendix D.4 avl_tree_node.h++

```
/*

  AVL node class
```

```
  Authors: Bo Simonsen, June 2008

*/

#include <string>
#include <iostream>
#include <cassert>

#include "tree_node.h++"
#include "tree_se_node.h++"
#include "tree_node_shared.h++"

namespace cphstl {

  template <typename V, bool se, typename N>
  class avl_tree_node_base : public tree_node_selector<V, se, N>::
      node_type {
  private:
    typedef typename tree_node_selector<V, se, N >::node_type
        superclass;
  public:
    typedef V value_type;
  private:
    /* We don't want default construction */
    avl_tree_node_base() {
    }
  public:
    avl_tree_node_base(value_type const& v) : superclass(v) {
      (*this)._balance = 0;
    }
    short balance() const {
      return (*this)._balance;
    }
    void balance(short const& b) {
      assert(b >= -1 && b <= 1);
      (*this)._balance = b;
    }
    void copy_aux(avl_tree_node_base* n) {
      (*this).balance((*n).balance());
    }
    operator std::string() const {
      std::stringstream ss, _ss;
      std::string balance_str;
      std::string data_str;

      ss << (*this)._balance;
      ss >> balance_str;

      _ss << (*this)._data;
      _ss >> data_str;

      std::string ret = "";

      if((*this)._left != 0) {
        ret += "Left\n" + (std::string) *(*this)._left;
      }

      ret += "Node: " + data_str + " balance: " + balance_str + "\n";

      if((*this)._right != 0) {
        ret += "Right\n" + (std::string) *(*this)._right;
      }

      return ret;
```

```
    }
  private:
    short _balance;

  };

  template <typename V, bool se = false>
  class avl_tree_node : public avl_tree_node_base<V, se, avl_tree_node
      <V, se> > {
  private:
    avl_tree_node() {}
  public:
    typedef avl_tree_node_base<V, se, avl_tree_node<V, se> >
        superclass;
    avl_tree_node(V const& v) : superclass(v) {
    }
  };
}
```

Appendix D.5 packed_avl_tree_node.h++

```
/*

  AVL node class (with bitpacking support)

  Authors: Bo Simonsen, June 2008

*/

#include <string>
#include <iostream>
#include <cassert>

#include "tree_node.h++"
#include "tree_se_node.h++"
#include "tree_node_shared.h++"

namespace cphstl {
  template <typename V, bool se, typename N>
  class avl_tree_bp_node_base : public tree_node_selector<V, se, N>::
      node_type {
  private:
    typedef N node_type;
    typedef typename tree_node_selector<V, se, N>::node_type
        superclass;
  public:
    typedef V value_type;
  private:
    /* We don't want default construction */
    avl_tree_bp_node_base() {
    }
  public:
    avl_tree_bp_node_base(value_type const& v) : superclass(v) {
    }
    node_type* left() const {
      return (node_type*) ( ((int) (*this)._left) & -2);
    }

    void left(node_type* n) {
      int color_indicator = ((int) (*this)._left) & 1;
      (*this)._left=(node_type*) ( ((int) n) + color_indicator);
    }

    node_type* right() const {
      return (node_type*) ( ((int) (*this)._right) & -2);
```

```
  }

  void right(node_type* n) {
    int color_indicator = ((int) (*this)._right) & 1;
    (*this)._right=(node_type*) ( ((int) n) + color_indicator);
  }

  short balance() const {
    if( ((int) (*this)._left) & 1 == 1 ) {
      return -1;
    }
    if ( ((int) (*this)._right) & 1 == 1 ) {
      return 1;
    }

    return 0;
  }
  void balance(short const& b) {
    assert(b >= -1 && b <= 1);
    if(b == 0) {
      (*this)._left = (node_type*) ( ((int) (*this)._left) & -2);
      (*this)._right = (node_type*) ( ((int) (*this)._right) & -2);
    }
    else if(b == -1) {
      (*this)._left = (node_type*) ( ((int) (*this)._left) | 1);
      (*this)._right = (node_type*) ( ((int) (*this)._right) & -2);
    }
    else if(b == 1) {
      (*this)._left = (node_type*) ( ((int) (*this)._left) & -2);
      (*this)._right = (node_type*) ( ((int) (*this)._right) | 1);
    }
  }

  void copy_aux(node_type* n) {
    (*this).balance((*n).balance());
  }

  operator std::string() const {
    std::stringstream ss, _ss;
    std::string balance_str;
    std::string data_str;

    ss << (*this).balance();
    ss >> balance_str;

    _ss << (*this).content();
    _ss >> data_str;

    std::string ret = "";

    ret += "Node:␣" + data_str + "␣balance:␣" + balance_str + "\n";

    if((*this).left() != 0) {
      ret += "Left\n" + (std::string) *(*this).left();
    }

    if((*this).right() != 0) {
      ret += "Right\n" + (std::string) *(*this).right();
    }

    return ret;
  }
};

template <typename V, bool se = false>
```

```
  class avl_tree_bp_node : public avl_tree_bp_node_base<V, se,
      avl_tree_bp_node<V, se> > {
  private:
    avl_tree_bp_node() {}
  public:
    typedef avl_tree_bp_node_base<V, se, avl_tree_bp_node<V, se> >
        superclass;
    avl_tree_bp_node(V const& v) : superclass(v) {
    }
  };

}
```

Appendix D.6 red_black_tree_balance.h++

```
/*

   Balancing mechanism for the red-black tree

   Authors: Frej Soya, Joergen T. Haahr, June 2008
            Bo Simonsen, December 2008

*/

#include "tree_balance.h++"

#ifndef _CPHSTL_RED_BLACK_TREE_BALANCE_H
#define _CPHSTL_RED_BLACK_TREE_BALANCE_H

namespace cphstl {
  template <typename N >
  class red_black_tree_balance_policy : public tree_balance_policy<N>
      {
  public:
    typedef N node_type;
  public:
    static void insert_fixup(node_type* z, node_type* header) {
      (*z).colour(N::red);
      node_type* y;

      while (z != (*header).parent() && (*(*z).parent()).is_red()) {
        if ((*(*z).parent()).is_left()) {
          y = (*(*z).grandparent()).right();
          if ((*y).is_red()) {
            (*(*z).parent()).colour(N::black);
            (*y).colour(N::black);
            (*(*z).grandparent()).colour(N::red);
            z = (*z).grandparent();
          } else {
            if ((*z).is_right()) {
              z = (*z).parent();
              (*z).rotate_left(header);
            }
            (*(*z).parent()).colour(N::black);
            (*(*z).grandparent()).colour(N::red);
            (*(*z).grandparent()).rotate_right(header);
          }
        } else {
          y = (*(*z).grandparent()).left();
          if ((*y).is_red()) {
            (*(*z).parent()).colour(N::black);
            (*y).colour(N::black);
            (*(*z).grandparent()).colour(N::red);
            z = (*z).grandparent();
          } else {
```

```
        if ((*z).is_left()) {
          z = (*z).parent();
          (*z).rotate_right(header);
        }
        (*(*z).parent()).colour(N::black);
        (*(*z).grandparent()).colour(N::red);
        (*(*z).grandparent()).rotate_left(header);
      }
    }
  }
  (*(*header).parent()).colour(N::black);
}
static void erase_fixup(node_type* x, node_type* y, node_type*
    header) {
  node_type* w;
  while (x != (*header).parent() && (*x).is_black()) {
    if ((*x).is_left()) {
      w = (*(*x).parent()).right();
      if ((*w).is_red()) {
        (*w).colour(N::black);
        (*(*x).parent()).colour(N::red);
        (*(*x).parent()).rotate_left(header);
        w = (*(*x).parent()).right();
      }
      if ((*(*w).left()).is_black() &&
          (*(*w).right()).is_black()) {
        (*w).colour(N::red);
        x = (*x).parent();
      } else {
        if ((*(*w).right()).is_black()) {
          (*w).colour(N::red);
          (*w).rotate_right(header);
          w = (*(*x).parent()).right();
        }
        (*w).colour((*(*x).parent()).colour());
        (*(*x).parent()).colour(N::black);
        (*(*w).right()).colour(N::black);
        (*(*x).parent()).rotate_left(header);
        x = (*header).parent();
      }
    } else if((*x).is_right()) {
      w = (*(*x).parent()).left();
      if ((*w).is_red()) {
        (*w).colour(N::black);
        (*(*x).parent()).colour(N::red);
        (*(*x).parent()).rotate_right(header);
        w = (*(*x).parent()).left();
      }
      if ((*(*w).right()).is_black() && (*(*w).left()).is_black())
          {
        (*w).colour(N::red);
        x = (*x).parent();
      }
      else {
        if ((*(*w).left()).is_black()) {
          (*w).colour(N::red);
          (*w).rotate_left(header);
          w = (*(*x).parent()).left();
        }
        (*w).colour((*(*x).parent()).colour());
        (*(*x).parent()).colour(N::black);
        (*(*w).left()).colour(N::black);
        (*(*x).parent()).rotate_right(header);
        x = (*header).parent();
      }
```

```
        }
      }
    }
  };
}
#endif
```

Appendix D.7 red_black_tree_node.h++

```
/*

  Red_black_tree node class

  Authors: Frej Soya, Joergen T. Haahr, June 2008
           Bo Simonsen, December 2008

*/

#include <string>
#include <iostream>

#include "tree_node.h++"
#include "tree_se_node.h++"
#include "tree_node_shared.h++"

#ifndef _CPHSTL_RED_BLACK_TREE_NODE_H
#define _CPHSTL_RED_BLACK_TREE_NODE_H

namespace cphstl {

  template <typename V, bool se, typename N>
  class red_black_tree_node_base : public tree_node_selector<V, se, N
      >::node_type {
  private:
    typedef typename tree_node_selector<V, se, N>::node_type
        superclass;
    typedef N node_type;
  public:
    typedef V value_type;
    enum colour_type { black = true, red = false};

    /* We don't want default construction */
    red_black_tree_node_base() {
    }
  public:
    red_black_tree_node_base(value_type const& v) : superclass(v) {
      (*this)._colour = red;
    }
    colour_type colour() {
      return (*this)._colour;
    }
    void colour(colour_type const& c) {
      if(this == 0)
        return;

      (*this)._colour = c;
    }
    bool is_red() const {
      if(this == 0)
        return false;
      return (*this)._colour == red;
    }
    bool is_black() const {
      if(this == 0)
        return true;
```

```
      return (*this)._colour == black;
    }
    void copy_aux(node_type* n) {
      (*this).colour((*n).colour());
    }
    operator std::string() const {
      std::stringstream ss;
      std::string colour_str;
      std::string data_str;

      if(_colour == black) {
        colour_str = std::string("black");
      } else {
        colour_str = std::string("red");
      }

      ss << (*this)._data;
      ss >> data_str;

      std::string ret = "";
      ret += "Node: " + data_str + " colour: " + colour_str + "\n";

      if((*this)._left != 0) {
        ret += "Left\n" + (std::string) *(*this)._left;
      }

      if((*this)._right != 0) {
        ret += "Right\n" + (std::string) *(*this)._right;
      }

      return ret;
    }
  private:
    colour_type _colour;
  };

  template <typename V, bool se = false>
  class red_black_tree_node : public red_black_tree_node_base<V, se,
      red_black_tree_node<V, se> > {
  private:
    red_black_tree_node() {}
  public:
    typedef red_black_tree_node_base<V, se, red_black_tree_node<V, se>
        > superclass;
    red_black_tree_node(V const& v) : superclass(v) {
    }
  };

}
#endif
```

Appendix D.8 packed_red_black_tree_node.h++

```
/*

  Red_black_tree node class which uses bitpacking

  Authors: Frej Soya, Joergen T. Haahr, June 2008
           Bo Simonsen, December 2008

*/

#include <string>
#include <iostream>
```

```
#include "tree_node.h++"
#include "tree_se_node.h++"
#include "tree_node_shared.h++"

namespace cphstl {

  template <typename V, bool se, typename N>
  class red_black_tree_bp_node_base : public tree_node_selector<V, se,
       N>::node_type  {
  private:
    typedef typename tree_node_selector<V, se, N >::node_type
        superclass;
    typedef N node_type;
  public:
    typedef V value_type;

    enum colour_type { black = true, red = false};

  private:
    /* We don't want default construction */
    red_black_tree_bp_node_base() {
    }
  public:
    red_black_tree_bp_node_base(value_type const& v) : superclass(v) {
    }
    node_type* parent() const {
      return (node_type*) ( ((int) (*this)._parent) & -2);
    }

    void parent(node_type* n) {
      int color_indicator = ((int) (*this)._parent) & 1;
      (*this)._parent=(node_type*) ( ((int) n) + color_indicator);
    }

    colour_type colour() const {
      if ( ((int) (*this)._parent) & 1 == 1) {
        return black;
      } else {
        return red;
      }
    }
    void colour(colour_type const& c) {
      if(this == 0)
        return;

      if(c == black) {
        (*this)._parent = (node_type*) ( ((int) (*this)._parent) | 1);
      }
      else {
        (*this)._parent= (node_type*) ( ((int) (*this)._parent) & -2);
      }
    }

    bool is_red() const {
      if(this == 0)
        return false;

      return (*this).colour() == red;
    }
    bool is_black() const {
      if(this == 0)
        return true;

      return (*this).colour() == black;
```

```
    }

    void copy_aux(node_type* n) {
    }

    operator std::string() const {
      std::stringstream ss;
      std::string colour_str;
      std::string data_str;

      if((*this).colour() == black) {
        colour_str = std::string("black");
      } else {
        colour_str = std::string("red");
      }

      ss << (*this).content();
      ss >> data_str;

      std::string ret = "";

      ret += "Node: " + data_str + " colour: " + colour_str + "\n";

      if((*this).left() != 0) {
        ret += "Left\n" + (std::string) *(*this).left();
      }

      if((*this).right() != 0) {
        ret += "Right\n" + (std::string) *(*this).right();
      }

      return ret;
    }
  };

  template <typename V, bool se = false>
  class red_black_tree_bp_node : public red_black_tree_bp_node_base<V,
       se, red_black_tree_bp_node<V, se> > {
  private:
    red_black_tree_bp_node() {}
  public:
    typedef red_black_tree_bp_node_base<V, se, red_black_tree_bp_node<
        V, se> > superclass;
    red_black_tree_bp_node(V const& v) : superclass(v) {
    }
  };


}
```

Appendix D.9 splay_tree_balance.h++

```
/*

   Balancing mechanism for the Splay tree

   Author: Bo Simonsen, December 2008
*/

#include "tree_balance.h++"

#ifndef _CPHSTL_SPLAY_TREE_BALANCE_H
#define _CPHSTL_SPLAY_TREE_BALANCE_H

namespace cphstl {
```

```
template <typename N>
class splay_tree_balance_policy : public tree_balance_policy<N> {
public:
  typedef N node_type;
private:
  static void splay(node_type* x, node_type* header) {
    node_type* y = (*x).parent();
    node_type* z = (*y).parent();

    /* We will stop when x is the new root */
    while((*x).parent() != header) {

      if(z == header) {
        if((*x).is_left()) {
          (*y).rotate_right(header);
        }
        else {
         (*y).rotate_left(header);
        }
      }
      else {
        /* zig zag */
        /*
              Z             Z
             /             /
            Y     ->      X     ->      X
             \           /             / \
              X         Y             Y   Z

        */

        if((*y).right() == x && (*z).left() == y) {
          (*y).rotate_left(header);
          (*z).rotate_right(header);
        }
        else if((*y).left() == x && (*z).right() == y) {
          (*y).rotate_right(header);
          (*z).rotate_left(header);
        }

        /* zig zig */
        /*
               Z          Z           X
              /          /             \
             Y     ->   X        ->     Z
            /           \               \
           X             Y               Y

         */

        else if((*y).left() == x && (*z).left() == y) {
          (*y).rotate_right(header);
          (*z).rotate_right(header);
        }

        /* zig zig - symmetric */
        else if((*y).right() == x && (*z).right() == y) {
          (*y).rotate_left(header);
          (*z).rotate_left(header);
        }
      }

      y = (*x).parent();
      z = (*y).parent();
```

```
      }
    }
  public:
    static void insert_fixup(node_type* z, node_type* header) {
      splay(z, header);
    }
    static void erase_fixup(node_type* z, node_type* y, node_type*
        header) {
      splay(z, header);
    }
    static void touch(node_type* z, node_type* header) {
      splay(z, header);
    }
  };
}
#endif
```

Appendix D.10 splay_tree_node.h++

```
/*

  Splay node class

  Authors: Bo Simonsen, June 2008

*/

#include <string>
#include <iostream>

#include "tree_node.h++"
#include "tree_se_node.h++"
#include "tree_node_shared.h++"

namespace cphstl {

  template <typename V, bool se, typename N>
  class splay_tree_node_base : public tree_node_selector<V, se, N>::
      node_type  {
  private:
    typedef typename tree_node_selector<V, se, N >::node_type
        superclass;
  public:
    typedef V value_type;
  public:
    splay_tree_node_base(value_type const& v) : superclass(v) {
    }
  private:
    /* We don't want default construction */
    splay_tree_node_base() {
    }
  };

  template <typename V, bool se = false>
  class splay_tree_node : public splay_tree_node_base<V, se,
      splay_tree_node<V, se> > {
  private:
    splay_tree_node() {}
  public:
    typedef splay_tree_node_base<V, se, splay_tree_node<V, se> >
        superclass;
    splay_tree_node(V const& v) : superclass(v) {
    }
  };
```

```
}
```

Appendix E. Safe Tree Framework: Test programs

Appendix E.1 test.c++

```
#include "node_iterator.h++"
#include "stl_set.h++"
#include "stl_multiset.h++"
#include "tree.h++"
#include "red_black_tree_balance.h++"
#include "red_black_tree_node.h++"
#include "red_black_tree_bp_node.h++"
#include "splay_tree_balance.h++"
#include "splay_tree_node.h++"
#include "avl_tree_balance.h++"
#include "avl_tree_node.h++"
#include "avl_tree_bp_node.h++"
#include "aa_tree_node.h++"
#include "aa_tree_balance.h++"
#if 0
#include "rank_search_node.h++"
#include "rank_search.h++"
#endif
#include <algorithm>
#include <cassert>

class print_int {
public:
  void operator()(int const& i) {
    std::cout << "Item:␣" << i << std::endl;
  }
};

int main() {
  typedef cphstl::set<int,
                      std::less<int>,
                      std::allocator<int>,
                      cphstl::tree<int, int, cphstl::unnamed::identity
                          <int>, std::less<int>, std::allocator<int>,
                         cphstl::avl_tree_node<int, true>,
                         cphstl::avl_tree_balance_policy<cphstl::
                             avl_tree_node<int, true> > >,
                      cphstl::node_iterator<cphstl::avl_tree_node<int,
                           true>, false>,
                      cphstl::node_iterator<cphstl::avl_tree_node<int,
                           true>, true> > cont_a;

  typedef cphstl::set<int,
                      std::less<int>,
                      std::allocator<int>,
                      cphstl::tree<int>,
                      cphstl::node_iterator<cphstl::
                          red_black_tree_node<int>, false>,
                      cphstl::node_iterator<cphstl::
                          red_black_tree_node<int>, true> > cont_r;
  typedef cphstl::set<int,
                      std::less<int>,
                      std::allocator<int>,
                      cphstl::tree<int, int, cphstl::unnamed::identity
                          <int>, std::less<int>, std::allocator<int>,
                         cphstl::splay_tree_node<int>,
                         cphstl::splay_tree_balance_policy<cphstl::
                             splay_tree_node<int> > >,
```

```
                    cphstl::node_iterator<cphstl::splay_tree_node<
                        int>, false>,
                    cphstl::node_iterator<cphstl::splay_tree_node<
                        int>, true> > cont_s;

typedef cphstl::multiset<int,
                    std::less<int>,
                    std::allocator<int>,
                    cphstl::tree<int, int, cphstl::unnamed::identity
                        <int>, std::less<int>, std::allocator<int>,
                       cphstl::avl_tree_node<int>,
                       cphstl::avl_tree_balance_policy<cphstl::
                           avl_tree_node<int> >, true>,
                    cphstl::node_iterator<cphstl::avl_tree_node<int
                        >, false>,
                    cphstl::node_iterator<cphstl::avl_tree_node<int
                        >, true> > mcont_r;
cont_a s;
s.insert(5);
s.insert(3);
s.insert(9);
s.insert(25);
s.insert(1);
std::for_each(s.begin(), s.end(), print_int());
s.erase(3);
s.insert(27);
std::for_each(s.begin(), s.end(), print_int());
cont_a ss(s);
assert(s == ss);
std::cout << "Copy␣cons" << std::endl;
std::for_each(ss.begin(), ss.end(), print_int());
std::cout << std::string(ss);
std::cout << "Copy␣cons..." << std::endl;
std::for_each(ss.begin(), ss.end(), print_int());
assert(*ss.find(9) == 9);
assert(*ss.find(25) == 25);
assert(*ss.find(1) == 1);
std::cout << "Cont:␣T" << std::endl;
cont_r t;
t.insert(5);
std::cout << std::string(t) << std::endl;
t.insert(3);
std::cout << std::string(t) << std::endl;
t.insert(9);
t.insert(25);
t.insert(1);
t.insert(7);
std::cout << std::string(t) << std::endl;
std::for_each(t.begin(), t.end(), print_int());
assert(*t.find(1) == 1);
assert(*t.find(9) == 9);
assert(*t.find(3) == 3);
assert(*t.find(25) == 25);
cont_r t2(t);
assert(t == t2);
assert(*t2.find(1) == 1);
std::cout << "T2:" << std::endl;
std::for_each(t2.begin(), t2.end(), print_int());

cont_r::iterator it = t2.begin();
for(int i=0; i < 100; i++) {
  if(it != t2.end())
    std::cout << "Elem:␣" << *it << std::endl;
  ++it;
}
```

```
std::cout << "1" << std::endl;
t.erase(9);
std::cout << "2" << std::endl;
t.erase(25);
std::cout << "3" << std::endl;
t.erase(5);
std::cout << std::string(t) << std::endl;

cont_s q;

q.insert(5);
q.insert(9);
q.insert(1);
q.insert(25);
q.insert(7);
q.insert(33);
q.insert(27);
q.insert(2);
q.insert(3);
q.insert(45);
assert(*q.find(25) == 25);
assert(*q.find(5) == 5);
assert(*q.find(7) == 7);
assert(*q.find(33) == 33);
std::cout << std::string(q) << std::endl;
mcont_r ms;
ms.insert(5);
ms.insert(5);
ms.insert(5);
ms.insert(7);
ms.insert(1);
ms.insert(9);
ms.insert(5);
ms.insert(7);
std::cout << "5␣count" << ms.count(5) << std::endl;
assert(ms.count(5) == 4);
mcont_r ms2(ms);
std::for_each(ms.begin(), ms.end(), print_int());
std::cout << "Occurances␣of␣5" << std::endl;
std::for_each(ms.lower_bound(5), ms.upper_bound(5), print_int());
std::cout << "Number:␣" << std::distance(ms.lower_bound(5), ms.
    upper_bound(5)) << std::endl;
ms.erase(++ms.lower_bound(5));
assert(ms.count(5) == 3);
std::cout << "Number:␣" << std::distance(ms.lower_bound(5), ms.
    upper_bound(5)) << std::endl;
std::for_each(ms.lower_bound(5), ms.upper_bound(5), print_int());
std::cout << "Occurances␣deleted:␣" << ms.erase(5) << std::endl;
ms.erase(ms.lower_bound(7));
std::cout << "Now:" << std::endl;
std::for_each(ms.begin(), ms.end(), print_int());
std::cout << std::string(ms) << std::endl;
assert(ms.find(5) == ms.end());

std::cout << "Ms2:" << std::endl;
ms2.insert(ms2.lower_bound(5), 5);
ms2.insert(ms2.lower_bound(5), 7);
std::for_each(ms2.begin(), ms2.end(), print_int());
assert(*ms2.find(5) == 5);
assert(*ms2.find(1) == 1);
assert(*ms2.find(7) == 7);
assert(*ms2.find(9) == 9);

mcont_r ms3;
```

```
ms3.swap(ms);

typedef cphstl::set<int,
                    std::less<int>,
                    std::allocator<int>,
                    cphstl::tree<int, int, cphstl::unnamed::identity
                        <int>, std::less<int>, std::allocator<int>,
                       cphstl::red_black_tree_bp_node<int, false>,
                       cphstl::red_black_tree_balance_policy<cphstl
                           ::red_black_tree_bp_node<int, false> >/*,
                            false*/ >,
                    cphstl::node_iterator<cphstl::
                        red_black_tree_bp_node<int, false>, false>,
                    cphstl::node_iterator<cphstl::
                        red_black_tree_bp_node<int, false>, true> >
                        cont_rb;

cont_rb w;

srand(time(NULL));

for(int i=0; i < 20; ++i) {
  w.insert(rand() % 100);
}
std::cout << std::string(w);

/* 4 word node */
typedef cphstl::set<int,
                    std::less<int>,
                    std::allocator<int>,
                    cphstl::tree<int, int, cphstl::unnamed::identity
                        <int>, std::less<int>, std::allocator<int>,
                       cphstl::avl_tree_bp_node<int, true>,
                       cphstl::avl_tree_balance_policy<cphstl::
                           avl_tree_bp_node<int, true> >/*, false*/
                           >,
                    cphstl::node_iterator<cphstl::avl_tree_bp_node<
                        int, true>, false>,
                    cphstl::node_iterator<cphstl::avl_tree_bp_node<
                        int, true>, true> > cont_ab;

cont_ab z;

srand(time(NULL));

for(int i=0; i < 20; ++i) {
  z.insert(rand() % 100);
}
std::cout << std::string(z);
std::for_each(z.begin(), z.end(), print_int());

/* 4 word node */
typedef cphstl::set<int,
                    std::less<int>,
                    std::allocator<int>,
                    cphstl::tree<int, int, cphstl::unnamed::identity
                        <int>, std::less<int>, std::allocator<int>,
                       cphstl::aa_tree_node<int, true>,
                       cphstl::aa_tree_balance_policy<cphstl::
                           aa_tree_node<int, true> >/*, false*/ >,
                    cphstl::node_iterator<cphstl::aa_tree_node<int,
                        true>, false>,
                    cphstl::node_iterator<cphstl::aa_tree_node<int,
                        true>, true> > cont_aa;
```

```
  cont_aa y;

  std::stack<int> stack;
  srand(time(NULL));

  for(int i=0; i < 20; ++i) {
    std::cout << i << std::endl;
    int tmp = rand() % 100;
    std::pair<cont_aa::iterator, bool> p = y.insert(tmp);
    if(p.second)
      stack.push(tmp);
  }

  while(!stack.empty()) {
    int tmp = stack.top();
    stack.pop();
    assert(*y.find(tmp) == tmp);
    y.erase(tmp);
  }

  assert(y.size() == 0);

#if 0
  /* 4 word node */
  typedef cphstl::rank_search_node< int, false > rank_node;
  typedef cphstl::set<int,
                      std::less<int>,
                      std::allocator<int>,
                      cphstl::tree<int, int, cphstl::unnamed::identity
                          <int>, std::less<int>, std::allocator<int>,
                         rank_node,
                         cphstl::avl_tree_balance_policy< rank_node >,
                              false, cphstl::rank_search<int, int,
                    cphstl::unnamed::identity<int>, cphstl::
                        comparator_proxy<std::less<int> >,
                          rank_node, cphstl::avl_tree_balance_policy <
                              rank_node>, false > >,
                         cphstl::node_iterator<rank_node, false>,
                         cphstl::node_iterator<rank_node, true > >
                             cont_rank;

  cont_rank cr;
  cr.insert(5);
  cr.insert(3);
  cr.insert(9);

  for(unsigned int i=1; i < cr.size(); ++i) {
    std::cout << *cr.select(i) << std::endl;
  }
#endif
}
```

Appendix E.2 iter_test.c++

```
#include "node_iterator.h++"
#include "stl_set.h++"
#include "stl_multiset.h++"
#include "tree.h++"
#include "red_black_tree_balance.h++"
#include "red_black_tree_node.h++"
#include "red_black_tree_bp_node.h++"
#include "splay_tree_balance.h++"
#include "splay_tree_node.h++"
#include "avl_tree_bp_node.h++"
#include <algorithm>
```

```
#include <cassert>
#include <iostream>

int main() {
  typedef cphstl::set<int,
                      std::less<int>,
                      std::allocator<int>,
                      cphstl::tree<int, int, cphstl::unnamed::identity
                          <int>, std::less<int>, std::allocator<int>,
                         cphstl::red_black_tree_node<int, false>,
                         cphstl::red_black_tree_balance_policy<cphstl
                             ::red_black_tree_node<int, false> > >,
                      cphstl::node_iterator<cphstl::
                          red_black_tree_node<int, false>, false>,
                      cphstl::node_iterator<cphstl::
                          red_black_tree_node<int, false>, true> >
                          cont;

  int arr[] = {1,6,3,8,7,5,4};
  cont c(arr, arr+7);

  assert(*c.begin() == 1);
  assert(--c.begin() == c.end());
  assert(*(--c.end()) == 8);
  assert(++c.end() == c.begin());
  assert(*(++c.begin()) == 3);

  std::cout << std::string(c);

  std::copy(c.begin(), c.end(), std::ostream_iterator<int>( std::cout,
       "\n" ));
}
```

Appendix F. B-Tree

Appendix F.1 btree.h++

```
#include <vector>

#include "type.h++"
#include "set_helper.c++"
#include "btree_node.h++"
#include "btree_cmp.h++"
#include "btree_iterator.h++"

#define BRANCH 20

namespace cphstl {
  template <typename K,
            typename V = K,
            typename F = cphstl::unnamed::identity<K>,
            typename C = std::less<K>,
            typename A = std::allocator<V>,
            typename N = cphstl::btree_node<K, V, F, C>,
            bool is_multiset = false
  >
  class btree {
  private:
    typedef N node_type;
    typedef F functor_type;
  protected:
    typedef K key_type;
    typedef V value_type;
    typedef C key_compare;
```

```
  typedef typename A::template rebind<node_type>::other
      allocator_type;
public:
  typedef std::size_t size_type;
  typedef value_type* pointer;
  typedef value_type const* const_pointer;
  typedef value_type& reference;
  typedef value_type const& const_reference;
  typedef ptrdiff_t difference_type;

  typedef btree_concrete_iterator<node_type, typename node_type::
      key_list::iterator> concrete_iterator;
protected:
  node_type* root;
  node_type* leftmost;
  node_type* rightmost;
  int branch;
  size_type tree_size;
  typedef btree_comparator<V, C, F> iter_key_comp_type;
  iter_key_comp_type iter_key_comp;
  allocator_type allocator;
  key_compare comparator;

  concrete_iterator insert_non_full(node_type * n, const value_type&
       t) {

    typename node_type::key_list::iterator pos = std::upper_bound(n
        ->keys().begin(), n->keys().end(), t, iter_key_comp);

    typename node_type::child_list::iterator cpos = n->children().
        begin() + (pos - n->keys().begin());

    if ((*n).leaf()) {
      (*this).tree_size++; // all insert pass through here
      return concrete_iterator(n,
                               n->keys().insert(pos, t),
                               comparator);
    }
    else {
      if ((*cpos)->size() == 2*branch-1) {
        split_child(n, pos, cpos+1,*cpos);

        pos = std::upper_bound(n->keys().begin(),
                               n->keys().end(),
                               t,
                               iter_key_comp);

        cpos = n->children().begin() + (pos - n->keys().begin());
      }

      return insert_non_full(*cpos,t);
    }
  }

  void merge_nodes(node_type * left, node_type * right, const
      value_type &t) {

    left->keys().insert(left->keys().end(), t);
    left->keys().insert(left->keys().end(),
                        right->keys().begin(),
                        right->keys().end());

    if (!left->leaf()) {
```

```
    for (typename node_type::child_list::iterator it = right->
        children().begin();
         it != right->children().end();
         ++it) {
      (*it)->parent() = left;
    }
    left->children().insert(left->children().end(),
                            right->children().begin(),
                            right->children().end());

    right->children().clear();
  }
  if(right == rightmost) {
    rightmost = left;
  }

  (*right).~N();
  allocator.deallocate(right, 1);
}

void delete_from(node_type * n, const key_type& t) {

  typename node_type::key_list::iterator pos = std::lower_bound(n
      ->keys().begin(), n->keys().end(), t, iter_key_comp);

  if (n->leaf()) {
    if ((pos != n->keys().end()) && !iter_key_comp(*pos,t) && !
        iter_key_comp(t,*pos)) {
      n->keys().erase(pos);
      (*this).tree_size--;
    }
    return;
  }

  if ((pos != n->keys().end()) && !iter_key_comp(*pos,t) && !
      iter_key_comp(t,*pos)) {
    typename node_type::child_list::iterator cpos = n->children().
        begin() + (pos - n->keys().begin());
    node_type * left_child = *cpos;
    node_type * right_child = *(cpos+1);

    if (left_child->size() >= branch) {
      node_type* c = left_child;

      while(!c->leaf()) {
        c = c->children().back();
      }

      *pos = c->keys().back();

      delete_from(left_child, F()(*pos));
    }
    else if (right_child->size() >= branch) {
      node_type* c = right_child;

      while(!c->leaf()) {
        c = c->children().front();
      }

      *pos = c->keys().front();

      delete_from(right_child, F()(*pos));
    }

    else {
```

```
    value_type tt = *pos;
    merge_nodes(left_child, right_child, tt);
    n->keys().erase(pos);
    n->children().erase(cpos+1);
    delete_from(left_child, F()(tt)); // todo, det kan vi vel
        goere her.
  }
}
else {
  typename node_type::child_list::iterator cpos = n->children().
      begin() + (pos - n->keys().begin());
  node_type* sub_tree;
  node_type* left_sibling;
  node_type* right_sibling;

  if (pos == n->keys().end()) {
    right_sibling=0;
  }
  else {
    right_sibling = *(cpos+1);
  }
  if (pos == n->keys().begin()) {
    left_sibling = 0;
  }
  else {
    left_sibling = *(cpos-1);
  }

  sub_tree = *cpos;

  if (sub_tree->size()==branch-1) {

    if (right_sibling && right_sibling->size() > branch-1) {
      sub_tree->keys().insert(sub_tree->keys().end(),*pos);
      if (!sub_tree->leaf()) {
        node_type* c = right_sibling->children().front();
        c->parent() = sub_tree;
        sub_tree->children().insert(sub_tree->children().end(),
            c);
        right_sibling->children().erase(right_sibling->children
            ().begin());
      }

      *pos = right_sibling->keys().front();
      right_sibling->keys().erase(right_sibling->keys().begin())
          ;
    }
    else if (left_sibling && left_sibling->size() > branch-1) {
      --pos;
      sub_tree->keys().insert(sub_tree->keys().begin(),*pos);

      if (!sub_tree->leaf()) {
        node_type* c = left_sibling->children().back();
        c->parent() = sub_tree;
        sub_tree->children().insert(sub_tree->children().begin()
            , c);
        left_sibling->children().pop_back();
      }

      *pos = left_sibling->keys().back();
      left_sibling->keys().pop_back();
    }
    else {
      if (right_sibling) {
        merge_nodes(sub_tree,right_sibling,(*pos));
```

```
          n->keys().erase(pos);
          n->children().erase(cpos+1);
        }
        else {
          if(pos == n->keys().end()) {
            --pos;
          }

          value_type p = *pos;
          n->children().erase(cpos);
          n->keys().erase(pos);

          merge_nodes(left_sibling,sub_tree,p);
          sub_tree = left_sibling;
        }
      }
    }
    delete_from(sub_tree,t);
  }
}

void split_child(node_type * parent, typename node_type::key_list
    ::iterator keyindex,
                 typename node_type::child_list::iterator
                     childindex, node_type * target_node ) {

  node_type* new_node = allocator.allocate(1);
  new (new_node) node_type(target_node,branch);

  if(target_node == rightmost) {
    rightmost = new_node;
  }

  new_node->parent() = parent;
  // update the parent
  parent->keys().insert(keyindex,target_node->keys()[branch-1]);
  parent->children().insert(childindex, new_node);

  // remove the deadwood
  target_node->keys().resize(branch-1);
  if (!target_node->leaf()) {
    target_node->children().resize(branch);
  }
}

concrete_iterator insert_elem(const value_type& t) {
  node_type *r = root;
  if (root->size() == 2 * branch-1) {
    root = allocator.allocate(1);
    new (root) node_type(branch,comparator);

    r->parent() = root;
    root->leaf() = false;
    root->children().insert(root->children().begin(),r);
    split_child(root, root->keys().begin(), root->children().begin
        () + 1,r);
    return insert_non_full(root, t);
  }
  else {
    return insert_non_full(r, t);
  }
}

void delete_elem(const key_type &t) {
  node_type * r = root;
```

```
    delete_from(r,t);
    if (r->size() == 0) {
      if (!r->leaf()) {
        root = r->children()[0];
        r->children().resize(0);

        (*r).~N();
        allocator.deallocate(r, 1);

        root->parent() = 0;
      }
    }
  }

  concrete_iterator bound(key_type const& key, bool upper) const {
    node_type* n = root;

    do {
      typename node_type::key_list::iterator pos = upper ?
        std::upper_bound(n->keys().begin(), n->keys().end(), key,
            iter_key_comp)
        : std::lower_bound(n->keys().begin(), n->keys().end(), key,
            iter_key_comp);

      if( pos != n->keys().end() && !(*this).iter_key_comp(*pos,key)
           && !(*this).iter_key_comp(key,*pos)) {
        return concrete_iterator(n,pos,comparator);
      }

      if(n->leaf()) {
        if(pos == n->keys().end()) {
          if(n != rightmost) {
            concrete_iterator c(n,pos-1,comparator);
            c.successor();
            return c;
          }
        }
        return concrete_iterator(n,pos,comparator);
      }
      else {
        n = *( n->children().begin() + (pos - n->keys().begin()) );
      }
    } while(1);
  }

  void destroy_tree(node_type* n) {
    for(typename node_type::child_list::iterator it = n->children().
        begin(); it != n->children().end(); ++it) {
      destroy_tree(*it);
    }

    (*n).~N();
    allocator.deallocate(n, 1);
  }

public:
  /* OK */
  btree(const btree& bt) : btree() {
    insert(bt.begin(),bt.end());
    comparator = bt.comparator;
    allocator = bt.allocator;
    iter_key_comp = iter_key_comp_type(comparator);
  }
```

```
btree(C const& c, A const& a) {
  branch=BRANCH;
  (*this).tree_size=0;

  root = allocator.allocate(1);
  new (root) node_type(branch,comparator);

  rightmost = leftmost = root;
  root->leaf() = true;
  allocator = a;
  comparator = c;
  iter_key_comp = iter_key_comp_type(comparator);
}

~btree() {
  destroy_tree(root);
}

allocator_type get_allocator() {
  return allocator;
}

key_compare key_comp() const {
  return comparator;
}

concrete_iterator begin() const {
  return concrete_iterator(leftmost,leftmost->keys().begin(),
      comparator);
}

concrete_iterator end() const {
  return concrete_iterator(rightmost,rightmost->keys().end(),
      comparator);
}

/* Not OK */
size_type erase(const key_type &key) {
  size_type c=0;
  size_type size_orig=(*this).tree_size;
  size_type size=(*this).tree_size;

  delete_elem(key);

  while(size!=(*this).tree_size) {
    size=(*this).tree_size;
    delete_elem(key);
  }
  return size_orig - (*this).tree_size;
}

void erase(concrete_iterator it) {
  erase(*it);
}

concrete_iterator insert(concrete_iterator position, const
    value_type& t) {
  return insert(t).first;
}

std::pair<concrete_iterator,bool> insert( const value_type &t) {
  concrete_iterator f = lower_bound(F()(t));
  if (f != end()) {
    return std::pair<concrete_iterator,bool>(f,false);
  }
```

```
      return std::pair<concrete_iterator,bool> (insert_elem(t),true);
    }

    template <typename I>
    void insert(I from, I to) {
      while (from != to) {
        insert_unique(*from);
        ++from;
      }
    }

    /* OK */
    concrete_iterator upper_bound(const key_type &key) const {
      return bound(key,true);
    }

    concrete_iterator lower_bound(const key_type &key) const {
      return bound(key,false);
    }

    size_type size() const {
      return (*this).tree_size;
    }

    size_type max_size() const {
      return (*this).allocator.max_size();
    }

    void swap( btree& b) {
      std::swap(root,b.root);
      std::swap(branch,b.branch);
      std::swap((*this).tree_size,b.tree_size);
      std::swap(leftmost,b.leftmost);
      std::swap(rightmost,b.rightmost);
    }
  };
}
```

Appendix F.2 btree_node.h++

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

#include <functional>
#include <algorithm>
#include <memory>

namespace cphstl {

template <typename K,
          typename V,
          typename F,
          typename C,
          typename KL = cphstl::vector<V>
  >
  class btree_node {
    public:
      typedef V value_type;
      typedef std::size_t size_type;
      typedef btree_node<K, V, F, C, KL> node_type;
      typedef btree_comparator<V, C, F> key_comp_type;
      typedef KL key_list;
      /* I really don't like that the childlist is hardcoded */
      typedef cphstl::vector<node_type*> child_list;
    private:
```

```
      key_list _keys;
      child_list _children;
      node_type* _parent;
      bool _leaf;
      key_comp_type _key_comp;
    public:
      btree_node(btree_node* n,int branch) {
        (*this)._leaf=(*n)._leaf;

        if (!_leaf) {
          children() = child_list((*n).children().begin() + branch, (*
              n).children().end());
          for (typename child_list::iterator it = children().begin();
              it != children().end(); it++)
            (*it)->parent() = this;
        }
        keys() = key_list((*n).keys().begin() + branch, (*n).keys().
            end());

        children().reserve(branch*2+1);
        keys().reserve(branch*2);
        parent() = (*n).parent();
        _key_comp = (*n)._key_comp;
      }

      btree_node(int branch, C const& c) {
        (*this).children().reserve(branch*2+1);
        (*this).keys().reserve(branch*2);
        (*this).parent() = 0;
        (*this)._leaf=true;
        (*this)._key_comp = key_comp_type(c);
      }

      ~btree_node() {
      }

      child_list& children() {
        return (*this)._children;
      }
      key_list& keys() {
        return (*this)._keys;
      }
      node_type*& parent() {
        return (*this)._parent;
      }
      size_type size() {
        return (*this)._keys.size();
      }
      bool& leaf() {
        return (*this)._leaf;
      }
  };
}
```

Appendix F.3 btree_cmp.h++

```
#ifndef _CPHSTL_BTREE_CMP_H
#define _CPHSTL_BTREE_CMP_H

namespace cphstl {
  template <typename V, typename C, typename F>
  class btree_comparator {
  public:
    typedef V value_type;
    typedef F functor_type;
```

```
    typedef C comparator_type;

    bool operator()(value_type const& x, value_type const& y) const {
      functor_type f;

      return (*this).c(f(x), f(y));
    }
    btree_comparator() {

    }
    btree_comparator(C const& cc) {
      c = cc;
    }
  private:
    comparator_type c;
  };
}
#endif
```

Appendix F.4 btree_iterator.h++

```
namespace cphstl {
  template <typename N, typename I>
  class btree_concrete_iterator {
  public:
    typedef N node_type;
    typedef I iterator;
    typedef typename N::value_type value_type;
    typedef typename N::key_comp_type key_comp_type;
  private:
    node_type* currentnode;
    iterator keyit;
    key_comp_type key_comp;

  public:
    btree_concrete_iterator() { }
    btree_concrete_iterator(node_type* n, iterator i, key_comp_type) {
      currentnode = n;
      keyit = i;
    }

    value_type& content() const {
      return *(keyit);
    }

    bool operator==(btree_concrete_iterator const& b) const {
      return(b.keyit == keyit);
    }
    bool operator!=(btree_concrete_iterator const& b) const {
      return (b.keyit != keyit);
    }

    void successor() {
      iterator next;
      if(currentnode->leaf()) {
        // if there are no more nodes here the next has to be above us

        if (keyit+1 == currentnode->keys().end()) {
          node_type* last = currentnode;
          currentnode = currentnode->parent();
          while( currentnode && key_comp( currentnode->keys().back(),*
              keyit) ) {
            currentnode = currentnode->parent();
          }
          if(!currentnode) {
```

```
            currentnode = last;
            keyit++;
            return;
          }

          keyit = lower_bound(currentnode->keys().begin(),currentnode
              ->keys().end(),*keyit,key_comp);
        }
        else {
          keyit++;
        }
      }
      else {
        // point to right subtree
        currentnode = *((currentnode->children().begin()+(keyit-
            currentnode->keys().begin())) +1);

        // find leftmost leaf node
        while(!currentnode->leaf()) {
          currentnode = currentnode->children().front();
        }
        keyit = currentnode->keys().begin();
      }
    }

    void predecessor() {
      iterator prev;
      // if we are in a leaf the previous node is either here or in
          the parent
      if(currentnode->leaf()) {
        // if there are no more nodes here the next has to be above us
        if (keyit == currentnode->keys().begin()) {
          node_type* n = currentnode;
          currentnode = currentnode->parent();

          // move up the tree as long as we are smaller than smallest
              element in nodes we pass

          while( key_comp(*keyit, currentnode->keys().front())) {
            currentnode = currentnode->parent();
          }
          keyit = lower_bound(currentnode->keys().begin(),currentnode
              ->keys().end(),*keyit,key_comp);
          --keyit;
        }
        else {
          --keyit;
        }
      }
      else {
        // point to left subtree
        currentnode = *(( currentnode->children().begin()+(keyit-
            currentnode->keys().begin())) );

        // find rightmost leaf node
        while(!currentnode->leaf()) {
          currentnode = currentnode->children().back();
        }
        keyit = currentnode->keys().end() - 1;
      }
    }
  };
}
```

Appendix F.5 main.cpp (Test)

```
#include <functional>
#include "vector.h++"
#include "iv_dynamic_array.h++"
#include "entry_iterator.h++"
#include "obj_iterator.h++"
#include "stl_set.h++"
#include "btree.h++"
#include <set>
#include <sys/time.h>
#include <unistd.h>
#include <cassert>

#define DEFAULT_SIZE 100

int SIZE=DEFAULT_SIZE;

using namespace cphstl;
typedef cphstl::iv_dynamic_array_entry<int, std::allocator<int> >
    ENTRY;
typedef cphstl::iv_dynamic_array<int, std::allocator<int>, ENTRY >
    KERNEL;
typedef cphstl::entry_iterator<KERNEL, ENTRY, false> ITERATOR;
typedef cphstl::entry_iterator<KERNEL, ENTRY, true> CONST_ITERATOR;
typedef cphstl::vector<int, std::allocator<int>, KERNEL, ITERATOR,
    CONST_ITERATOR> VECTOR;
typedef cphstl::btree_node<int, int, cphstl::unnamed::identity<int>,
    std::less<int>, VECTOR> N;
typedef cphstl::btree<int, int, cphstl::unnamed::identity<int>, std::
    less<int>, std::allocator<int>, N> R;
typedef cphstl::set<int,
                    std::less<int>,
                    std::allocator<int>,
                    R,
                    cphstl::obj_iterator<R::concrete_iterator, false>,
                    cphstl::obj_iterator<R::concrete_iterator, true> >
                         B;

void profile_btree(B& tree)
{
        for (int i = 1; i < SIZE; i++)
                tree.insert(i);
}


void profile_find_btree(B& tree)
{
        for (int i = 1; i < SIZE; i++)
                tree.find(i);
}


void profile_del_btree(B& tree)
{
        for (int i = 1; i < SIZE; i++)
                tree.erase(i);
}

int main(int argc, char* argv[])
{

  B bt;
  std::pair<B::iterator, bool> i1, i2, i3, i4;
```

```
  i1 = bt.insert(5);
  i2 = bt.insert(7);
  i3 = bt.insert(1);
  i4 = bt.insert(13);

  bt.erase(7);

  std::cout << "It3:␣" << *i3.first << std::endl;
  assert(*i1.first == 5);
  assert(*i3.first == 1);
  assert(*i4.first == 13);

  B s;
        profile_btree(s);

  for(B::iterator it = s.begin(); it != s.end(); ++it) {
    std::cout << "Item:␣" << *it << std::endl;
  }
  for(B::reverse_iterator it = s.rbegin(); it != s.rend(); ++it) {
    std::cout << "Item:␣" << *it << std::endl;
  }


        profile_find_btree(s);
        profile_del_btree(s);


}
```

Appendix F.6 obj_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_NODE_ITERATOR__
#define __CPHSTL_NODE_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 "type.h++" // defines cphstl::if_then_else

namespace cphstl {

  /* Forward declarations of bridge classes */

  template <typename V, typename A, typename R>
  class list;

  template <typename V, typename C, typename A, typename R>
  class meldable_priority_queue;

  template <typename V, typename C, typename A, typename R, typename I
      , typename J>
  class multiset;
```

```
template <typename V, typename C, typename A, typename R, typename I
    , typename J>
class set;

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
class set_base;

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
class map;

template <typename K, typename V, typename C, typename A, typename R
    , typename I, typename J>
class map_base;


template <typename N, bool is_const = false>
class obj_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 obj_iterator<N, !is_const> complement;
private:

  // types

  typedef typename if_then_else<is_const, N const, N>::type
      node_pointer;

public:

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

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

  template <typename V, typename A, typename R>
  friend class cphstl::list;

  template <typename V, typename C, typename A, typename R>
  friend class cphstl::meldable_priority_queue;

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

  template <typename K, typename V, typename C, typename A, typename
       R, typename I, typename J>
  friend class cphstl::set_base;
```

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

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

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

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

    // structors

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

    // operators

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

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

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

  private:
    // converters to be used by the friends

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

    node_pointer link;

  };

}

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

#endif
```

Appendix F.7 obj_iterator.c++

```
/*
  Implementation of cphstl::obj_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>
  obj_iterator<N, is_const>::obj_iterator()
    : link() {
  }

  // copy constructor

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

  // assignment

  template <typename N, bool is_const>
  obj_iterator<N, is_const>&
  obj_iterator<N, is_const>::operator=(obj_iterator<N, is_const> const
      & a) {
    link = a.link;
    return *this;
  }

  // destructor

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

  // operator*

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

  // operator->

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

  // operator++; pre-increment

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

   // operator++; post-increment

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

```
    ++(*this);
    return temporary;
  }

  // operator--; pre-increment

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

  // operator--; post-increment

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

  // operator==

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

  // operator!=

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

  // parametrized constructor (node -> iterator)

  template <typename N, bool both>
  obj_iterator<N, both>::obj_iterator(node_pointer p)
    : link(p) {
  }

  // conversion operators (iterator -> node)

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

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

  template <typename N, bool is_const>
  obj_iterator<N, is_const>::operator std::string() const {
/*    std::stringstream ss;
    std::string address;
    ss << (int)(char*)((*this).link);
    ss >> address;
```

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

  // representation
  /*
  template <typename N, bool both>
  std::ostream&
  operator<<(std::ostream& s, obj_iterator<N, both> const& i) {
    s  << std::string(i);
    return s;
  }*/

}
```