

# Implementing the AVL-trees for the CPH STL

Stephan Lynge

Project consultant: Jyrki Katajainen

CPH STL Report 2004-1, January 2004. Revised March 2004.

## 1 Introduction

This project is merely a small contribution to the big CPH-STL project. The intention of this project is to implement a data structure that can be used to implement STL's set, multiset, map and multimap, but from here on I will be concentrated only on set.

The implementation will be based on an expansion of existing code, which already contains a broad variety of data structures and a simple test environment, which will be reused in this project.

The main tasks of this project can be described as follows:

- gaining sufficient knowledge of the existing code to be able to expand it
- finding information about a suitable data structure (AVL-trees)
- a bit of catching up on my C++
- implementation
- tests
- report writing - which should contain the following elements:
  - an overview of the complete code so that other students, who might have to continue the work, can have an easier way of getting started
  - a short description of the chosen data structure
  - a comparison/evaluation of the implementation.

All in all this report will contain one section about the implementation, a section about the results of the implementation, and finally a conclusion.

## 2 Implementation

This section will be divided into three parts. In the first part I will try to give a short overview of the code, which was handed out at the beginning of the project [Hervé2003]. The second part will describe the thoughts behind my implementation. Finally, the last part will briefly go through the changes and additions that I have made to the handed out code.

### 2.1 Existing code

Fundamentally the system is build around three main parts:

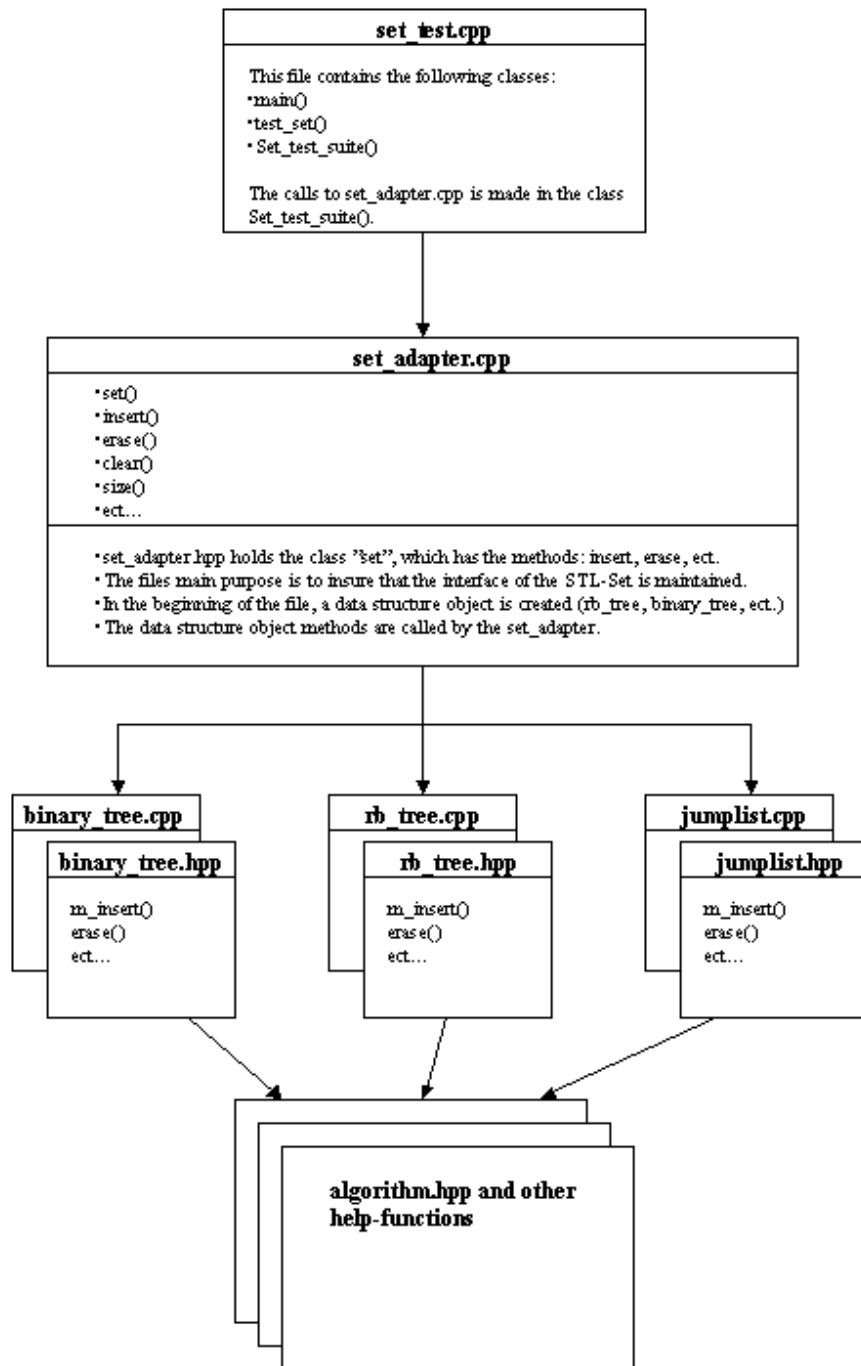
- First we have the test file (`set_test.cpp`), which is used to make set tests, and the function calls are independent of the data structure. The functions which are called are defined in the `set_adapter.hpp` file.
- Next we have the part that implements the set-class and its interface namely the `set_adapter.hpp`. This file sets up an interface to the set-class, and uses one of many data structures, to be able to offer the functionality of a set class.
- Finally, we have the data structure. This structure should support all of the function calls that are made in the `set_adapter.hpp`, basically any data structure can be developed and used, as long as this functionality is supported.

The files can be found in the following directories:

- `set_test.cpp`: `/boost_tree/libs/tree/experiments/`
- `set_adapter.hpp`: `/boost_tree/boost/`
- "data structures": `/boost_tree/boost/tree/`.

### 2.2 Implementation thoughts

As already mentioned I had decided to implement a data structure based on AVL-trees. It is clear that this structure has a lot of similarities with the binary tree structure, which was implemented in the handed-out code. For



that reason, it was a natural choice to expand the binary tree code, so it could support the properties of the AVL-tree.

The expansion should meet the following demands:

- The basic component in AVL-trees is the balancing property, and because of that each node in the binary tree had to be enlarged with a value that could satisfy the need to indicate, whether the specific node is in balance or not.
- The insert operation had to be extended so that after inserting an element, the structure had to be rebalanced. In other words I had to implement a rebalancing function.
- Just like insert the same actions are needed for delete, after each call you have to make sure that the tree is in balance.
- Finally, there will be some small corrections, where the binary tree structure differs from the AVL-tree, for example the tree validation function, which has to be updated so it will check the extended properties of the AVL-tree.

In short, the plan was to make a copy of the files `binary_tree.cpp` and `binary_tree.hpp`, and then modify the new files so that they met the demands listed above. Primarily this meant altering the insert and delete functions and then adding the necessary help functions in the file `algorithm.hpp`. Besides from that I had to make a few changes to files like the `Makefile` and the `set_adapter.hpp`, but these are smaller changes, probably not more than a few lines. The next section will go through all the changes made both for already existing and newly constructed files.

### 2.3 New implementation and changes

The code I have implemented can be divided into two parts. The first contains the files, which already existed, but which were subject to some changes. The second part contains the new files, which I have added. To start with the latter one, the following files have been added:

- `avl_tree.hpp`:
  - This file is based on a copy of the file `binary_tree.hpp`.
  - The class names have been changed from "binary\_something" to "avl\_something", except for places where the binary class should still be used.

- The interface should of course remain the same, because of that, there are only minor surface changes.
- `avl_tree.cpp`:
  - Just like `avl_tree.hpp` this file has been build upon a copy of `binary_tree.cpp`.
  - All functions and classes have been renamed just like `avl_tree.hpp`.
  - Changes have been made to the following functions:
    - \* `m_insert()`:
      - After insertion of a node, the balance of the node will be set to zero, since all nodes are inserted at the bottom of the tree.
      - The "side" of the node is found, where "side" is defined as the new node's position (left/right) relative to its parent.
      - Finally, the function `avl_tree_rebalance()` is called (see more in `algorithm.hpp`) to bring the tree back in to balance.
    - \* `erase()`:
      - After a node is deleted, the function `avl_tree_relink_for_erase()` (see the file `algorithm.hpp` for more information) is used to link the tree together without the deleted node.
      - After the "relink" the function `avl_tree_rebalance()` is called to bring the tree back in balance (see the file `algorithm.hpp` for more information).
    - \* `is_valid()`:
      - The function has been altered so that it checks, if the properties of the AVL-tree hold (the balance of each parents' subtree can never differ more the one and the balance of each node is set to the right value).
    - \* `m_debug_print()`:
      - This function has been changed so the debugging data is represented in a more convenient way (in my opinion).
- `set_test_latex.cpp`:
  - This file has been built based upon `set_test.cpp`, because of that the overall structure is the same.
  - The difference between this file and `set_test.cpp` is that this one writes out the result in a file (a `.tex` file).
  - There are less arguments to this test than to `set_test.cpp` and it is focused on making of a `LATEX` document (with `begin/end` document and table).

- It also runs less tests, than those represented or possible in `set_test.cpp`.

Now I will describe the files which were changed according to the ones handed out. I will try to point out, where the changes have been made and what impact these have.

- `set_adapter.cpp`:

- The only change to this file is in the template in the beginning of the file, so the object will accept the selection of an AVL-tree.

- `algorithm.hpp`:

- This file has been extended with a few functions, which have been specialized to the AVL-structure. Several of these are made on basis of other functions just in the binary tree structure.

- The following functions have been implemented:

- \* `avl_tree_rebalance()`:

- All nodes of the tree from the changed position (inserted or deleted) are checked, until the root is reached or until it can be established, that the balance has not changed.
    - If rotation is needed, the "binary tree rotation"-function is used.
    - The help function `getSide()` is used in the rebalancing (see later on).

- \* `avl_tree_relink_for_erase()`:

- The function is just used to link to an AVL-tree together, then a node is deleted.
    - The function tests whether it can just remove the node or it has to do a swap with the successor and then remove the node.
    - This function looks much like the similar binary function, the main difference is that this function returns the side of the relinked parent, which is used later in connection with calling the rebalance function.

- \* `getSide()`:

- This function returns whether the node is attached to the left or right of its parent.

- \* `avl_tree_get_max_depth()`:

- This function finds the maximum depth of a given node.
    - The function is used as a part of the validation test.

- No attempt has been done, to make this function fast, since it is only used for unit testing and debugging.

- **Makefile:**

- This file has been changed, so it is able to compile the new implementation.
- Furthermore, it has been extended with the possibility of compiling the `set_test_latex.cpp` and run all of the tests, which has been used in this report.

### 3 Results

The enclosed test, A.1, has been performed on a laptop computer with the following specifications:

- **Hardware:**

- IBM R40e
- Mobile Intel Celeron 1.70 GHz
- 128 KB L2 cache
- 256 MB RAM

- **Software:**

- operating system: Red Hat 9 (Shrike) - kernel version: 2.4
- compiler: gcc 3.2.2
- compiler options: `-O3`

At a first glance at the test results, there are only few differences to find between "STD" (red/black-tree) and "AVL" (AVL-tree). The differences between each "pair of tests" are rather small but still notable.

In some of the test categories, it looks like the AVL-tree is the fastest ("Range ins.", "Forw. trans."), but also in these categories the red/black-tree is faster in the end. In the other categories red/black-trees are fastest right from the beginning.

The bottom line is that the red/black-tree "STD" in overall is a fraction faster than the AVL-tree.

## 4 Conclusion

Looking back on this assignment, I think I have achieved the main goals that were described in the beginning of the project, which were:

- getting to understand how the handed-out code was designed
- implementation of another data structure
- the writing of a report which sums up the main work in the project.

The test results of my data structure, didn't show that the structure that I have implemented was faster than the STD's, which was actually expected from the beginning.

Besides these more concrete results, I think that I personally have achieved a lot of things, just to mention a few:

- The fact that I was expected to write this report in English has given me a good opportunity to catch up on my foreign language abilities. (I hope it is understandable :-))
- The level on which the handed-out code was written, forced me to learn some new things about C++ or at least catch up in some areas.
- Finally, the project gave me a good opportunity to try how it is to work on a "free project" like this.

Despite all of this, there are areas which could be further explored, but this project had a time limit, so I had to draw the line somewhere. Just to mention a few of the things, which could be further explored:

- Jyrki suggested that I could use the benchmark tool, which he and a group of students had implemented. It could be exciting to try using the tool to run the tests.
- The data structure may not be usable directly in the CPH-STL architecture, it will properly need some adjustments.
- The number of data structures available could always be extended.

All in all it has been a very exciting assignment. I had a hard time getting started, because it is often hard work to get into someone else's code. The fact that there was no documentation at all, didn't make it easier, but when I got past that obstacle, I was eager to go on.

## Acknowledgements

I would like to thank Hervé Brönnimann for providing his code base for this project. Also, I want to thank Jyrki Katajainen; the relatively short time I have been in touch with him, he has always been helpful and understanding.

## References

- [Hervé2003] Hervé Brönnimann, Experiments with the design and implementation of binary search trees, Unpublished manuscript including the code base
- [Google] <http://www.google.com>, Information searching about C++, AVL-trees etc.
- [SGISTL] Silicon Graphics, Inc., Standard Template Library Programmer's Guide, 1993–2004. Website accessible at <http://www.sgi.com/tech/stl/>
- [BS1997] Bjarne Stroustrup, The C++ Programming Language, 3rd edition, Addison-Wesley, 1997
- [MAW1998] Mark Allen Weiss, Data Structures and Problem Solving Using JAVA, 1st edition, Addison-Wesley, 1998

# A Appendix

## A.1 Test results

a-2

Structure	Type	Nr. of elem.	Range ins.	Forw.trans	Itera.search	Ran.suc.search	Ran.unsuc.search	Ran.del	Itera.search
AVL Set	int	10000	0.01s	0s	0s	0s	0.01s	0s	0s
STD Set	int	10000	0.01s	0s	0s	0s	0s	0.01s	0s
AVL Set	int	100000	0.22s	0.04s	0.21s	0.03s	0.02s	0.01s	0.05s
STD Set	int	100000	0.22s	0.04s	0.21s	0.03s	0.02s	0.02s	0.04s
AVL Set	int	500000	1.76s	0.19s	1.71s	0.18s	0.19s	0.08s	0.24s
STD Set	int	500000	1.77s	0.2s	1.66s	0.18s	0.17s	0.07s	0.24s
AVL Set	int	1000000	4.04s	0.39s	3.97s	0.43s	0.42s	0.17s	0.49s
STD Set	int	1000000	4.11s	0.4s	3.91s	0.42s	0.41s	0.14s	0.49s
AVL Set	int	2000000	9.57s	0.86s	9.62s	1.02s	1.01s	0.37s	1.06s
STD Set	int	2000000	9.76s	0.84s	9.42s	1s	0.99s	0.3s	1.07s
AVL Set	double	10000	0.01s	0s	0.01s	0s	0s	0s	0.01s
STD Set	double	10000	0.01s	0s	0.01s	0s	0s	0s	0s
AVL Set	double	100000	0.23s	0.03s	0.22s	0.03s	0.02s	0.01s	0.05s
STD Set	double	100000	0.24s	0.04s	0.22s	0.03s	0.02s	0.02s	0.04s
AVL Set	double	500000	1.83s	0.2s	1.74s	0.19s	0.18s	0.08s	0.26s
STD Set	double	500000	1.82s	0.2s	1.7s	0.18s	0.19s	0.06s	0.26s
AVL Set	double	1000000	4.16s	0.38s	3.99s	0.43s	0.42s	0.17s	0.54s
STD Set	double	1000000	4.2s	0.4s	3.94s	0.42s	0.42s	0.14s	0.54s
AVL Set	double	2000000	9.68s	0.84s	9.67s	1.02s	1.01s	0.35s	1.15s
STD Set	double	2000000	9.63s	0.84s	9.29s	0.98s	0.97s	0.29s	1.12s

## A.2 Code

### A.2.1 avl\_tree.hpp

```
/*
 * Copyright (c) 2003
 * Stephan Lyngge, Department of Computer Science University of Copenhagen
 *
 * Copyright (c) 2002, 2003
 * Herve Bronnimann, Polytechnic University
 *
 * Copyright (c) 1996,1997
 * Silicon Graphics Computer Systems, Inc.
 *
 * Copyright (c) 1994
 * Hewlett-Packard Company
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear in
 * supporting documentation. Neither the author nor Polytechnic
 * University, nor Silicon Graphics, nor Hewlett Packard, makes any
 * representations about the suitability of this software for any
 * purpose. It is provided "as is" without express or implied warranty.
 */

#ifndef BOOST_AVL_TREE_HPP
#define BOOST_AVL_TREE_HPP

/*
AVL search tree class, designed for use in implementing STL
associative containers (set, multiset, map, and multimap).
*/

#include <functional>
#include <algorithm>
#include <iterator>
#include <memory>

#include <boost/tree/iterator.hpp>
#include <boost/tree/set_helper.hpp>

namespace boost {

    namespace tree {

        /* AVL tree node classes
         * A color is needed to distinguish the header from the root
         * and is compacted into the parent pointer
        */
    }
}

```

```

*/

struct avl_tree_node_base
{
    typedef avl_tree_node_base* base_ptr;
    base_ptr m_parent;
    base_ptr m_left;
    base_ptr m_right;
    short m_balance;
};

template <class Value>
struct avl_tree_node : public avl_tree_node_base
{
    typedef avl_tree_node<Value>* link_type;
    Value m_data;
};

/* Avl tree base class
*/

template <class Key, class Value, class KeyOfValue, class Alloc>
class avl_tree_base
    : protected Alloc::rebind<avl_tree_node<Value> >::other
{
    // node types and accessors
protected:
    typedef avl_tree_node<Value>          node;
    typedef node*                          link_type;
protected:
    link_type& m_root()          const { return (link_type&) m_header->m_parent; }
    link_type& m_leftmost()     const { return (link_type&) m_header->m_left; }
    link_type& m_rightmost()    const { return (link_type&) m_header->m_right; }

    // public types
public:
    typedef typename Alloc::rebind<node>::other    allocator_type;
    typedef Key                                    key_type;
    typedef Value                                  value_type;
    typedef value_type*                            pointer;
    typedef const value_type*                      const_pointer;
    typedef value_type&                            reference;
    typedef const value_type&                      const_reference;
    typedef std::size_t                            size_type;
    typedef std::ptrdiff_t                        difference_type;

    typedef binary_tree_iterator<value_type, reference, pointer,
        node, parent_map<node>, left_map<node>, right_map<node> >
        iterator;
    typedef binary_tree_iterator<value_type, const_reference, const_pointer,
        node, parent_map<node>, left_map<node>, right_map<node> >
        const_iterator;
    typedef typename boost::reverse_iterator_generator<iterator>

```

```

        ::type                                reverse_iterator;
typedef typename boost::reverse_iterator_generator<const_iterator>
        ::type                                const_reverse_iterator;

        // structors
public:
    explicit avl_tree_base(const allocator_type& a = allocator_type())
        : allocator_type(a), m_node_count(0)
        { m_header = m_get_node(); m_init(); }
    ~avl_tree_base() { m_destroy(m_root()); m_put_node(m_header); }
protected:
    void m_init();

        // allocation
public:
    allocator_type get_allocator() const
        { return *static_cast<allocator_type*>(this); }
protected:
    node* m_get_node() { return allocate(1); }
    void m_put_node(link_type p) { deallocate(p, 1); }
    link_type m_create_node(const value_type& x);
    link_type m_clone_node(link_type p);
    void m_destroy_node(link_type p);
    void m_destroy(link_type x);
protected:
    void construct(Value* p, Value const& v) const { new((void*)p) Value(v); }
    void destroy(Value* p) const { p->~Value(); }

        // copy and assignment
public:
    avl_tree_base(const avl_tree_base& x);
    avl_tree_base& operator=(const avl_tree_base& x);
protected:
    link_type m_copy(link_type x, link_type p);

        // iterators
public:
    iterator      begin()      { return iterator(m_leftmost()); }
    const_iterator begin() const { return const_iterator(m_leftmost()); }
    iterator      end()        { return iterator(m_header); }
    const_iterator end() const { return const_iterator(m_header); }

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

        // size accessors
public:
    bool empty() const { return m_node_count == 0; }
    size_type size() const { return m_node_count; }
    size_type max_size() const { return size_type(-1); }

```

```

// modifying members

public:
    void clear() {
        if (m_node_count != 0) { m_destroy(m_root()); m_init(); }
    }
    void swap(avl_tree_base& t) {
        std::swap(m_header, t.m_header);
        std::swap(m_node_count, t.m_node_count);
    }

protected:
#ifdef BOOST_USE_SPLAY_TREE
    mutable
#endif
    node*      m_header;
    size_type m_node_count;
};

/* Avl tree class, at last
*/

template <class Key, class Value, class KeyOfValue, class Compare,
          class Alloc = std::allocator<Value> >
class avl_tree
    : protected Compare
    , public avl_tree_base<Key, Value, KeyOfValue, Alloc>
    , public set_helper< avl_tree<Key, Value, KeyOfValue, Compare, Alloc>,
                       avl_tree_base<Key, Value, KeyOfValue, Alloc>
    >
{

protected:
    typedef avl_tree_base<Key, Value, KeyOfValue, Alloc> base_type;
    typedef typename base_type::node                    node;
    typedef typename base_type::link_type               link_type;

// public types

public:
    typedef typename base_type::allocator_type          allocator_type;
    typedef typename base_type::key_type               key_type;
    typedef typename base_type::value_type             value_type;
    typedef typename base_type::pointer                pointer;
    typedef typename base_type::const_pointer          const_pointer;
    typedef typename base_type::reference              reference;
    typedef typename base_type::const_reference        const_reference;
    typedef typename base_type::size_type              size_type;
    typedef typename base_type::difference_type        difference_type;
    typedef typename base_type::iterator               iterator;
    typedef typename base_type::const_iterator        const_iterator;
    typedef typename base_type::const_reverse_iterator const_reverse_iterator;
    typedef typename base_type::reverse_iterator       reverse_iterator;

```

```

// structors
public:
    avl_tree(const Compare& comp = Compare(),
             const allocator_type& a = allocator_type())
        : Compare(comp), base_type(a) {}
    ~avl_tree() { clear(); }

// copy and assignment
public:
    avl_tree(const avl_tree& x) : Compare(x), base_type(x) {}
    avl_tree& operator=(const avl_tree& x) {
        Compare::operator=(x.key_compare());
        base_type::operator=(x);
        return *this;
    }

// comparison object
public:
    Compare key_compare() const { return m_compare(); }
protected:
    Compare& m_compare()
        { return static_cast<Compare&>(*this); }
    Compare const& m_compare() const
        { return static_cast<Compare const&>(*this); }

// search functions
public:
    iterator find(const key_type& x);
    const_iterator find(const key_type& x) const;
    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;

// insert functions
public:
    std::pair<iterator,bool> insert_unique(const value_type& x);
    iterator insert_equal(const value_type& x);
    iterator insert_unique(iterator position, const value_type& x);
    iterator insert_equal(iterator position, const value_type& x);
protected:
    iterator m_insert(link_type x, link_type y, const value_type& v);

// erase functions
public:
    void erase(iterator position);

// checking and statistics
public:
    bool is_valid() const;
    std::pair<size_type,size_type> search_length(const key_type& x) const;

```

```

                                                    // debugging
public:
    template <class OS> OS& debug_print(OS& os);
protected:
    template <class OS> void m_debug_print(OS& os, node* x, int depth);

};

} // namespace tree

} // namespace boost

#include <boost/tree/avl_tree.cpp>

#endif // BOOST_AVL_TREE_HPP
```

## A.2.2 avl\_tree.cpp

```
/*
 * Copyright (c) 2003
 * Stephan Lyngge, Department of Computer Science University of Copenhagen
 *
 * Copyright (c) 2002, 2003
 * Herve Bronnimann, Polytechnic University
 *
 * Copyright (c) 1996,1997
 * Silicon Graphics Computer Systems, Inc.
 *
 * Copyright (c) 1994
 * Hewlett-Packard Company
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear in
 * supporting documentation. Neither the author nor Polytechnic
 * University, nor Silicon Graphics, nor Hewlett Packard, makes any
 * representations about the suitability of this software for any
 * purpose. It is provided "as is" without express or implied warranty.
 */

#ifndef BOOST_AVL_TREE_CPP
#define BOOST_AVL_TREE_CPP

/*
Avl tree class, designed for use in implementing STL
associative containers (set, multiset, map, and multimap).
*/

#include <boost/ref.hpp> // to solve problem with template removing &
#include <boost/tree/algorithm.hpp>

namespace boost {

    namespace tree {

        // Use property-map-style access, rather than static member
        // functions (the property maps will be optimized away anyway).

        template <class T>
        inline void
        avl_tree_use_property_map_ignore_unused_variable_warning(const T&) { }

        #define AVL_TREE_USE_PROPERTY_MAPS( node ) \
            parent_map<node>                parent; \
            left_map<node>                   left; \
            right_map<node>                  right; \
    }
    }
}
```

```

key_map<node,Key,Value,KeyOfValue>          key; \
KeyOfValue                                 kov; \
typedef typename compact_pointer_map<node,parent_map<node> >::reference \
                                           parent_reference; \
avl_tree_use_property_map_ignore_unused_variable_warning(parent); \
avl_tree_use_property_map_ignore_unused_variable_warning(left); \
avl_tree_use_property_map_ignore_unused_variable_warning(right); \
avl_tree_use_property_map_ignore_unused_variable_warning(key); \
avl_tree_use_property_map_ignore_unused_variable_warning(kov);

// make an empty tree
template <class Key,class Value,class KeyOfValue,class Alloc>
void
avl_tree_base<Key,Value,KeyOfValue,Alloc>
  ::m_init()
{
    AVL_TREE_USE_PROPERTY_MAPS( node )
    // used to distinguish header from root, in iterator.operator--
    m_root() = 0;
    m_leftmost() = m_header;
    m_rightmost() = m_header;
    m_node_count = 0;
}

// node allocation and deallocation, with value construction

template <class Key,class Value,class KeyOfValue,class Alloc>
typename avl_tree_base<Key,Value,KeyOfValue,Alloc>::link_type
avl_tree_base<Key,Value,KeyOfValue,Alloc>
  ::m_create_node(const value_type& v)
{
    AVL_TREE_USE_PROPERTY_MAPS( node )
    link_type tmp = m_get_node();
    try {
        construct(&tmp->m_data, v);
    }
    catch (...) {
        m_put_node(tmp);
        // __throw_exception_again; // how do we do that?
    }
    return tmp;
}

template <class Key,class Value,class KeyOfValue,class Alloc>
typename avl_tree_base<Key, Value, KeyOfValue, Alloc>::link_type
avl_tree_base<Key,Value,KeyOfValue,Alloc>
  ::m_clone_node(link_type x)
{
    AVL_TREE_USE_PROPERTY_MAPS( node )
    link_type tmp = m_create_node(x->m_data);
    left[tmp] = 0;
    right[tmp] = 0;
}

```

```

    return tmp;
}

template <class Key,class Value,class KeyOfValue,class Alloc>
void
avl_tree_base<Key,Value,KeyOfValue,Alloc>
    :m_destroy_node(link_type p)
{
    AVL_TREE_USE_PROPERTY_MAPS( node )
    destroy(&p->m_data);
    m_put_node(p);
}

// this recursive procedure destroys the whole subtree of x

template <class Key,class Value,class KeyOfValue,class Alloc>
void
avl_tree_base<Key,Value,KeyOfValue,Alloc>
    :m_destroy(link_type x)
{
    AVL_TREE_USE_PROPERTY_MAPS( node )
    while (x != 0) {
        m_destroy(right[x]);
        link_type y = left[x];
        m_destroy_node(x);
        x = y;
    }
}

// copy and assignment member functions

template <class Key,class Value,class KeyOfValue,class Alloc>
typename avl_tree_base<Key,Value,KeyOfValue,Alloc>::link_type
avl_tree_base<Key,Value,KeyOfValue,Alloc>
    :m_copy(link_type x, link_type p)
{
    AVL_TREE_USE_PROPERTY_MAPS( node )
    // structural copy.  x and p must be non-null.
    link_type top = m_clone_node(x);
    parent[top] = p;

    try {
        if (right[x])
            right[top] = m_copy(right[x], top);
        p = top;
        x = left[x];

        while (x != 0) {
            link_type y = m_clone_node(x);
            left[p] = y;
            parent[y] = p;
            if (right[x])

```

```

        right[y] = m_copy(right[x], y);
        p = y;
        x = left[x];
    }
}
catch (...) {
    m_destroy(top);
}
return top;
}

```

```

template <class Key,class Value,class KeyOfValue,class Alloc>
avl_tree_base<Key,Value,KeyOfValue,Alloc>
    :avl_tree_base(const avl_tree_base& x)
    : allocator_type(x), m_node_count(0)
{
    AVL_TREE_USE_PROPERTY_MAPS( node )
    m_header = m_get_node();
    if (x.m_root() == 0)
        m_init();
    else {
        m_root() = m_copy(x.m_root(), m_header);
        m_leftmost() = binary_tree_minimum(m_root(), left);
        m_rightmost() = binary_tree_maximum(m_root(), right);
        m_node_count = x.m_node_count;
    }
}

```

```

template <class Key,class Value,class KeyOfValue,class Alloc>
avl_tree_base<Key,Value,KeyOfValue,Alloc>&
avl_tree_base<Key,Value,KeyOfValue,Alloc>
    :operator=(const avl_tree_base& x)
{
    AVL_TREE_USE_PROPERTY_MAPS( node )
    if (this != &x) {
        clear();
        if (x.m_root() == 0)
            m_init();
        else {
            m_root() = m_copy(x.m_root(), m_header);
            m_leftmost() = binary_tree_minimum(m_root(), left);
            m_rightmost() = binary_tree_maximum(m_root(), right);
            m_node_count = x.m_node_count;
        }
    }
    return *this;
}

```

// insertion of an element

```

template <class Key,class Value,class KeyOfValue,class Compare,class Alloc>
typename avl_tree_base<Key,Value,KeyOfValue,Alloc>::iterator
avl_tree<Key,Value,KeyOfValue,Compare,Alloc>

```

```

        :m_insert(link_type x, link_type y, const value_type& v)
    {
        AVL_TREE_USE_PROPERTY_MAPS( node )
        link_type z;

        if (y == m_header || x != 0 || m_compare()(kov(v), key[y])) {
            z = m_create_node(v);
            left[y] = z;                // also makes m_leftmost() = z
                                       // when y == m_header

            if (y == m_header) {
                m_root() = z;
                m_rightmost() = z;
            }
            else if (y == m_leftmost())
                m_leftmost() = z;    // maintain m_leftmost() pointing to min node
        }
        else {
            z = m_create_node(v);
            right[y] = z;
            if (y == m_rightmost())
                m_rightmost() = z;    // maintain m_rightmost() pointing to max node
        }
        parent[z] = y;
        left[z] = 0;
        right[z] = 0;
        z->m_balance = 0; // the new node has no children

        // After insertion of a node, we have to rebalance (unless it is the first node)
        if( m_root() != z ) {
            short linkSide = getSide( z, ref(parent[m_header]), parent, left, right );
            avl_tree_rebalance(parent[z], ref(parent[m_header]), parent, left, right, 1, linkSide);
        }

        // parent_reference root = parent[m_header];
        #ifdef BOOST_USE_SPLAY_TREE
        splay_tree_splay_node(z, ref(parent[m_header]), parent, left, right);
        #endif
        ++m_node_count;
        return iterator(z);
    }

template <class Key,class Value,class KeyOfValue,
          class Compare,class Alloc>
typename avl_tree_base<Key,Value,KeyOfValue,Alloc>::iterator
avl_tree<Key,Value,KeyOfValue,Compare,Alloc>
    ::insert_equal(const value_type& v)
    {
        AVL_TREE_USE_PROPERTY_MAPS( node )
        link_type y = m_header;
        link_type x = m_root();
        while (x != 0) {
            y = x;
            if (m_compare()(kov(v), key[x]))
                x = left[x];
        }
    }

```

```

        else
            x = right[x];
    }
    return m_insert(x, y, v);
}

```

```

template <class Key,class Value,class KeyOfValue,class Compare,class Alloc>
std::pair<typename avl_tree_base<Key,Value,KeyOfValue,Alloc>::iterator,
        bool>
avl_tree<Key,Value,KeyOfValue,Compare,Alloc>
    ::insert_unique(const value_type& v)
{
    AVL_TREE_USE_PROPERTY_MAPS( node )
    link_type y = m_header;
    link_type x = m_root();
    bool comp = true;
    while (x != 0) {
        y = x;
        comp = m_compare()(kov(v), key[x]);
        if (comp)
            x = left[x];
        else
            x = right[x];
    }
    iterator j = iterator(y);
    if (comp)
        if (j == begin())
            return std::pair<iterator,bool>(m_insert(x, y, v), true);
        else
            --j;
    if (m_compare()(key[j.m_node], kov(v)))
        return std::pair<iterator,bool>(m_insert(x, y, v), true);
    return std::pair<iterator,bool>(j, false);
}

```

```

template <class Key,class Value,class KeyOfValue,
        class Compare,class Alloc>
typename avl_tree_base<Key,Value,KeyOfValue,Alloc>::iterator
avl_tree<Key, Value, KeyOfValue, Compare, Alloc>
    ::insert_unique(iterator position, const value_type& v)
{
    AVL_TREE_USE_PROPERTY_MAPS( node )
    if (position.m_node == left[m_header]) { // begin()
        if (size() > 0 &&
            m_compare()(kov(v), key[position.m_node]))
            return m_insert(position.m_node, position.m_node, v);
        // first argument just needs to be non-null
        else
            return insert_unique(v).first;
    } else if (position.m_node == m_header) { // end()
        if (m_compare()(key[m_rightmost()], kov(v)))
            return m_insert(0, m_rightmost(), v);
    }
}

```

```

        else
            return insert_unique(v).first;
    } else {
        iterator before = position;
        --before;
        if (m_compare()(key[before.m_node], kov(v))
            && m_compare()(kov(v), key[position.m_node])) {
            if (s_right(before.m_node) == 0)
                return m_insert(0, before.m_node, v);
            else
                return m_insert(position.m_node, position.m_node, v);
            // first argument just needs to be non-null
        } else
            return insert_unique(v).first;
    }
}

template <class Key,class Value,class KeyOfValue,class Compare,class Alloc>
typename avl_tree_base<Key,Value,KeyOfValue,Alloc>::iterator
avl_tree<Key,Value,KeyOfValue,Compare,Alloc>
::insert_equal(iterator position, const value_type& v)
{
    AVL_TREE_USE_PROPERTY_MAPS( node )
    if (position.m_node == left[m_header]) { // begin()
        if (size() > 0 &&
            m_compare()(kov(v), key[position.m_node]))
            return m_insert(position.m_node, position.m_node, v);
        // first argument just needs to be non-null
        else
            return insert_equal(v);
    } else if (position.m_node == m_header) { // end()
        if (!m_compare()(kov(v), key[m_rightmost()]))
            return m_insert(0, m_rightmost(), v);
        else
            return insert_equal(v);
    } else {
        iterator before = position;
        --before;
        if (!m_compare()(kov(v), key[before.m_node])
            && !m_compare()(key[position.m_node], kov(v))) {
            if (s_right(before.m_node) == 0)
                return m_insert(0, before.m_node, v);
            else
                return m_insert(position.m_node, position.m_node, v);
            // first argument just needs to be non-null
        } else
            return insert_equal(v);
    }
}

// erasure of an element

template <class Key,class Value,class KeyOfValue,class Compare,class Alloc>

```

```

inline void
avl_tree<Key,Value,KeyOfValue,Compare,Alloc>
    ::erase(iterator position)
{
    AVL_TREE_USE_PROPERTY_MAPS( node )

    short linkSide = getSide( position.m_node, ref(parent[m_header]), parent, left, right );
    short linkSideNew = avl_tree_relink_for_erase(position.m_node,
                                                ref(parent[m_header]),
                                                left[m_header],
                                                right[m_header],
                                                parent, left, right);

    if( linkSideNew != 0 )
        linkSide = linkSideNew;
    link_type z = parent[(position.m_node)];
    m_destroy_node(position.m_node);
    --m_node_count;

    // After a delete we have to rebalance
    if( m_root() != z )
        avl_tree_rebalance(z, ref(parent[m_header]), parent, left, right, -1, linkSide);
}

// various ways to search a tree

template <class Key,class Value,class KeyOfValue,class Compare,class Alloc>
typename avl_tree_base<Key,Value,KeyOfValue,Alloc>::iterator
avl_tree<Key,Value,KeyOfValue,Compare,Alloc>
    ::find(const key_type& k)
{
    AVL_TREE_USE_PROPERTY_MAPS( node )
    link_type y = m_header; /* Last node which is not less than k. */
    link_type x = m_root(); /* Current node. */

    while (x != 0) {
        if (!m_compare()(key[x], k))
            y = x, x = left[x];
        else
            x = right[x];
    }

    iterator j = iterator(y);
#ifdef BOOST_USE_SPLAY_TREE
    splay_tree_splay_node(y, ref(parent[m_header]), parent, left, right);
#endif
    return (j == end() || m_compare()(k, key[j.m_node])) ? end() : j;
}

template <class Key,class Value,class KeyOfValue,class Compare,class Alloc>
typename avl_tree_base<Key,Value,KeyOfValue,Alloc>::const_iterator
avl_tree<Key,Value,KeyOfValue,Compare,Alloc>
    ::find(const key_type& k) const
{

```

```

AVL_TREE_USE_PROPERTY_MAPS( node )
link_type y = m_header; /* Last node which is not less than k. */
link_type x = m_root(); /* Current node. */

while (x != 0) {
    if (!m_compare()(key[x], k))
        y = x, x = left[x];
    else
        x = right[x];
}

const_iterator j = const_iterator(y);
#ifdef BOOST_USE_SPLAY_TREE
splay_tree_splay_node(y, ref(parent[m_header]), parent, left, right);
#endif
return (j == end() || m_compare()(k, key[j.m_node])) ? end() : j;
}

template <class Key,class Value,class KeyOfValue,class Compare,class Alloc>
typename avl_tree_base<Key,Value,KeyOfValue,Alloc>::iterator
avl_tree<Key,Value,KeyOfValue,Compare,Alloc>
::lower_bound(const key_type& k)
{
    AVL_TREE_USE_PROPERTY_MAPS( node )
    link_type y = m_header; /* Last node which is not less than k. */
    link_type x = m_root(); /* Current node. */

    while (x != 0)
        if (!m_compare()(key[x], k))
            y = x, x = left[x];
        else
            x = right[x];

    #ifdef BOOST_USE_SPLAY_TREE
    splay_tree_splay_node(y, ref(parent[m_header]), parent, left, right);
    #endif
    return iterator(y);
}

template <class Key,class Value,class KeyOfValue,class Compare,class Alloc>
typename avl_tree_base<Key,Value,KeyOfValue,Alloc>::const_iterator
avl_tree<Key,Value,KeyOfValue,Compare,Alloc>
::lower_bound(const key_type& k) const
{
    AVL_TREE_USE_PROPERTY_MAPS( node )
    link_type y = m_header; /* Last node which is not less than k. */
    link_type x = m_root(); /* Current node. */

    while (x != 0)
        if (!m_compare()(key[x], k))
            y = x, x = left[x];
        else
            x = right[x];
}

```

```

    #ifdef BOOST_USE_SPLAY_TREE
    splay_tree_splay_node(y, ref(parent[m_header]), parent, left, right);
    #endif
    return const_iterator(y);
}

template <class Key,class Value,class KeyOfValue,class Compare,class Alloc>
typename avl_tree_base<Key,Value,KeyOfValue,Alloc>::iterator
avl_tree<Key,Value,KeyOfValue,Compare,Alloc>
    ::upper_bound(const key_type& k)
{
    AVL_TREE_USE_PROPERTY_MAPS( node )
    link_type y = m_header; /* Last node which is greater than k. */
    link_type x = m_root(); /* Current node. */

    while (x != 0)
        if (m_compare()(k, key[x]))
            y = x, x = left[x];
        else
            x = right[x];

    #ifdef BOOST_USE_SPLAY_TREE
    splay_tree_splay_node(y, ref(parent[m_header]), parent, left, right);
    #endif
    return iterator(y);
}

template <class Key,class Value,class KeyOfValue,
          class Compare,class Alloc>
typename avl_tree_base<Key,Value,KeyOfValue,Alloc>::const_iterator
avl_tree<Key,Value,KeyOfValue,Compare,Alloc>
    ::upper_bound(const key_type& k) const
{
    AVL_TREE_USE_PROPERTY_MAPS( node )
    link_type y = m_header; /* Last node which is greater than k. */
    link_type x = m_root(); /* Current node. */

    while (x != 0)
        if (m_compare()(k, key[x]))
            y = x, x = left[x];
        else
            x = right[x];

    #ifdef BOOST_USE_SPLAY_TREE
    splay_tree_splay_node(y, ref(parent[m_header]), parent, left, right);
    #endif
    return const_iterator(y);
}

template <class Key,class Value,class KeyOfValue,class Compare,class Alloc>
inline
std::pair<
    typename avl_tree_base<Key,Value,KeyOfValue,Alloc>::iterator,
    typename avl_tree_base<Key,Value,KeyOfValue,Alloc>::iterator>

```

```

avl_tree<Key, Value, KeyOfValue, Compare, Alloc>
    ::equal_range(const key_type& k)
{
    return std::pair<iterator, iterator>(lower_bound(k), upper_bound(k));
}

template <class Key, class Value, class KeyOfValue, class Compare, class Alloc>
inline
std::pair<
    typename avl_tree_base<Key, Value, KeyOfValue, Alloc>::const_iterator,
    typename avl_tree_base<Key, Value, KeyOfValue, Alloc>::const_iterator>
avl_tree<Key, Value, KeyOfValue, Compare, Alloc>
    ::equal_range(const key_type& k) const
{
    return std::pair<const_iterator, const_iterator>(lower_bound(k),
                                                    upper_bound(k));
}

// verification of validity

template <class Key, class Value, class KeyOfValue, class Compare, class Alloc>
bool
avl_tree<Key, Value, KeyOfValue, Compare, Alloc>
    ::is_valid() const
{
    AVL_TREE_USE_PROPERTY_MAPS( node )
    if (m_node_count == 0 || begin() == end())
        return m_node_count == 0 && begin() == end() &&
            left[m_header] == m_header && right[m_header] == m_header;

    for (const_iterator it = begin(); it != end(); ++it) {
        link_type x = it.m_node;
        link_type L = left[x];
        link_type R = right[x];

        // Check to see if AVL properties holds
        int l = 0;
        if ( L != 0 )
            l = 1 + awl_tree_get_max_depth( L, left, right );
        int r = 0;
        if ( R != 0 )
            r = 1 + awl_tree_get_max_depth( R, left, right );
        if ( x->m_balance != r - l )
            return false;
        if ( r - l < -1 || r - l > 1 )
            return false;

        // this is not sufficient for ensuring validity
        // it's not even true for multisets!
        if ( L && m_compare()(key[x], key[L]))
            return false;
        if ( R && m_compare()(key[R], key[x]))
            return false;
    }
}

```

```

    }

    if (m_leftmost() != binary_tree_minimum(m_root(), left))
        return false;
    if (m_rightmost() != binary_tree_maximum(m_root(), right))
        return false;

    return true;
}

template <class Key,class Value,class KeyOfValue,class Compare,class Alloc>
std::pair<
    typename avl_tree_base<Key,Value,KeyOfValue,Alloc>::size_type,
    typename avl_tree_base<Key,Value,KeyOfValue,Alloc>::size_type >
avl_tree<Key,Value,KeyOfValue,Compare,Alloc>
::search_length(const key_type& k) const
{
    AVL_TREE_USE_PROPERTY_MAPS( node )
    link_type x = m_root(); /* Current node. */

    size_type search_length = 0;
    while (x != 0) {
        ++search_length;
        if (!m_compare()(key[x], k))
            x = left[x];
        else
            x = right[x];
    }
    // find needs one more comparison to check if present
    return std::make_pair(search_length,search_length+1);
}

template <class Key,class Value,class KeyOfValue,class Compare,class Alloc>
template <class OS>
void
avl_tree<Key,Value,KeyOfValue,Compare,Alloc>
::m_debug_print(OS& os, node* x, int depth)
{
    AVL_TREE_USE_PROPERTY_MAPS( node )
    if (x != 0) {
        os << "Node [depth=" << depth << "] : ";
        os << x->m_data;

        if( x->m_left )
            os << " - (left: " << left[x]->m_data << ")";
        else
            os << " - (left: nil)";

        if( x->m_right )
            os << " - (right: " << right[x]->m_data << ")";
        else
            os << " - (right: nil)";

        if( depth )

```

```

        os << " - (parent: " << parent[x]->m_data << ")";
    else
        os << " - (parent: nil)";

    os << " - (balance: " << x->m_balance << ")";
    os << "\n";

    m_debug_print(os, left[x], depth+1);
    m_debug_print(os, right[x], depth+1);
}
}

template <class Key,class Value,class KeyOfValue,class Compare,class Alloc>
template <class OS>
OS&
avl_tree<Key,Value,KeyOfValue,Compare,Alloc>
::debug_print(OS& os)
{
    AVL_TREE_USE_PROPERTY_MAPS( node )
    node* x = m_header;
    os << "Tree: \n";
    m_debug_print(os, parent[x], 0);
    os << "\n";
    return os;
}

#undef AVL_TREE_USE_PROPERTY_MAPS

} // namespace detail

} // namespace boost

#endif // BOOST_AVL_TREE_HPP

```

### A.2.3 algorithm.hpp

```
/*
 * Copyright (c) 2003
 * Stephan Lyngé, Department of Computer Science University of Copenhagen
 *
 * Copyright (c) 2002
 * Herve Bronnimann, Polytechnic University
 *
 * Copyright (c) 1996,1997
 * Silicon Graphics Computer Systems, Inc.
 *
 * Copyright (c) 1994
 * Hewlett-Packard Company
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear in
 * supporting documentation. Neither the author nor Polytechnic
 * University, nor Silicon Graphics, nor Hewlett Packard, makes any
 * representations about the suitability of this software for any
 * purpose. It is provided "as is" without express or implied warranty.
 */

#ifndef BOOST_BINARY_TREE_ALGORITHM_HPP
#define BOOST_BINARY_TREE_ALGORITHM_HPP

#include <vector>

#include <boost/ref.hpp>
#include <boost/tree/iterator.hpp>
#include <boost/tree/random_number.hpp>

namespace boost {

    namespace tree {

        // Binary tree: minimum / maximum of subtree of x

        template <class Node, class Left>
        inline Node*
        binary_tree_minimum(Node* x, Left left)
        {
            while (left[x] != 0)
                x = left[x];
            return x;
        }

        template <class Node, class Right>
        inline Node*
        binary_tree_maximum(Node* x, Right right)
    }
}

```

```

{
    while (right[x] != 0)
        x = right[x];
    return x;
}

template <class Node, class Left, class HasLeft>
inline Node*
threaded_binary_tree_minimum(Node* x, Left left, HasLeft has_left)
{
    while (has_left[x])
        x = left[x];
    return x;
}

template <class Node, class Right, class HasRight>
inline Node*
threaded_binary_tree_maximum(Node* x, Right right, HasRight has_right)
{
    while (has_right[x])
        x = right[x];
    return x;
}

// Binary tree: predecessor / successor
// The special cases make a cyclic ordering among the nodes and the header

template <class Node, class Parent, class Left, class Right>
inline Node*
binary_tree_successor(Node* x, Parent parent, Left left, Right right)
{
    if (right[x] != 0) {
        // x could be the header, in which case return leftmost
        if (parent[right[x]] != x)
            return left[x];
        // else x could still be the header, if the rightmost is the root
        // but in that case the code below still works
        x = right[x];
        while (left[x] != 0)
            x = left[x];
    }
    else {
        Node* y = parent[x];
        while (x == right[y]) {
            x = y;
            y = parent[y];
        }
        if (right[x] != y)
            x = y; // not if x is the header, y is the root and right[y]==0
    }
    return x;
}
}

```

```

template <class Node, class Parent, class Left, class Right>
inline Node*
binary_tree_predecessor(Node* x, Parent parent, Left left, Right right)
{
    return binary_tree_successor(x, parent, right, left);
}

template <class Node,
          class Left, class HasLeft, class Right, class HasRight>
inline Node*
threaded_binary_tree_successor(Node* x,
                               Left left, HasLeft has_left,
                               Right right, HasRight has_right)
{
    if (has_right[x]) {
        // x cannot be the header
        x = right[x];
        while (has_left[x])
            x = left[x];
    }
    else
        x = right[x]; // simply follow thread (also fro header)
    return x;
}

template <class Node,
          class Left, class HasLeft, class Right, class HasRight>
inline Node*
threaded_binary_tree_predecessor(Node* x,
                                 Left left, HasLeft has_left,
                                 Right right, HasRight has_right)
{
    return threaded_binary_tree_successor(x, right, has_right, left, has_left);
}

// Binary tree: rotate left / right

template <class Node, class NodePtrProxy, class Parent, class Left, class Right>
inline void
binary_tree_rotate_left(Node* x, reference_wrapper<NodePtrProxy> root,
                        Parent parent, Left left, Right right)
{
    Node* y = right[x];
    right[x] = left[y];
    if (left[y] != 0)
        parent[left[y]] = x;
    parent[y] = parent[x];

    if (x == root.get())
        root.get() = y;
    else if (x == left[parent[x]])
        left[parent[x]] = y;
}

```

```

    else
        right[parent[x]] = y;
    left[y] = x;
    parent[x] = y;
}

template <class Node, class NodePtrProxy, class Parent, class Left, class Right>
inline void
binary_tree_rotate_right(Node* x, reference_wrapper<NodePtrProxy> root,
                        Parent parent, Left left, Right right)
{
    binary_tree_rotate_left(x, root, parent, right, left);
}

// Binary tree: relink for erase, internal version (keeps x, y, x_parent)

template <class Node, class NodePtrProxy,
          class Parent, class Left, class Right>
inline void
binary_tree_relink_for_erase(Node* z,
    reference_wrapper<NodePtrProxy> root_proxy,
    Node*& leftmost, Node*& rightmost,
    Node*& x, Node*& y, Node*& x_parent,
    Parent parent, Left left, Right right)
{
    NodePtrProxy& root(root_proxy.get());
    y = z;
    if (left[y] == 0) // z has at most one non-null child. y == z.
        x = right[y]; // x might be null.
    else
        if (right[y] == 0) // z has exactly one non-null child. y == z.
            x = left[y]; // x is not null.
        else { // z has two non-null children. Set y to
            y = right[y]; // z's successor. x might be null.
            while (left[y] != 0)
                y = left[y];
            x = right[y];
        }
    if (y != z) { // relink y in place of z. y is z's successor
        parent[left[z]] = y;
        left[y] = left[z];
        if (y != right[z]) {
            x_parent = parent[y];
            if (x) parent[x] = parent[y];
            left[parent[y]] = x; // y must be a left child
            right[y] = right[z];
            parent[right[z]] = y;
        }
        else
            x_parent = y;
        if (root == z)
            root_proxy.get() = y;
        else if (left[parent[z]] == z)

```

```

        left[parent[z]] = y;
    else
        right[parent[z]] = y;
    parent[y] = parent[z];
}
else { // y == z
    x_parent = parent[y];
    if (x) parent[x] = parent[y];
    if (root == z)
        root_proxy.get() = x;
    else
        if (left[parent[z]] == z)
            left[parent[z]] = x;
        else
            right[parent[z]] = x;
    if (leftmost == z)
        if (right[z] == 0) // left[z] must be null also
            leftmost = parent[z];
    // makes leftmost == m_header if z == root
    else
        leftmost = binary_tree_minimum(x, left);
    if (rightmost == z)
        if (left[z] == 0) // right[z] must be null also
            rightmost = parent[z];
    // makes rightmost == m_header if z == root
    else // x == left[z]
        rightmost = binary_tree_maximum(x, right);
}
}

// version without x, y and x_parent
template <class Node, class NodePtrProxy,
         class Parent, class Left, class Right>
inline void
binary_tree_relink_for_erase(Node* z,
    reference_wrapper<NodePtrProxy> root_proxy,
    Node*& leftmost, Node*& rightmost,
    Parent parent, Left left, Right right)
{
    Node* x = 0;
    Node* y = 0;
    Node* x_parent = 0;
    binary_tree_relink_for_erase(z, root_proxy, leftmost, rightmost,
        x, y, x_parent, parent, left, right);
}

// Randomized tree: rebalance for insert

template <class RandomAccessIter, class Parent, class Left, class Right>
typename std::iterator_traits<RandomAccessIter>::value_type
randomized_tree_perfect_rebalance(
    RandomAccessIter first, RandomAccessIter last,
    Parent parent, Left left, Right right)

```

```

{
    if (first != last) {
        RandomAccessIter root = first + (last-first)/2;
        left[*root] = randomized_tree_perfect_rebalance(first, root,
                                                    parent, left, right);

        if (left[*root] != 0)
            parent[left[*root]] = *root;
        right[*root] = randomized_tree_perfect_rebalance(root+1, last,
                                                    parent, left, right);

        if (right[*root] != 0)
            parent[right[*root]] = *root;
        (*root)->set_left_size(root - first);
        return *root;
    }
    return 0;
}

template <class Node, class NodePtrProxy,
          class Parent, class Left, class Right>
inline void
randomized_tree_rebalance_for_insert(Node* z,
    reference_wrapper<NodePtrProxy> root_proxy,
    Parent parent, Left left, Right right)
{
    NodePtrProxy& root(root_proxy.get());
    Node* header = parent[(Node*)root];
    std::size_t n = 1;
    z = parent[z];
    n += z->m_size+1; // uses the fact that m_size has been flipped
    // during insertion, so that it contains the size of the other subtree
    while (z != header) {
        if (random_number(n) == 0) {
            Node* y = parent[z];
            // gather all the nodes in subtree by inorder traversal
            std::vector<Node*> subtree; subtree.reserve(n);
            Node* x = binary_tree_minimum(z, left);
            Node* t = binary_tree_maximum(z, right);
            while (x != t) {
                subtree.push_back(x);
                x = binary_tree_successor(x, parent, left, right);
            }
            subtree.push_back(t);
            // reconstruct tree
            t = randomized_tree_perfect_rebalance(subtree.begin(), subtree.end(),
                                                parent, left, right);

            // reattach tree to y
            parent[t] = y;
            if (left[y] == z)
                left[y] = t;
            else if (right[y] == z)
                right[y] = t;
            else { // z is the root and y is the header
                // parent[y] = (Node*)t;
                root = t;
            }
        }
    }
}

```

```

    }
    return;
  }
  z = parent[z]; n += z-> m_size+1;
}
}

```

// Red/black tree: rebalance for insert

```

template <class Node, class NodePtrProxy,
          class Parent, class Color, class Left, class Right>
inline void
rb_tree_rebalance_for_insert(Node* x,
  reference_wrapper<NodePtrProxy> root_proxy,
  Parent parent, Color color, Left left, Right right)
{
  NodePtrProxy& root(root_proxy.get());
  color[x] = s_rb_tree_red;
  while (x != root && color[parent[x]] == s_rb_tree_red) {
    if (parent[x] == left[parent[parent[x]]) {
      Node* y = right[parent[parent[x]]];
      if (y && color[y] == s_rb_tree_red) {
        color[parent[x]] = s_rb_tree_black;
        color[y] = s_rb_tree_black;
        color[parent[parent[x]]] = s_rb_tree_red;
        x = parent[parent[x]];
      }
      else {
        if (x == right[parent[x]]) {
          x = parent[x];
          binary_tree_rotate_left(x, root_proxy,
            parent, left, right);
        }
        color[parent[x]] = s_rb_tree_black;
        color[parent[parent[x]]] = s_rb_tree_red;
        binary_tree_rotate_right((Node*)(parent[parent[x]]),
          root_proxy,
          parent, left, right);
      }
    }
  }
  else {
    Node* y = left[parent[parent[x]]];
    if (y && color[y] == s_rb_tree_red) {
      color[parent[x]] = s_rb_tree_black;
      color[y] = s_rb_tree_black;
      color[parent[parent[x]]] = s_rb_tree_red;
      x = parent[parent[x]];
    }
    else {
      if (x == left[parent[x]]) {
        x = parent[x];
        binary_tree_rotate_right(x, root_proxy,
          parent, left, right);
      }
    }
  }
}

```

```

    }
    color[parent[x]] = s_rb_tree_black;
    color[parent[parent[x]]] = s_rb_tree_red;
    binary_tree_rotate_left((Node*)(parent[parent[x]]),
                            root_proxy,
                            parent, left, right);
    }
}
}
color[root] = s_rb_tree_black;
}

// Red/black tree: rebalance for erase

template <class Node, class NodePtrProxy,
          class Parent, class Color, class Left, class Right>
Node*
rb_tree_rebalance_for_erase(Node* z,
    reference_wrapper<NodePtrProxy> root_proxy,
    Node*& leftmost, Node*& rightmost,
    Parent parent, Color color, Left left, Right right)
{
    NodePtrProxy& root(root_proxy.get());
    Node* x = 0;
    Node* x_parent = 0;
    // remove node, as in binary tree, except that
    // there are two cases: y is either the successor of z if it has
    // two children (and has been relinked instead of z), or z itself
    // we need to remember the color of z and put it in y
    Node* y = z;
    binary_tree_relink_for_erase(z, root_proxy, leftmost, rightmost,
        x, y, x_parent, parent, left, right);

    color[y] = color[z];
    // this part just takes care of the coloring with rotations
    if (color[y] != s_rb_tree_red) {
        while (x != root && (x == 0 || color[x] == s_rb_tree_black))
            if (x == left[x_parent]) {
                Node* w = right[x_parent];
                if (color[w] == s_rb_tree_red) {
                    color[w] = s_rb_tree_black;
                    color[x_parent] = s_rb_tree_red;
                    binary_tree_rotate_left(x_parent, root_proxy,
                        parent, left, right);
                    w = right[x_parent];
                }
            }
            if ((left[w] == 0 ||
                color[left[w]] == s_rb_tree_black) &&
                (right[w] == 0 ||
                color[right[w]] == s_rb_tree_black)) {
                color[w] = s_rb_tree_red;
                x = x_parent;
                x_parent = parent[x_parent];
            } else {

```

```

    if (right[w] == 0 ||
        color[right[w]] == s_rb_tree_black) {
        if (left[w]) color[left[w]] = s_rb_tree_black;
        color[w] = s_rb_tree_red;
        binary_tree_rotate_right(w, root_proxy,
                                parent, left, right);
        w = right[x_parent];
    }
    color[w] = color[x_parent];
    color[x_parent] = s_rb_tree_black;
    if (right[w]) color[right[w]] = s_rb_tree_black;
    binary_tree_rotate_left(x_parent, root_proxy,
                            parent, left, right);
    break;
}
} else { // same as above, with m_right <-> m_left.
    Node* w = left[x_parent];
    if (color[w] == s_rb_tree_red) {
        color[w] = s_rb_tree_black;
        color[x_parent] = s_rb_tree_red;
        binary_tree_rotate_right(x_parent, root_proxy,
                                parent, left, right);
        w = left[x_parent];
    }
    if ((right[w] == 0 ||
        color[right[w]] == s_rb_tree_black) &&
        (left[w] == 0 ||
        color[left[w]] == s_rb_tree_black)) {
        color[w] = s_rb_tree_red;
        x = x_parent;
        x_parent = parent[x_parent];
    } else {
        if (left[w] == 0 ||
            color[left[w]] == s_rb_tree_black) {
            if (right[w]) color[right[w]] = s_rb_tree_black;
            color[w] = s_rb_tree_red;
            binary_tree_rotate_left(w, root_proxy,
                                    parent, left, right);
            w = left[x_parent];
        }
        color[w] = color[x_parent];
        color[x_parent] = s_rb_tree_black;
        if (left[w]) color[left[w]] = s_rb_tree_black;
        binary_tree_rotate_right(x_parent, root_proxy,
                                parent, left, right);
        break;
    }
}
    if (x) color[x] = s_rb_tree_black;
}
return y;
}

```

```

// Splay tree: splay node

template <class Node, class NodePtrProxy,
          class Parent, class Left, class Right>
inline void
splay_tree_splay_node(Node* x,
                      reference_wrapper<NodePtrProxy> root_proxy,
                      Parent parent, Left left, Right right)
{
    NodePtrProxy& root(root_proxy.get());
    if (x == parent[root]) return;
    while (x != root) {
        Node* y = parent[x];
        if (y == root)
            if (x == left[y])
                binary_tree_rotate_right(y, root_proxy, parent, left, right);
            else
                binary_tree_rotate_left(y, root_proxy, parent, left, right);
        else if (x == left[y]) {
            Node* z = parent[y];
            if (y == left[z]) {
                binary_tree_rotate_right(z, root_proxy, parent, left, right);
                binary_tree_rotate_right(y, root_proxy, parent, left, right);
            } else {
                binary_tree_rotate_right(y, root_proxy, parent, left, right);
                binary_tree_rotate_left(z, root_proxy, parent, left, right);
            }
        } else {
            Node* z = parent[y];
            if (y == right[z]) {
                binary_tree_rotate_left(z, root_proxy, parent, left, right);
                binary_tree_rotate_left(y, root_proxy, parent, left, right);
            } else {
                binary_tree_rotate_left(y, root_proxy, parent, left, right);
                binary_tree_rotate_right(z, root_proxy, parent, left, right);
            }
        }
    }
}

// Treap: restore heap property after insertion

template <class Node, class NodePtrProxy,
          class Parent, class Priority, class Left, class Right>
inline void
treap_heapify_after_insert(Node* x,
                           reference_wrapper<NodePtrProxy> root_proxy,
                           Parent parent, Priority priority, Left left, Right right)
{
    // NodePtrProxy& root(root_proxy.get());
    // note: priority[header] = 0 so the following loop terminates
    Node* y = parent[x];
    while (priority[x] < priority[y] ) {
        // std::cerr << "Rotating\n";

```

```

        if (x == left[y])
            binary_tree_rotate_right(y, root_proxy, parent, left, right);
        else
            binary_tree_rotate_left(y, root_proxy, parent, left, right);
        y = parent[x];
    }
}

// Bring avl_tree in balance after insert or delete
template <class Node, class NodePtrProxy,
          class Parent, class Left, class Right>
inline void
avl_tree_rebalance( Node* x, reference_wrapper<NodePtrProxy> root,
                   Parent parent, Left left, Right right, int cmd, short side )
{
    while( side != 0 ) {
        if( x->m_balance != cmd * side )
            x->m_balance = x->m_balance + cmd * side;
        else {
            side = x->m_balance;
            Node* y = left[x];
            if( x->m_balance == 1 )
                y = right[x];

            if( y->m_balance == -side ) {
                Node* z = right[y];
                if( side == 1 )
                    z = left[y];

                if( z->m_balance == -side ) {
                    x->m_balance = 0;
                    y->m_balance = side;
                    z->m_balance = 0;
                }
                else {
                    if( z->m_balance == side ) {
                        x->m_balance = -side;
                        y->m_balance = 0;
                        z->m_balance = 0;
                    }
                    else {
                        x->m_balance = 0;
                        y->m_balance = 0;
                    }
                }
            }
            if( side == -1 )
                binary_tree_rotate_left(y, root, parent, left, right );
            else
                binary_tree_rotate_right(y, root, parent, left, right );
        }
        else {
            if( y->m_balance == side ) {
                x->m_balance = 0;
            }
        }
    }
}

```

```

        y->m_balance = 0;
    }
    else {
        if( y->m_balance == 0 ) {
            x->m_balance = side;
            y->m_balance = -side;
        }
    }
}
if( side == -1 ) {
    y = left[x];
    binary_tree_rotate_right(x, root, parent, left, right );
}
else {
    y = right[x];
    binary_tree_rotate_left(x, root, parent, left, right );
}
x = y;
}

if( ( cmd == 1 && x->m_balance == 0 ) || ( cmd == -1 && x->m_balance != 0 ) )
    break;

side = getSide( x, root, parent, left, right );
x = parent[x];
}
}

```

```

// Avl tree: relink for erase, internal version (keeps x, y, x_parent)
// just like binary-tree-relink, with a few modifications
// main difference: returns the side of the relinked parent
template <class Node, class NodePtrProxy,
          class Parent, class Left, class Right>
inline short
avl_tree_relink_for_erase(Node* z,
    reference_wrapper<NodePtrProxy> root_proxy,
    Node*& leftmost, Node*& rightmost,
    Node*& x, Node*& y, Node*& x_parent,
    Parent parent, Left left, Right right)
{
    NodePtrProxy& root(root_proxy.get());
    short side = 0;
    y = z;
    if (left[y] == 0) // z has at most one non-null child. y == z.
        x = right[y]; // x might be null.
    else
        if (right[y] == 0) // z has exactly one non-null child. y == z.
            x = left[y]; // x is not null.
        else { // z has two non-null children. Set y to
            y = right[y]; // z's successor. x might be null.
            while (left[y] != 0)
                y = left[y];
            x = right[y];
        }
}

```

```

    }
Node* new_z_parent = parent[y];
if (y != z) {
    // relink y in place of z. y is z's successor
    side = getSide(y, root_proxy, parent, left, right); // SLY
    parent[left[z]] = y;
    left[y] = left[z];
    if (y != right[z]) {
        x_parent = parent[y];
        if (x) parent[x] = parent[y];
        left[parent[y]] = x; // y must be a left child
        right[y] = right[z];
        parent[right[z]] = y;
    }
    else
    {
        x_parent = y;
        new_z_parent = y;
    }
    if (root == z)
        root_proxy.get() = y;
    else if (left[parent[z]] == z)
        left[parent[z]] = y;
    else
        right[parent[z]] = y;
    parent[y] = parent[z];
    parent[z] = new_z_parent;
    y->m_balance = z->m_balance;
}
else {
    // y == z
    x_parent = parent[y];
    if (x) parent[x] = parent[y];
    if (root == z)
        root_proxy.get() = x;
    else
        if (left[parent[z]] == z)
            left[parent[z]] = x;
        else
            right[parent[z]] = x;
    if (leftmost == z)
        if (right[z] == 0) // left[z] must be null also
            leftmost = parent[z];
    // makes leftmost == m_header if z == root
    else
        leftmost = binary_tree_minimum(x, left);
    if (rightmost == z)
        if (left[z] == 0) // right[z] must be null also
            rightmost = parent[z];
    // makes rightmost == m_header if z == root
    else // x == left[z]
        rightmost = binary_tree_maximum(x, right);
}

return side;
}

```

```

// version without x, y and x_parent
// Just like the binary version
template <class Node, class NodePtrProxy,
         class Parent, class Left, class Right>
inline short
avl_tree_relink_for_erase(Node* z,
    reference_wrapper<NodePtrProxy> root_proxy,
    Node*& leftmost, Node*& rightmost,
    Parent parent, Left left, Right right)
{
    Node* x = 0;
    Node* y = 0;
    Node* x_parent = 0;
    return avl_tree_relink_for_erase(z, root_proxy, leftmost, rightmost,
                                     x, y, x_parent, parent, left, right);
}

// AVL - Is the node attached to the left or right side of the parent
template <class Node, class NodePtrProxy,
         class Parent, class Left, class Right>
inline short
getSide( Node* x, reference_wrapper<NodePtrProxy> root,
        Parent parent, Left left, Right right)
{
    Node* p = parent[x];
    if( root == x )
        return 0;

    if( left[p] == x )
        return -1;
    if( right[p] == x )
        return 1;

    return 0; // Should never come her
}

// Avl tree - Get the maximum depth from a node
template <class Node, class Left, class Right>
inline int
avl_tree_get_max_depth(Node* x, Left left, Right right)
{
    if( left[x] == 0 && right[x] == 0 )
        return 0;
    else {
        if( left[x] != 0 && right[x] == 0 )
            return 1 + (avl_tree_get_max_depth(left[x], left, right));
        if( left[x] == 0 && right[x] != 0 )
            return 1 + (avl_tree_get_max_depth(right[x], left, right));
    }
}

```

```
int l = awl_tree_get_max_depth(left[x], left, right);
int r = awl_tree_get_max_depth(right[x], left, right);

if( l < r )
    return r + 1;
else
    return l + 1;
}

} // namespace tree

} // namespace boost

#endif // BOOST_BINARY_TREE_ALGORITHM_HPP
```

## A.2.4 set\_adapter.hpp

```
/*
 * Copyright (c) 2003
 * Stephan Lyngge, Department of Computer Science University of Copenhagen
 *
 * Copyright (c) 2002, 2003
 * Herve Bronnimann, Polytechnic University
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear in
 * supporting documentation. Neither the author nor Polytechnic
 * University makes any representations about the suitability of this
 * software for any purpose. It is provided "as is" without express or
 * implied warranty.
 */

#ifndef BOOST_SET_ADAPTER_HPP
#define BOOST_SET_ADAPTER_HPP

#if defined( BOOST_USE_JUMLIST )
#include <boost/tree/jumplist.hpp>
#elif defined( BOOST_USE_RB_TREE ) || defined( BOOST_USE_COMPACT_RB_TREE )
#include <boost/tree/rb_tree.hpp>
#elif defined( BOOST_USE_RANDOMIZED_TREE )
#include <boost/tree/randomized_tree.hpp>
#elif defined( BOOST_USE_BINARY_TREE )
#include <boost/tree/binary_tree.hpp>
#elif defined( BOOST_USE_AVL_TREE )
#include <boost/tree/avl_tree.hpp>
#elif defined( BOOST_USE_SPLAY_TREE )
#include <boost/tree/splay_tree.hpp>
#elif defined( BOOST_USE_TREAP )
#include <boost/tree/treap.hpp>
#endif

namespace boost {

    namespace detail {

        template <class T>
        struct Identity {
            T operator()(T const& t) const { return t; }
            T& operator()(T& t) const { return t; }
        };

    } // namespace detail

    // the main class: a set adapter that ensures that all the members of
    // std::set are present. (Could be much shorter by using inheritance,
    // but in this way we make sure that other members are not inherited,
```

```

// so we offer a strict std::set interface...)

template <class Key, class Compare = std::less<Key>,
         class Alloc = std::allocator<int>, // the type is irrelevant
#if defined( BOOST_USE_JUMLIST )
         class Impl = tree::jumplist<Key,Key,detail::Identity<Key>,
                                     Compare,Alloc>
#elif defined( BOOST_USE_RB_TREE ) || defined( BOOST_USE_COMPACT_RB_TREE )
         class Impl = tree::rb_tree<Key,Key,detail::Identity<Key>,
                                     Compare,Alloc>
#elif defined( BOOST_USE_BINARY_TREE )
         class Impl = tree::binary_tree<Key,Key,detail::Identity<Key>,
                                     Compare,Alloc>
#elif defined( BOOST_USE_AVL_TREE )
         class Impl = tree::avl_tree<Key,Key,detail::Identity<Key>,
                                     Compare,Alloc>
#elif defined( BOOST_USE_RANDOMIZED_TREE )
         class Impl = tree::randomized_tree<Key,Key,detail::Identity<Key>,
                                     Compare,Alloc>
#elif defined( BOOST_USE_SPLAY_TREE )
         class Impl = tree::splay_tree<Key,Key,detail::Identity<Key>,
                                     Compare,Alloc>
#elif defined( BOOST_USE_TREAP )
         class Impl = tree::treap<Key,Key,detail::Identity<Key>,
                                     Compare,Alloc>
#endif
        >
class set {
public:
    // typedefs:

    typedef Key    key_type;
    typedef Key    value_type;
    typedef Compare key_compare;
    typedef Compare value_compare;
private:
    Impl m_t; // jumplist, red/black tree, etc. representing set
public:
    typedef typename Impl::pointer pointer;
    typedef typename Impl::const_pointer const_pointer;
    typedef typename Impl::reference reference;
    typedef typename Impl::const_reference const_reference;
    typedef typename Impl::iterator iterator;
    typedef typename Impl::const_iterator const_iterator;
#ifndef BOOST_JUMLIST_FORWARD
    typedef typename Impl::reverse_iterator reverse_iterator;
    typedef typename Impl::const_reverse_iterator const_reverse_iterator;
#endif
    typedef typename Impl::size_type size_type;
    typedef typename Impl::difference_type difference_type;
    typedef typename Impl::allocator_type allocator_type;

    // allocation/deallocation

```

```

explicit set(const Compare& comp = Compare(), const Alloc& a = Alloc())
    : m_t(comp, a) {}

#ifndef BOOST_NO_MEMBER_TEMPLATES
template <class InputIterator>
set(InputIterator first, InputIterator last)
    : m_t(Compare(), allocator_type())
    { m_t.insert_unique(first, last); }

template <class InputIterator>
set(InputIterator first, InputIterator last, const Compare& comp,
    const allocator_type& a = allocator_type())
    : m_t(comp, a)
    { m_t.insert_unique(first, last); }
#else
set(const value_type* first, const value_type* last)
    : m_t(Compare(), allocator_type())
    { m_t.insert_unique(first, last); }

set(const value_type* first,
    const value_type* last, const Compare& comp,
    const allocator_type& a = allocator_type())
    : m_t(comp, a)
    { m_t.insert_unique(first, last); }

set(const_iterator first, const_iterator last)
    : m_t(Compare(), allocator_type())
    { m_t.insert_unique(first, last); }

set(const_iterator first, const_iterator last, const Compare& comp,
    const allocator_type& a = allocator_type())
    : m_t(comp, a) { m_t.insert_unique(first, last); }
#endif /* __STL_MEMBER_TEMPLATES */

set(const set& x) : m_t(x.m_t) {}
set& operator=(const set& x)
{
    m_t = x.m_t;
    return *this;
}

// statistics and jumplist specific
bool is_valid() const { return m_t.is_valid(); }
void rebalance(bool perfect) { m_t.rebalance_jump_pointers(perfect); }
std::pair<size_type, size_type> search_length(const Key& k) const
    { return m_t.search_length(k); }
template <class OS> OS& debug_print(OS& os)
    { return m_t.debug_print(os); }

// accessors
key_compare key_comp() const { return m_t.comp(); }
value_compare value_comp() const { return m_t.comp(); }
allocator_type get_allocator() const { return m_t.get_allocator(); }

```

```

iterator begin() { return m_t.begin(); }
const_iterator begin() const { return m_t.begin(); }
iterator end() { return m_t.end(); }
const_iterator end() const { return m_t.end(); }

#ifndef BOOST_JUMLIST_FORWARD
reverse_iterator rbegin() { return m_t.rbegin(); }
const_reverse_iterator rbegin() const { return m_t.rbegin(); }
reverse_iterator rend() { return m_t.rend(); }
const_reverse_iterator rend() const { return m_t.rend(); }
#endif

bool empty() const { return m_t.empty(); }
size_type size() const { return m_t.size(); }
size_type max_size() const { return m_t.max_size(); }
void swap(set& x) { m_t.swap(x.m_t); }

// insert/erase
std::pair<iterator,bool> insert(const value_type& x) {
    return m_t.insert_unique(x);
}
iterator insert(iterator position, const value_type& x) {
    return m_t.insert_unique(position, x);
}
#ifndef BOOST_NO_MEMBER_TEMPLATES
template <class InputIterator>
void insert(InputIterator first, InputIterator last) {
    m_t.insert_unique(first, last);
}
#else // BOOST_NO_MEMBER_TEMPLATES
void insert(const_iterator first, const_iterator last) {
    m_t.insert_unique(first, last);
}
void insert(const value_type* first, const value_type* last) {
    m_t.insert_unique(first, last);
}
#endif // BOOST_NO_MEMBER_TEMPLATES
void erase(iterator position) { m_t.erase(position); }
size_type erase(const key_type& x) { return m_t.erase(x); }
void erase(iterator first, iterator last) { m_t.erase(first, last); }
void clear() { m_t.clear(); }

// search operations
iterator find(const key_type& x) { return m_t.find(x); }
const_iterator find(const key_type& x) const { return m_t.find(x); }
size_type count(const key_type& x) const { return m_t.count(x); }
iterator lower_bound(const key_type& x) { return m_t.lower_bound(x); }
const_iterator lower_bound(const key_type& x) const { return m_t.lower_bound(x); }
iterator upper_bound(const key_type& x) { return m_t.upper_bound(x); }
const_iterator upper_bound(const key_type& x) const { return m_t.upper_bound(x); }
std::pair<iterator,iterator> equal_range(const key_type& x)
    { return m_t.equal_range(x); }
std::pair<const_iterator,const_iterator> equal_range(const key_type& x) const
    { return m_t.equal_range(x); }

```

```

friend bool operator==(const set& x, const set& y) {
    if (x.size() != y.size())
        return false;
    return std::equal(x.begin(), x.end(), y.begin());
}
friend bool operator<(const set& x, const set& y) {
    return std::lexicographical_compare(x.begin(), x.end(),
                                         y.begin(), y.end(),
                                         x.key_compare());
}
};

template <class Key, class Compare, class Alloc>
inline void swap(set<Key,Compare,Alloc>& x,
                 set<Key,Compare,Alloc>& y) {
    x.swap(y);
}

} // namespace boost

#endif

```

## A.2.5 set\_test\_latex.cpp

```
// Stephan Lynge - 2004

// Writes out experiments in a latex-document
// This file is a reuse of set_test.cpp - and because of that
// the file includes some code, which isn't used.

// Basicly all cout operations has been replaced, with a output-file-stream
// and a basic paremeter for start and end has been added.

#include <cstring>
#include <vector>
#include <algorithm>
#include <iterator>
#include <functional>
#include <iostream>
#include <fstream>

#include <boost/timer.hpp>

#ifdef BOOST_USE_STD_SET
#include <set>
#else
#include <boost/set_adapter.hpp>
#endif

#ifdef BOOST_ALLOC
#include <boost/pool/pool_alloc.hpp>
#endif

#include "random_sample_n.hpp" // includes custom boost::random_sample_n()

using namespace std;

// Basic options
//=====

bool sorted = false;
bool reversed = false;
bool rebalance = false;
bool perfect = false;
bool docopy = true;
bool dodeletion = true;
bool verify = false;
int n = 65535;

// Measurements for set implementation
//=====

template <class SET, class IT, class OS>
```

```

class Set_test_suite {
    typedef typename SET::value_type T;
    IT m_first, m_last;
    SET m_set;
public:
    Set_test_suite(IT first, IT last)
        : m_first(first), m_last(last), m_set() {}

    void test(std::size_t random_search_size = 16384, OS& cout = cerr) {
        boost::timer t;
        typedef typename SET::iterator      iterator;

        // RANGE INSERTION
        //cout << /* range insertion...          */ ";
        t.restart();
        for (IT it = m_first; it != m_last; ++it)
            m_set.insert(*it);
        cout << t.elapsed() << "s\n" << " & ";

        // TRAVERSAL
        //cout << " /* forward traversal...      */ ";
        t.restart();
        unsigned int n = 0;
        typename SET::iterator first( m_set.begin() ), last( m_set.end() );
        for (typename SET::iterator i=first; i!=last; ++i, ++n) {}
        cout << t.elapsed() << "s\n" << " & ";
        if (n != m_set.size()) cerr << "Traversed only " << n << " elements!\n";
        assert( n == m_set.size() );

        // ITERATIVE SEARCH
        // cout << " /* iterative search...      */ ";
        t.restart();
        for (IT i=m_first; i!=m_last; ++i)
            assert( *m_set.find(*i) == *i );
        cout << t.elapsed() << "s\n" << " & ";

        // RANDOM SUCCESSFUL SEARCH
        typedef typename std::vector<T>::const_iterator IT2;
        std::vector<T> X; X.reserve(random_search_size);
        std::vector<iterator> Y; Y.reserve(random_search_size);
        boost::random_sample_n(m_first, m_last,
            std::back_inserter(X), random_search_size);
        // cout << " /* random successful search... */ ";
        t.restart();
        for (IT2 i=X.begin(); i!=X.end(); ++i) {
            Y.push_back(m_set.find(*i));
            assert( *Y.back() == *i );
        }
        cout << t.elapsed() << "s\n" << " & ";
    }
};

```

```

// RANDOM UNSUCCESSFUL SEARCH
// cout << " /* random unsuccessful search... */ ";
t.restart();
for (IT2 i=X.begin(); i!=X.end(); ++i)
    assert( *m_set.lower_bound(*i-1) == *i );
cout << t.elapsed() << "s\n" << " & ";

// DELETION
if (dodeletion) {
    //cout << " /* random delete... */ ";
    t.restart();
    while (!Y.empty()) {
        m_set.erase(Y.back());
        Y.pop_back();
        //m_set.debug_print(cerr);
        //assert( m_set.is_valid() );
    }
    cout << t.elapsed() << "s\n" << " & ";

    //cout << " /* iterative search... */ ";
    t.restart();
    for (iterator i=m_set.begin(); i!=m_set.end(); ++i)
        assert( *m_set.find(*i) == *i );
    cout << t.elapsed() << "s\n" << "\n";
}

}
};

// Dispatch
//=====

template <class T, class OS>
void test_set(char *Tname, int size, OS& cout) {
    typedef std::vector<T> container;
    typedef typename container::const_iterator iterator;

#ifdef BOOST_USE_POOL_ALLOC
    typedef std::allocator<T> alloc;
#else
    typedef typename boost::fast_pool_allocator<T,
        boost::default_user_allocator_new_delete,
        boost::details::pool::null_mutex,128> alloc;
#endif

#ifdef BOOST_USE_STD_SET
    #ifdef MULTI
        typedef std::multiset<T, std::less<T>, alloc> tree_container;
    #else
        typedef std::set<T, std::less<T>, alloc> tree_container;
    #endif
#endif

```

```

#else
    typedef boost::set<T, std::less<T>, alloc> tree_container;
#endif

    container x;
    for (int i=0; i<size; ++i)
        x.push_back(static_cast<T>(2*i));
    char* str;
    if (sorted)
        str = "sorted";
    else if (reversed) {
        std::reverse(x.begin(), x.end());
        str = "reversed";
    } else {
        std::random_shuffle(x.begin(), x.end());
        str = "random";
    }

    cout << Tname << " & " << "\n";
    cout << size << " & " << "\n";

    Set_test_suite<tree_container, iterator, ostream> m1(x.begin(), x.end() );
    m1.test(size/10, cout);

    cout << "\\\\" << "\n";
}

// Main function
//=====

int main(int argc, char **argv)
{
    bool begin_table = false;
    bool end_table = false;
    bool begin_doc = false;
    bool end_doc = false;
    char* pText = "";

    if(!strcmp(argv[0], "./avl_tree_set_test_latex_int" )
        || !strcmp(argv[0], "./avl_tree_set_test_latex_double" ))
        pText = "AVL Set";
    if(!strcmp(argv[0], "./rb_std_set_test_latex_int" )
        || !strcmp(argv[0], "./rb_std_set_test_latex_double" ))
        pText = "STD Set";

    while (argc>1) {
        if (!strcmp(argv[1], "--beginTable"))
            begin_table = true;
        else if (!strcmp(argv[1], "--endTable"))
            end_table = true;
        else if (!strcmp(argv[1], "--beginDoc"))
            begin_doc = true;
        else if (!strcmp(argv[1], "--endDoc"))

```



```
if( end_doc ) {  
    cout << "\\end{landscape}" << "\n";  
    cout << "\\end{document}";  
}  
  
fb.close();  
return 0;  
}
```

## A.2.6 Makefile

```
CXX=g++
#CXX = g++3
#CXX = g++-3.0
#CXX = g++-3
DEBUG = -O3
CXXFLAGS += -W -Wall $(DEBUG) -DBOOST_ALLOC
BOOST = /home/stephan/Project/boost-1.30.2/
INCLUDE = -I../.. -I$(BOOST)

TESTFLAGS = -c -d

RB_SIZE      = 100000
JL_SIZE      = 100000
SPLAY_SIZE   = 100000
TREAP_SIZE   = 100000
BST_SIZE     = 100000
AVL_SIZE     = 100000

RB_SORTED_SIZE   = 100000
JL_SORTED_SIZE   = 100000
SPLAY_SORTED_SIZE = 100000
TREAP_SORTED_SIZE = 100000
BST_SORTED_SIZE   = 100000
AVL_SORTED_SIZE   = 100000

latex: clean \
      latex_test

latex_test: avl_tree_set_test_latex_int \
            rb_std_set_test_latex_int \
            avl_tree_set_test_latex_double \
            rb_std_set_test_latex_double
echo "AVL: 10000 elements (int)"
./avl_tree_set_test_latex_int --beginTable 10000
echo "STD: 10000 elements (int)"
./rb_std_set_test_latex_int 10000
echo "AVL: 100000 elements (int)"
./avl_tree_set_test_latex_int 100000
echo "STD: 100000 elements (int)"
./rb_std_set_test_latex_int 100000
echo "AVL: 500000 elements (int)"
./avl_tree_set_test_latex_int 500000
echo "STD: 500000 elements (int)"
./rb_std_set_test_latex_int 500000
echo "AVL: 1000000 elements (int)"
./avl_tree_set_test_latex_int 1000000
echo "STD: 1000000 elements (int)"
./rb_std_set_test_latex_int 1000000
echo "AVL: 2000000 elements (int)"
./avl_tree_set_test_latex_int 2000000
```

```

echo "STD: 2000000 elements (int)"
./rb_std_set_test_latex_int 2000000
echo "AVL: 10000 elements (double)"
./avl_tree_set_test_latex_double 10000
echo "STD: 10000 elements (double)"
./rb_std_set_test_latex_double 10000
echo "AVL: 100000 elements (double)"
./avl_tree_set_test_latex_double 100000
echo "STD: 100000 elements (double)"
./rb_std_set_test_latex_double 100000
echo "AVL: 500000 elements (double)"
./avl_tree_set_test_latex_double 500000
echo "STD: 500000 elements (double)"
./rb_std_set_test_latex_double 500000
echo "AVL: 1000000 elements (double)"
./avl_tree_set_test_latex_double 1000000
echo "STD: 1000000 elements (double)"
./rb_std_set_test_latex_double 1000000
echo "AVL: 2000000 elements (double)"
./avl_tree_set_test_latex_double 2000000
echo "STD: 2000000 elements (double)"
./rb_std_set_test_latex_double --endTable 2000000

jumplists: jl_rand_set_test jl_fw_rand_set_test  jl_cp_rand_set_test jl_fw_cp_rand_set_test

all: jumplists \
    rb_cp_tree_set_test rb_tree_set_test rb_std_set_test \
    bin_tree_set_test avl_tree_set_test splay_tree_set_test rand_tree_set_test \
    treap_cp_set_test treap_set_test

test: all test_jl test_trees

test_jl: jumplists test_jl_rand_set_test test_jl_fw_rand_set_test \
    test_jl_cp_rand_set_test  test_jl_fw_cp_rand_set_test

test_jl_rand_multiset_test: jl_rand_multiset_test
echo "TESTING RANDOMIZED BIDIRECTIONAL (MULTI)JUMPLIST"
./jl_rand_multiset_test $(TESTFLAGS) --sorted $(JL_SORTED_SIZE)
./jl_rand_multiset_test $(TESTFLAGS) --reversed $(JL_SORTED_SIZE)
./jl_rand_multiset_test $(TESTFLAGS) $(JL_SIZE)

test_jl_rand_set_test: jl_rand_set_test
echo "TESTING RANDOMIZED BIDIRECTIONAL JUMPLIST"
./jl_rand_set_test $(TESTFLAGS) --sorted $(JL_SORTED_SIZE)
./jl_rand_set_test $(TESTFLAGS) --reversed $(JL_SORTED_SIZE)
./jl_rand_set_test $(TESTFLAGS) $(JL_SIZE)

test_jl_fw_rand_set_test: ./jl_fw_rand_set_test
echo "TESTING RANDOMIZED FORWARD JUMPLIST"
./jl_fw_rand_set_test $(TESTFLAGS) --sorted $(JL_SORTED_SIZE)
./jl_fw_rand_set_test $(TESTFLAGS) --reversed $(JL_SORTED_SIZE)
./jl_fw_rand_set_test $(TESTFLAGS) $(JL_SIZE)

test_jl_cp_rand_set_test: ./jl_cp_rand_set_test

```

```

        echo "TESTING RANDOMIZED BIDIRECTIONAL JUMPLIST WITH COMPACT NODES"
        ./jl_cp_rand_set_test $(TESTFLAGS) --sorted $(JL_SORTED_SIZE)
        ./jl_cp_rand_set_test $(TESTFLAGS) --reversed $(JL_SORTED_SIZE)
        ./jl_cp_rand_set_test $(TESTFLAGS) $(JL_SIZE)

test_jl_fw_cp_rand_set_test: ./jl_fw_cp_rand_set_test
        echo "TESTING RANDOMIZED FORWARD JUMPLIST WITH COMPACT NODES"
        ./jl_fw_cp_rand_set_test $(TESTFLAGS) --sorted $(JL_SORTED_SIZE)
        ./jl_fw_cp_rand_set_test $(TESTFLAGS) --reversed $(JL_SORTED_SIZE)
        ./jl_fw_cp_rand_set_test $(TESTFLAGS) $(JL_SIZE)

test_trees: test_rand_tree_set_test test_rb_std_set_test \
        test_bin_tree_set_test test_splay_tree_set \
        test_treap_set test_treap_cp_set

test_bin_tree_set_test: ./bin_tree_set_test
        echo "TESTING BINARY TREE (ADAPTATION OF SGI STL IMPLEMENTATION)"
        ./bin_tree_set_test $(TESTFLAGS) -c --sorted $(BST_SORTED_SIZE)
        ./bin_tree_set_test $(TESTFLAGS) -c --reversed $(BST_SORTED_SIZE)
        ./bin_tree_set_test $(TESTFLAGS) $(BST_SIZE)

test_avl_tree_set_test: ./avl_tree_set_test
        echo "TESTING AVL TREE (ADAPTATION OF SGI STL IMPLEMENTATION)"
        ./avl_tree_set_test $(TESTFLAGS) -c --sorted $(AVL_SORTED_SIZE)
        ./avl_tree_set_test $(TESTFLAGS) -c --reversed $(AVL_SORTED_SIZE)
        ./avl_tree_set_test $(TESTFLAGS) $(AVL_SIZE)

test_splay_tree_set_test: ./splay_tree_set_test
        echo "TESTING SPLAY TREE (ADAPTATION OF SGI STL IMPLEMENTATION)"
        ./splay_tree_set_test $(TESTFLAGS) --sorted $(SPLAY_SORTED_SIZE)
        ./splay_tree_set_test $(TESTFLAGS) --reversed $(SPLAY_SORTED_SIZE)
        ./splay_tree_set_test $(TESTFLAGS) $(SPLAY_SIZE)

test_treap_set_test: ./treap_set_test
        echo "TESTING SPLAY TREE (ADAPTATION OF SGI STL IMPLEMENTATION)"
        ./treap_set_test $(TESTFLAGS) --sorted $(TREAP_SORTED_SIZE)
        ./treap_set_test $(TESTFLAGS) --reversed $(TREAP_SORTED_SIZE)
        ./treap_set_test $(TESTFLAGS) $(TREAP_SIZE)

test_treap_cp_set_test: ./treap_cp_set_test
        echo "TESTING SPLAY TREE (ADAPTATION OF SGI STL IMPLEMENTATION)"
        ./treap_cp_set_test $(TESTFLAGS) --sorted $(TREAP_SORTED_SIZE)
        ./treap_cp_set_test $(TESTFLAGS) --reversed $(TREAP_SORTED_SIZE)
        ./treap_cp_set_test $(TESTFLAGS) $(TREAP_SIZE)

test_rand_tree_set_test: ./rand_tree_set_test
        echo "TESTING RANDOMIZED TREE (ADAPTATION OF SGI STL IMPLEMENTATION)"
        ./rand_tree_set_test $(TESTFLAGS) --sorted $(RB_SORTED_SIZE)
        ./rand_tree_set_test $(TESTFLAGS) --reversed $(RB_SORTED_SIZE)
        ./rand_tree_set_test $(TESTFLAGS) $(RB_SIZE)

test_rb_tree_set_test: ./rb_tree_set_test
        echo "TESTING RED/BLACK TREE (ADAPTATION OF SGI STL IMPLEMENTATION)"
        ./rb_tree_set_test $(TESTFLAGS) --sorted $(RB_SORTED_SIZE)

```

```

./rb_tree_set_test $(TESTFLAGS) --reversed $(RB_SORTED_SIZE)
./rb_tree_set_test $(TESTFLAGS) $(RB_SIZE)

test_rb_cp_tree_set_test: ./rb_tree_set_test
    echo "TESTING COMPACT RED/BLACK TREE (ADAPTATION OF SGI STL IMPLEMENTATION)"
    ./rb_cp_tree_set_test $(TESTFLAGS) --sorted $(RB_SORTED_SIZE)
    ./rb_cp_tree_set_test $(TESTFLAGS) --reversed $(RB_SORTED_SIZE)
    ./rb_cp_tree_set_test $(TESTFLAGS) $(RB_SIZE)

test_rb_std_set_test:
    echo "TESTING STANDARD C++ SET"
    ./rb_std_set_test $(TESTFLAGS) --sorted $(RB_SORTED_SIZE)
    ./rb_std_set_test $(TESTFLAGS) --reversed $(RB_SORTED_SIZE)
    ./rb_std_set_test $(TESTFLAGS) $(RB_SIZE)

jl_fw_cp_rand_set_test: set_test.cpp
    $(CXX) $(CXXFLAGS) $(INCLUDE) -o $@ $< \
        -DBOOST_USE_JUMPLIST -DBOOST_JUMPLIST_FORWARD -DBOOST_COMPACT_JUMPLIST_NODE

jl_cp_rand_set_test: set_test.cpp
    $(CXX) $(CXXFLAGS) $(INCLUDE) -o $@ $< \
        -DBOOST_USE_JUMPLIST -DBOOST_COMPACT_JUMPLIST_NODE

jl_fw_rand_set_test: set_test.cpp
    $(CXX) $(CXXFLAGS) $(INCLUDE) -o $@ $< \
        -DBOOST_USE_JUMPLIST -DBOOST_JUMPLIST_FORWARD

jl_rand_set_test: set_test.cpp
    $(CXX) $(CXXFLAGS) $(INCLUDE) -o $@ $< \
        -DBOOST_USE_JUMPLIST

jl_rand_multiset_test: set_test.cpp
    $(CXX) $(CXXFLAGS) $(INCLUDE) -o $@ $< \
        -DBOOST_USE_JUMPLIST -DMULTI

bin_tree_set_test: set_test.cpp
    $(CXX) $(CXXFLAGS) $(INCLUDE) -o $@ $< -DBOOST_USE_BINARY_TREE

avl_tree_set_test: set_test.cpp
    $(CXX) $(CXXFLAGS) $(INCLUDE) -o $@ $< -DBOOST_USE_AVL_TREE

splay_tree_set_test: set_test.cpp
    $(CXX) $(CXXFLAGS) $(INCLUDE) -o $@ $< -DBOOST_USE_BINARY_TREE -DBOOST_USE_SPLAY_TREE

treap_set_test: set_test.cpp
    $(CXX) $(CXXFLAGS) $(INCLUDE) -o $@ $< -DBOOST_USE_TREAP

treap_cp_set_test: set_test.cpp
    $(CXX) $(CXXFLAGS) $(INCLUDE) -o $@ $< -DBOOST_USE_TREAP -DBOOST_USE_COMPACT_TREAP

rand_tree_set_test: set_test.cpp
    $(CXX) $(CXXFLAGS) $(INCLUDE) -o $@ $< -DBOOST_USE_RANDOMIZED_TREE

```

```

rb_cp_tree_set_test: set_test.cpp
    $(CXX) $(CXXFLAGS) $(INCLUDE) -o $@ $< -DBOOST_USE_COMPACT_RB_TREE

rb_tree_set_test: set_test.cpp
    $(CXX) $(CXXFLAGS) $(INCLUDE) -o $@ $< -DBOOST_USE_RB_TREE

rb_std_set_test: set_test.cpp
    $(CXX) $(CXXFLAGS) $(INCLUDE) -o $@ $< -DBOOST_USE_STD_SET

rb_std_multiset_test: set_test.cpp
    $(CXX) $(CXXFLAGS) $(INCLUDE) -o $@ $< -DBOOST_USE_STD_SET -DMULTI

avl_tree_set_test_latex_int: set_test_latex.cpp
    $(CXX) $(CXXFLAGS) $(INCLUDE) -o $@ $< -DBOOST_USE_AVL_TREE

rb_std_set_test_latex_int: set_test_latex.cpp
    $(CXX) $(CXXFLAGS) $(INCLUDE) -o $@ $< -DBOOST_USE_STD_SET

avl_tree_set_test_latex_double: set_test_latex.cpp
    $(CXX) $(CXXFLAGS) $(INCLUDE) -o $@ $< -DBOOST_USE_AVL_TREE -DBOOST_DONT_USE_INT -DBOOST_USE_D

rb_std_set_test_latex_double: set_test_latex.cpp
    $(CXX) $(CXXFLAGS) $(INCLUDE) -o $@ $< -DBOOST_USE_STD_SET -DBOOST_DONT_USE_INT -DBOOST_USE_D

clean:
    - rm -rf *.o jl*_set_test
    - rm -rf *.o rb*_set_test*
    - rm -rf *.o bin*_set_test
    - rm -rf *.o avl*_set_test*
    - rm -rf *.o rand*_set_test
    - rm -rf *.o splay*_set_test
    - rm -rf *.o treap*_set_test
    - rm -rf *.bbg *.bb *.da *.gcov
    - rm -rf *.prof
    - rm -rf a.out gmon.out cachegrind.out
    - rm -rf tree_test_latex.*

```