

Project proposal:

Associative containers with strong guarantees

Jyrki Katajainen

*Department of Computing, University of Copenhagen
Universitetsparken 1, 2100 Copenhagen East, Denmark*

Abstract. The Standard Template Library (STL) is a collection of generic algorithms and data structures that is part of the standard run-time environment of the C++ programming language. The STL provides four kinds of associative element containers: `set`, `multiset`, `map`, and `multimap`. In this project the goal is to develop an associative container that is safer, more reliable, more usable, and/or more efficient (with respect to time and space) than any of the existing realizations.

Keywords. Data structures, associative containers, search trees

Problem formulation

Let \mathcal{U} be a universe of elements which can be compared. Elements of \mathcal{U} can be singletons or pairs, each consisting of a key and some satellite information associated with that key. A data structure D that stores a collection of elements drawn from \mathcal{U} and supports efficient insert, delete, and search operations is called an *ordered dictionary*. The collection stored may or may not contain duplicates. In an abstract setting, D maintains a changing collection of elements in sorted order. Ordered dictionaries are a central piece of any programmer's toolbox, and are provided as a facility in a number of high-level programming languages.

In C++ terminology [5, Clause 23], an ordered dictionary is called an *associative container*. Each realization of the C++ standard library should provide four kinds of associative containers: `set` (elements singletons, no duplicates), `map` (elements pairs, no duplicates, ordering based on keys), `multiset` (elements singletons, duplicates allowed), and `multimap` (elements pairs, duplicates allowed, ordering based on keys). In the design of the C++ standard library, the main reason to support these four different dictionary variants is to allow a direct modification of satellite data, instead of forcing users to invoke a delete followed by an insert to make such an update. This feature is frequently employed in practical applications.

There are many data structures that could be used in the realization of associative containers; almost any elementary textbook on data structures offers one. Most relevant alternatives are based on balanced search trees; hash tables cannot be used because of the element ordering. Most publicly

available realizations (see, for instance, HP STL [23, Chapters 13–14], SGI STL [25], CPH STL [10]) are based on red-black trees [12] (see also [9, Chapter 13]). Often, as done in the HP STL, a unique interface is defined for red-black trees and this is then used for the realization of all four associative containers with appropriate bridge classes (cf. [27, Section 14.4]).

In this project, the goal is to develop competitors for red-black trees, and to compare the efficiency of these competitors to that of red-black trees. Any of the search trees presented in the algorithmic literature (see, for example, [9, Chapter 13] and the references mentioned therein) can be used as the basis of the implementation work. However, you should focus on data structures that provide good worst-case, not amortized or average-case, complexity bounds (see Figure 1). Especially noteworthy alternatives to red-black trees are the following data structures, but other data structures may be considered as well:

Data structure	Reference
AA trees	[3] (see also [28, Section 18.6])
(a, b) trees	[14, 18] (see also [20, Section III.5.2])
AVL trees	[1] (see also [29, Section 4.5])
BB[α] trees	[22] (see also [8] and [20, Section III.5.1])
deterministic skip lists	[21]
exponential trees	[4]
k -neighbour trees	[19]
logarithmic binary search trees	[24]
succinct red-black trees	[7]

Be aware that some of these data structures are rather complicated. Now you have been warned!

A complete declaration of the class `set` is given in Appendix A. In the CPH STL, each container is a bridge class that just calls the functions available in the given realization. This way a user has the flexibility to choose a realization that suits best for his or her purposes. The meaning of the member types and member functions declared should be self-explanatory. Further details can be found from the C++ standard [5, Clause 23], or any other source on the STL.

Let D be the data structure to be developed. The challenge to be taken up in this project is to meet the requirements specified in the C++ standard:

- R_1 : Allow insertion and deletion of an element (based on its value or position), successful and unsuccessful search (returning a position to the element or to the one-past-the-end element if unsuccessful) at logarithmic cost. Here the *cost* means the sum of element comparisons, element constructions, element destructions, and other operations on C++ built-in types, including memory allocation and deallocation.
- R_2 : Allow forward and backward iteration over the elements stored in D , starting at any position. A position is referred to by an *iterator*, which

Operation	Effect	Cost
$++p$	returns an iterator to the successor	amortized $O(1)$
$--p$	returns an iterator to the predecessor	amortized $O(1)$
$D.begin()$	returns an iterator to the minimum element	$O(1)$
$D.end()$	returns an iterator to the one-pass-the-end element; $--D.end()$ refers to the maximum element	$O(1)$
$D.insert(e)$	inserts an element	$O(\lg n)$
$D.insert(p,e)$	inserts an element with a hint	$O(\lg n)$, but amortized $O(1)$ if e is inserted just after p
$D.insert(S)$	inserts a collection of elements	$O S \lg(n + S)$
$D.erase(e)$	deletes all elements equal to e ; let S denote the removed collection	$O(\lg n + S)$
$D.erase(p)$	deletes the element at the given position	amortized $O(1)$
$D.erase(p,q)$	deletes all elements between the two iterators; let S denote the removed collection	$O(\lg n + S)$
$D.lower_bound(e)$	returns an iterator to the first element not less than e , if such exists	$O(\lg n)$
$D.upper_bound(e)$	returns an iterator to the first element greater than e , if such exists	$O(\lg n)$

Figure 1. Most important associative-container operations and their cost requirements. Here D is the container under consideration, p and q are iterators, e is an element, S is a collection of elements, $|S|$ denotes the cardinality of S , and n denotes the number of elements stored in D just prior to the operation in question.

is a small object to be passed by value. Increasing or decreasing an iterator must take amortized constant time. Also, constant-time access must be provided for the first and last element stored in D .

R_3 : Provide insertion and deletion operations that do not invalidate iterators (except the one being deleted). An iterator is *invalidated* if it points to a value which is either no longer in D , or differs from the one with which the iterator was created.

R_4 : Provide a constructor that constructs D from a sequence of values (e.g. by repeated insertion) at logarithmic cost per element; however, if the sequence is already sorted, the construction must have linear cost.

The most important operations to be supported and their cost requirements, as specified in the C++ standard, are summarized in Figure 1.

In addition to the requirements specified in the C++ standard, the aim is to develop a realization of an associative container that provides stronger guarantees than any of the existing realizations. In particular, the following guarantees would be of interest for the CPH STL project:

R_5 : Provide iterators, for which all operations take $O(1)$ time in the worst case. Many of the existing realizations fail to meet this requirement (or

even the weaker requirement specified in the C++ standard). Namely, the execution of a sequence of k ++ operations can take $\Omega(k + \lg n)$ time, where n is the current size of the associative container in question.

- R_6 : Provide the strong guarantee of exception safety for *all* operations on D . That is, if an exception is thrown, D should remain in the state it was before the operation started [26, Appendix E]. In theory, the strong guarantee of exception safety can be achieved with no penalty in asymptotic efficiency [15], but it has turned out to be difficult to write exception-safe code (cf. [26, p. 943]).
- R_7 : Provide location-based insertion and deletion operations that take $O(1)$ time in the worst case (not $O(1)$ amortized time as specified in the C++ standard). This performance can be achieved with the data structure described in [16] (see also [11]).
- R_8 : Provide container operations that are optimal with respect to the number of element comparisons performed. In a generic environment, the type of the elements being stored is not known at development time. Since the cost of element comparisons can be high compared to the cost of other primitives, this optimization can be significant in some applications. For red-black trees, searches and updates can perform up to about $2 \lg n$ element comparisons, whereas the bound of $\lg n + O(1)$ element comparisons is known to be achievable by other means [2].
- R_9 : Provide container operations that perform well on a hierarchical memory. A search tree that adapts optimally to the parameters of the memory hierarchy without knowing them beforehand, i.e. the data structure is *cache-oblivious*, is described in [4].
- R_{10} : Provide a separate core of D that collapses duplicates and keeps all equal elements together. This approach is suited for the realization of `multiset` and `multimap`. If all elements are equal, without collapsing `lower_bound` has logarithmic cost, whereas with collapsing its cost would be $O(1)$ [6].
- R_{11} : Provide a realization with small memory overhead without introducing any significant overhead in running times. For example, it is known that the memory overhead of red-black trees can be reduced from $4n + O(1)$ words to εn or $(1 + \varepsilon)n$ words, for any $\varepsilon > 0$ and $n > n(\varepsilon)$, depending on the way iterators are implemented [7].

The code base for any of the existing realizations available at the CPH STL [6, 13, 17] can be utilized for this project. Note, however, that these earlier prototype realizations do not provide the strong guarantees we aim at achieving in this project.

There are two *deliverables* you have to produce:

1. Midway report (\approx a sketch of the final progress report including the code produced so far) that will be reviewed by members of another project group. (You should also review one other midway report.) Details will be provided in Assignment 5.
2. Progress report that will be used as a starting point for the oral exams.

The code written should be provided in an appendix. (You can use this document as a model.)

In our evaluation, we give special importance on the following:

Algorithmic elegance: The STL is an algorithmic library so it is natural that the algorithmic arguments behind the data structure being developed must be correct and well understood by the developers.

Design choices: You are *not* expected to provide a realization that fulfils all requirements R_1 – R_{11} . Many of the requirements are conflicting which forces you to make design decisions. Good design demands good compromises!

Code quality: Maintenance is one of the challenges faced by library developers. Also, portability is an important issue. Good coding discipline is important when writing program libraries!

Tool usage: Subtle errors can be difficult to find without dedicated tools. You may consider using tools for unittesting to enhance the trust on your code and profiling to identify the performance bottlenecks in your code. Good handicraft requires good tools!

General impression: As additional (subjective) criteria, we look at the overall set-up for the project (ambitiousness) and the progress made during the project period (completeness).

Appendix: Source code

The source code given in this appendix can be used as an inspiration for the project. This appendix describes the bridge class `set` and sketches a single realization of that class based on red-black trees. All the files described, including the associated scripts, are available for download on the CPH STL website [10] (see menu item Downloads).

An associative container is a generic data structure which depends on several type parameters:

- V: the type of the *elements* (or *values*) manipulated;
- C: the type of the *comparator* which is a function object used in element comparisons;
- A: the type of the *allocator* which provides an interface to allocate, construct, destroy, and deallocate objects;
- N: the type of the *nodes* used for storing the elements and satellite data like pointers to other nodes; and
- R: the type of the *concrete data structure* (or *realization*) used for storing the nodes.

These variable names will be used for template parameters throughout this appendix.

A. Sets	6
<i>A.1 Program/set.h++</i>	6
<i>A.2 Program/set.c++</i>	8

B. Example	13
<i>B.1 A red-black tree storing 9 integers</i>	13
C. Red-black trees	13
<i>C.1 Program/red_black_tree.h++</i>	13
<i>C.2 Program/red_black_tree.c++</i>	16
D. Bases	22
<i>D.1 Program/management.h++</i>	22
E. Nodes	24
<i>E.1 Program/node.h++</i>	24
F. Property maps	27
<i>F.1 Program/property_maps.h++</i>	27
G. Traits	30
<i>G.1 Program/traits.h++</i>	30
H. Bidirectional iterators	30
<i>H.1 Program/bidirectional_iterator.h++</i>	30
I. Benz forms	34
<i>I.1 Benchmark/sort_int_pentium.py</i>	34
<i>I.2 Benchmark/experiment.c++</i>	35
<i>I.3 Benchmark/drive.c++</i>	36
J. UNIX makefiles	37
<i>J.1 Program/makefile</i>	37
<i>J.2 Benchmark/makefile</i>	37

A. Sets

A.1 Program/set.h++

```

/*
Author: Jyrki Katajainen, May 2007

A set is a reversible container which provides constant bidirectional
iterators to the elements stored.

This header is taken quite directly from the C++ standard [2003].
*/

#ifndef __CPHSTL_SET__
#define __CPHSTL_SET__

#include <functional> // defines std::less
#include <iterator> // defines std::reverse_iterator
#include <memory> // defines std::allocator
#include <utility> // defines std::pair
#include <algorithm> // defines std::equal and std::lexicographical_compare
#include "red_black_tree.h++" // defines cphstl::red_black_tree

namespace cphstl {

    namespace {

```

```

template <typename V>
class identity {
public:
    V operator()(V const& e) const {
        return e;
    }
    V& operator()(V& e) const {
        return e;
    }
};

} // unnamed namespace

template <
    typename V,
    typename C = std::less<V>,
    typename A = std::allocator<V>,
    typename R = cphstl::red_black_tree<V, V, identity<V>, C, A,
                                     cphstl::tree::node<V>, false>
>
class set {
public:
    // types

    typedef V key_type;
    typedef V value_type;
    typedef C key_compare;
    typedef C value_compare;
    typedef A allocator_type;
    typedef typename A::reference reference;
    typedef typename A::const_reference const_reference;
    typedef typename R::iterator iterator;
    typedef typename R::const_iterator const_iterator;
    typedef typename R::size_type size_type;
    typedef typename R::difference_type difference_type;
    typedef typename A::pointer pointer;
    typedef typename A::const_pointer const_pointer;
    typedef std::reverse_iterator<iterator> reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
    typedef R container_type;

protected:
    container_type container;

public:
    // structors

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

    template <typename I>
    set(I, I, key_compare const& = C(), allocator_type const& = A());

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

    // observers

    allocator_type get_allocator() const;
    key_compare key_comp() const;
    value_compare value_comp() const;

    // iterators

    iterator begin();

```

```

    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;

    // capacity

    bool empty() const;
    size_type size() const;
    size_type max_size() const;

    // modifiers

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

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

    void erase(iterator);
    size_type erase(key_type const&);
    void erase(iterator, iterator);
    void swap(set&);
    void clear();

    // visitors

    const_iterator find(key_type const&) const;
    size_type count(key_type const&) const;
    const_iterator lower_bound(key_type const&) const;
    const_iterator upper_bound(key_type const&) const;
    std::pair<const_iterator, const_iterator> equal_range(key_type const&) const;
};

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

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

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

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

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

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

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

} // namespace cphstl

#include "set.c++" // defines cphstl::set
#endif // __CPHSTL_SET__

```

A.2 Program/set.c++

```

/*
Author: Jyrki Katajainen, May 2007

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

namespace cphstl {

    namespace {

        template <typename K, typename V>
        class key_map {
        public:
            // static_assert: V ≡ std::pair<K, X>
            K operator[](V const& e) const {
                return e.first;
            }
        };

        template <typename K>
        class key_map<K, K> {
        public:
            K operator[](K const& k) const {
                return k;
            }
        };

    } // unnamed namespace

    template <typename V, typename C, typename A, typename R>
    set<V, C, A, R>::set(C const& comparator, A const& allocator) {
        : container(comparator, allocator) {
    }

    template <typename V, typename C, typename A, typename R>
    template <typename I>
    set<V, C, A, R>::set(I p, I q, C const& comparator, A const& allocator)
        : container(comparator, allocator) {
        container.insert(p, q);
    }

    template <typename V, typename C, typename A, typename R>
    set<V, C, A, R>::set(set const& other) : container(other.container) {
    }

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

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

    template <typename V, typename C, typename A, typename R>
    typename set<V, C, A, R>::allocator_type
    set<V, C, A, R>::get_allocator() const {
        return (*this).container.get_allocator();
    }

    template <typename V, typename C, typename A, typename R>
    typename set<V, C, A, R>::key_compare

```

```

set<V, C, A, R>::key_comp() const {
    return (*this).container.key_comp();
}

template <typename V, typename C, typename A, typename R>
typename set<V, C, A, R>::value_compare
set<V, C, A, R>::value_comp() const {
    return (*this).container.value_comp();
}

template <typename V, typename C, typename A, typename R>
typename set<V, C, A, R>::iterator
set<V, C, A, R>::begin() {
    return (*this).container.begin();
}

template <typename V, typename C, typename A, typename R>
typename set<V, C, A, R>::const_iterator
set<V, C, A, R>::begin() const {
    return const_iterator((*this).container.begin());
}

template <typename V, typename C, typename A, typename R>
typename set<V, C, A, R>::iterator
set<V, C, A, R>::end() {
    return (*this).container.end();
}

template <typename V, typename C, typename A, typename R>
typename set<V, C, A, R>::const_iterator
set<V, C, A, R>::end() const {
    return const_iterator((*this).container.end());
}

template <typename V, typename C, typename A, typename R>
typename set<V, C, A, R>::reverse_iterator
set<V, C, A, R>::rbegin() {
    return reverse_iterator((*this).end());
}

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

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

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

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

template <typename V, typename C, typename A, typename R>

```

```

typename set<V, C, A, R>::size_type
set<V, C, A, R>::size() const {
    return (*this).container.size();
}

template <typename V, typename C, typename A, typename R>
typename set<V, C, A, R>::size_type
set<V, C, A, R>::max_size() const {
    return (*this).container.max_size();
}

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

template <typename V, typename C, typename A, typename R>
typename set<V, C, A, R>::iterator
set<V, C, A, R>::insert(iterator p, value_type const& e) {
    return (*this).container.insert(p, e);
}

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

template <typename V, typename C, typename A, typename R>
void
set<V, C, A, R>::erase(iterator p) {
    (*this).container.erase(p);
}

template <typename V, typename C, typename A, typename R>
typename set<V, C, A, R>::size_type
set<V, C, A, R>::erase(key_type const& e) {
    return (*this).container.erase(e);
}

template <typename V, typename C, typename A, typename R>
void
set<V, C, A, R>::erase(iterator p, iterator q) {
    (*this).container.erase(p, q);
}

template <typename V, typename C, typename A, typename R>
void
set<V, C, A, R>::swap(set<V, C, A, R>& r) {
    (*this).container.swap(r.container);
}

template <typename V, typename C, typename A, typename R>
void
set<V, C, A, R>::clear() {
    (*this).container.clear();
}

template <typename V, typename C, typename A, typename R>
typename set<V, C, A, R>::const_iterator
set<V, C, A, R>::find(key_type const& k) const {
    const_iterator p = (*this).lower_bound(k);
    key_map<key_type, value_type> key;

```

```

    key_compare comp = (*this).key_comp();
    return (p ≡ (*this).end() || comp(k, key[*p]) ? (*this).end() : p);
}

template <typename V, typename C, typename A, typename R>
typename set<V, C, A, R>::size_type
set<V, C, A, R>::count(key_type const& k) const {
    return (*this).container.count(k);
}

template <typename V, typename C, typename A, typename R>
typename set<V, C, A, R>::const_iterator
set<V, C, A, R>::lower_bound(key_type const& k) const {
    return (*this).container.lower_bound(k);
}

template <typename V, typename C, typename A, typename R>
typename set<V, C, A, R>::const_iterator
set<V, C, A, R>::upper_bound(key_type const& k) const {
    return (*this).container.upper_bound(k);
}

template <typename V, typename C, typename A, typename R>
std::pair<typename set<V, C, A, R>::const_iterator,
         typename set<V, C, A, R>::const_iterator>
set<V, C, A, R>::equal_range(key_type const& k) const {
    return std::make_pair((*this).lower_bound(k), (*this).upper_bound(k));
}

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

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

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

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

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

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

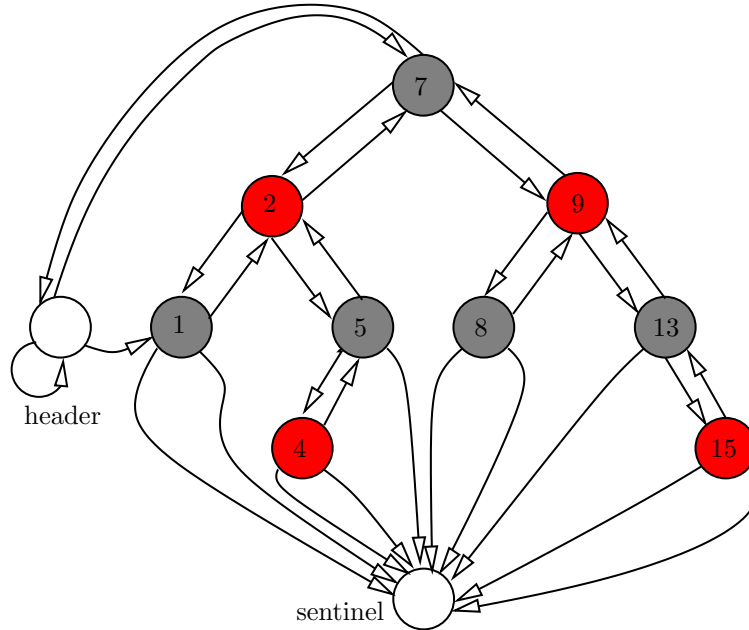
```

```

template <typename V, typename C, typename A, typename R>
void
swap(set<V, C, A, R>& r, set<V, C, A, R>& s) {
    r.swap(s);
}
} // namespace cphstl
    
```

B. Example

B.1 A red-black tree storing 9 integers



In the above picture only the tree pointers are shown. In addition to these tree pointers, each node has two list pointers which are used to keep the nodes in a doubly-linked list where the elements are stored in sorted order. There are two special dummies:

1. A unique *sentinel* to which other nodes point if some of their neighbouring nodes are missing. All pointers of this node point to the node itself.
2. The *header node* stores pointers pointing to the root (parent pointer), to the node storing the minimum (right-child pointer), and to the node storing the maximum (previous pointer), respectively. The left-child pointer points to the node itself to distinguish this node from normal nodes. The node is also used as the last node in the doubly-linked list tying the nodes together.

C. Red-black trees

C.1 Program/red_black_tree.h++

```

/*
Author: Jyrki Katajainen, May 2007

This class was designed to be used in the realization of the STL
associative containers (set, multiset, map, and multimap). The
insertion and deletion algorithms are based on those described in
[Cormen, Leiserson, Rivest, and Stein, Introduction to Algorithms, 2nd
Edition, The MIT Press (2001)].

In the header node, links are maintained to get a direct access to
the root, the leftmost node of the tree, to enable constant time
begin(), and to the rightmost node of the tree, to enable linear time
performance when used in the generic STL algorithms.

When a node being deleted has two children, its successor node is
relinked into its place, rather than copied, so that the only
iterators invalidated are those referring to the deleted node.

CPH STL guarantees:

- All member functions guarantee the complexity bounds specified in
  the C++ standard.

- The amount of extra space used is  $6n + O(1)$  words.

- Iterator operations take constant time in the worst case.

- Iterators remain valid throughout their life time.
*/

#ifndef __CPHSTL_RED_BLACK_TREE__
#define __CPHSTL_RED_BLACK_TREE__

#include <functional> // defines std::less
#include <memory> // defines std::allocator
#include <utility> // defines std::pair
#include "node.h++" // defines the nodes used
#include "management.h++" // defines the management classes used
#include "property_maps.h++" // defines the property maps used
#include "traits.h++" // defines the type traits
#include "bidirectional_iterator.h++" // defines the iterators used

namespace cphstl {

    namespace tree {

        template <typename T>
        class identity {
        public:
            T operator()(T const& t) const {
                return t;
            }
            T& operator()(T& t) const {
                return t;
            }
        };

    } // namespace tree

    template <
        typename K,
        typename V = K,
        typename F = cphstl::tree::identity<V>,
        typename C = std::less<K>,
        typename A = std::allocator<V>,

```

```

    typename N = cphstl::tree::node<V>,
    bool is_multiset = false
>
class red_black_tree
    : public cphstl::tree::value_management<K, V, C, A, N> {
public:
    // types

    typedef K key_type;
    typedef V value_type;
    typedef C key_compare;
    typedef typename cphstl::if_then_else<cphstl::types<K, V>::are_same,
        C, cphstl::tree::value_compare<K, V, C> >::type value_compare;
    typedef typename A::template rebind<value_type>::other allocator_type;
    typedef typename allocator_type::reference reference;
    typedef typename allocator_type::const_reference const_reference;
    typedef cphstl::bidirectional_iterator<cphstl::tree::value_map<N, V>,
        cphstl::tree::successor_map<N>,
        cphstl::tree::predecessor_map<N>,
        false> iterator;
    typedef cphstl::bidirectional_iterator<cphstl::tree::value_map<N, V>,
        cphstl::tree::successor_map<N>,
        cphstl::tree::predecessor_map<N>,
        true> const_iterator;
    typedef typename allocator_type::size_type size_type;
    typedef typename allocator_type::difference_type difference_type;
    typedef typename allocator_type::pointer pointer;
    typedef typename allocator_type::const_pointer const_pointer;

protected:
    typedef red_black_tree<K, V, F, C, A, N, is_multiset> container_type;
    typedef N node_type;
    typedef typename N::link_type link_type;
    typedef typename N::colour_type colour_type;
    typedef cphstl::tree::comparator_management<K, V, C> comparator_management;
    typedef cphstl::tree::node_management<K, V, C, A, N> node_management;
    typedef cphstl::tree::link_management<K, V, C, A, N> link_management;
    typedef cphstl::tree::colour_management<K, V, C, A, N> colour_management;
    typedef cphstl::tree::value_management<K, V, C, A, N> value_management;

    link_type header;
    size_type n;

public:
    // structs

    explicit red_black_tree(key_compare const& = key_compare(),
        allocator_type const& = allocator_type());
    red_black_tree(red_black_tree const&);
    ~red_black_tree();
    red_black_tree& operator=(red_black_tree const&);

    // observers

    allocator_type get_allocator() const;
    key_compare key_comp() const;
    value_compare value_comp() const;

    // iterators

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

```

```

    // capacity
    size_type size() const;
    size_type max_size() const;

    // modifiers

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

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

    void erase(iterator);
    size_type erase(key_type const&);
    void erase(iterator, iterator);
    void swap(red_black_tree&);
    void clear();

    // visitors

    size_type count(key_type const&) const;
    iterator lower_bound(key_type const&);
    const_iterator lower_bound(key_type const&) const;
    iterator upper_bound(key_type const&);
    const_iterator upper_bound(key_type const&) const;

protected:
    // helpers

    link_type construct_node();
    void construct_value(link_type, value_type const&);
    void destroy_value(link_type);
    void destroy_node(link_type);
    void initialize_header();
};

} // namespace cphstl

#include "red_black_tree.c++" // defines cphstl::red_black_tree
#endif // __CPHSTL_RED_BLACK_TREE__

C.2 Program/red_black_tree.c++

/*
Author: Jyrki Katajainen, May 2007
*/

#include <algorithm> // defines std::swap
#include <iostream>

namespace cphstl {

    // structors

    template <typename K, typename V, typename F, typename C, typename A, typename N, bool B>
    red_black_tree<K, V, F, C, A, N, B>::red_black_tree(key_compare const& c,
                                                       allocator_type const& a)
        : value_management(c, a), n(size_type(0)) {
        (*this).header = (*this).construct_node();
        (*this).initialize_header();
    }

    template <typename K, typename V, typename F, typename C, typename A, typename N, bool B>
    red_black_tree<K, V, F, C, A, N, B>::red_black_tree(red_black_tree const& other)

```

```

: value_management(other.comparator, other.value_allocator), n() {
(*this).header = (*this).construct_node();
(*this).initialize_header();
try {
    (*this).insert(other.begin(), other.end());
}
catch (...) {
    (*this).clear();
    (*this).destroy_node((*this).header);
    (*this).n = size_type(0);
    (*this).header = 0;
    throw;
}
}

template <typename K, typename V, typename F, typename C, typename A, typename N, bool B>
red_black_tree<K, V, F, C, A, N, B>::~red_black_tree() {
    (*this).clear();
    (*this).destroy_node((*this).header);
    (*this).n = 0;
    (*this).header = 0;
}

template <typename K, typename V, typename F, typename C, typename A, typename N, bool B>
red_black_tree<K, V, F, C, A, N, B>&
red_black_tree<K, V, F, C, A, N, B>::operator=(red_black_tree const& other) {
    if (this != &other) {
        typename N::link_type p = (*this).construct_node();
        std::swap(p, (*this).header);
        (*this).initialize_header();
        try {
            (*this).insert(other.begin(), other.end()); //only this can fail
            std::swap(p, (*this).header);
            (*this).clear();
            std::swap(p, (*this).header);
            (*this).destroy_node(p);
            (*this).n = other.n;
            (*this).comparator = other.comparator;
            (*this).node_allocator = other.node_allocator;
            (*this).link_allocator = other.link_allocator;
            (*this).colour_allocator = other.colour_allocator;
            (*this).value_allocator = other.value_allocator;
        }
        catch(...) {
            (*this).clear();
            std::swap(p, (*this).header);
            (*this).destroy_node(p);
            throw;
        }
    }
    return *this;
}

// observers

template <typename K, typename V, typename F, typename C, typename A, typename N, bool B>
typename red_black_tree<K, V, F, C, A, N, B>::allocator_type
red_black_tree<K, V, F, C, A, N, B>::get_allocator() const {
    return (*this).value_allocator;
}

template <typename K, typename V, typename F, typename C, typename A, typename N, bool B>
typename red_black_tree<K, V, F, C, A, N, B>::key_compare
red_black_tree<K, V, F, C, A, N, B>::key_comp() const {
    return (*this).comparator;
}

```

```

}

template <typename K, typename V, typename F, typename C, typename A, typename N, bool B>
typename red_black_tree<K, V, F, C, A, N, B>::value_compare
red_black_tree<K, V, F, C, A, N, B>::value_comp() const {
    return value_compare((*this).comparator);
}

// iterators

template <typename K, typename V, typename F, typename C, typename A, typename N, bool B>
typename red_black_tree<K, V, F, C, A, N, B>::iterator
red_black_tree<K, V, F, C, A, N, B>::begin() {
    USE_TREE_PROPERTY_MAPS(V, N)
    return iterator(right[(*this).header]);
}

template <typename K, typename V, typename F, typename C, typename A, typename N, bool B>
typename red_black_tree<K, V, F, C, A, N, B>::const_iterator
red_black_tree<K, V, F, C, A, N, B>::begin() const {
    USE_TREE_PROPERTY_MAPS(V, N)
    return const_iterator(right[(*this).header]);
}

template <typename K, typename V, typename F, typename C, typename A, typename N, bool B>
typename red_black_tree<K, V, F, C, A, N, B>::iterator
red_black_tree<K, V, F, C, A, N, B>::end() {
    return iterator((*this).header);
}

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

// capacity

template <typename K, typename V, typename F, typename C, typename A, typename N, bool B>
typename red_black_tree<K, V, F, C, A, N, B>::size_type
red_black_tree<K, V, F, C, A, N, B>::size() const {
    return size_type((*this).n);
}

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

// modifiers

template <typename K, typename V, typename F, typename C, typename A, typename N, bool B>
std::pair<typename red_black_tree<K, V, F, C, A, N, B>::iterator, bool>
red_black_tree<K, V, F, C, A, N, B>::insert(value_type const& e) {
    ++(*this).n;
    // not implemented
    return std::make_pair((*this).end(), false);
}

```

```

template <typename K, typename V, typename F, typename C, typename A, typename N, bool B>
typename red_black_tree<K, V, F, C, A, N, B>::iterator
red_black_tree<K, V, F, C, A, N, B>::insert(iterator p, value_type const& e) {
    ++(*this).n;
    // not implemented
    return (*this).end();
}

template <typename K, typename V, typename F, typename C, typename A, typename N, bool B>
template <typename I>
void
red_black_tree<K, V, F, C, A, N, B>::insert(I p, I q) {
    // not implemented
}

template <typename K, typename V, typename F, typename C, typename A, typename N, bool B>
void
red_black_tree<K, V, F, C, A, N, B>::erase(iterator p) {
    --(*this).n;
    // not implemented
}

template <typename K, typename V, typename F, typename C, typename A, typename N, bool B>
typename red_black_tree<K, V, F, C, A, N, B>::size_type
red_black_tree<K, V, F, C, A, N, B>::erase(key_type const& e) {
    --(*this).n;
    // not implemented
    return size_type(0);
}

template <typename K, typename V, typename F, typename C, typename A, typename N, bool B>
void
red_black_tree<K, V, F, C, A, N, B>::erase(iterator p, iterator q) {
    // not implemented
}

template <typename K, typename V, typename F, typename C, typename A, typename N, bool B>
void
red_black_tree<K, V, F, C, A, N, B>::swap(red_black_tree& other) {
    if ((*this).get_allocator() == other.get_allocator()) {
        std::swap((*this).comparator, other.comparator);
        std::swap((*this).header, other.header);
        std::swap((*this).n, other.n);
    }
    else {
        container_type t = *this;
        *this = other;
        other = t;
    }
}

template <typename K, typename V, typename F, typename C, typename A, typename N, bool B>
void
red_black_tree<K, V, F, C, A, N, B>::clear() {
    iterator p = (*this).begin();
    while (p != (*this).end()) {
        iterator q = p;
        ++p;
        (*this).destroy_value(link_type(q));
        (*this).destroy_node(link_type(q));
    }
}

// visitors

```

```

template <typename K, typename V, typename F, typename C, typename A, typename N, bool B>
typename red_black_tree<K, V, F, C, A, N, B>::size_type
red_black_tree<K, V, F, C, A, N, B>::count(key_type const& e) const {
    // not implemented
    return (*this).n;
}

template <typename K, typename V, typename F, typename C, typename A, typename N, bool B>
typename red_black_tree<K, V, F, C, A, N, B>::iterator
red_black_tree<K, V, F, C, A, N, B>::lower_bound(key_type const& k) {
    // not implemented
    return (*this).end();
}

template <typename K, typename V, typename F, typename C, typename A, typename N, bool B>
typename red_black_tree<K, V, F, C, A, N, B>::const_iterator
red_black_tree<K, V, F, C, A, N, B>::lower_bound(key_type const& k) const {
    // not implemented
    return (*this).end();
}

template <typename K, typename V, typename F, typename C, typename A, typename N, bool B>
typename red_black_tree<K, V, F, C, A, N, B>::iterator
red_black_tree<K, V, F, C, A, N, B>::upper_bound(key_type const& k) {
    // not implemented
    return (*this).end();
}

template <typename K, typename V, typename F, typename C, typename A, typename N, bool B>
typename red_black_tree<K, V, F, C, A, N, B>::const_iterator
red_black_tree<K, V, F, C, A, N, B>::upper_bound(key_type const& k) const {
    // not implemented
    return (*this).end();
}

// helpers

template <typename K, typename V, typename F, typename C, typename A, typename N, bool B>
typename red_black_tree<K, V, F, C, A, N, B>::link_type
red_black_tree<K, V, F, C, A, N, B>::construct_node() {
    typename N::link_type p = (*this).node_allocator.allocate(1);
    (*this).link_allocator.construct(&(*p).parent, &(*p).sentinel);
    (*this).link_allocator.construct(&(*p).left, &(*p).sentinel);
    (*this).link_allocator.construct(&(*p).right, &(*p).sentinel);
    (*this).link_allocator.construct(&(*p).previous, &(*p).sentinel);
    (*this).link_allocator.construct(&(*p).next, &(*p).sentinel);
    (*this).colour_allocator.construct(&(*p).colour, cphstl::tree::node<V>::red);
    return p;
}

template <typename K, typename V, typename F, typename C, typename A, typename N, bool B>
void
red_black_tree<K, V, F, C, A, N, B>::construct_value(link_type p, value_type const& v) {
    (*this).value_allocator.construct(&(*p).value, v);
}

template <typename K, typename V, typename F, typename C, typename A, typename N, bool B>
void
red_black_tree<K, V, F, C, A, N, B>::destroy_value(link_type p) {
    (*this).value_allocator.destroy(&(*p).value);
}

template <typename K, typename V, typename F, typename C, typename A, typename N, bool B>
void
red_black_tree<K, V, F, C, A, N, B>::destroy_node(link_type p) {

```

```

    (*this).colour_allocator.destroy(&(*p).colour);
    (*this).link_allocator.destroy(&(*p).next);
    (*this).link_allocator.destroy(&(*p).previous);
    (*this).link_allocator.destroy(&(*p).right);
    (*this).link_allocator.destroy(&(*p).left);
    (*this).link_allocator.destroy(&(*p).parent);
    (*this).node_allocator.deallocate(p, 1);
}

template <typename K, typename V, typename F, typename C, typename A, typename N, bool B>
void
red_black_tree<K, V, F, C, A, N, B>::initialize_header() {
    link_type p = (*this).header;
    USE_TREE_PROPERTY_MAPS(V, N)
    predecessor[p] = p; successor[p] = p;
    parent[p] = p; left[p] = p; right[p] = p;
    colour[p] = N::red;
}

} // namespace cphstl

#undef USE_TREE_PROPERTY_MAPS
#if defined(UNITTEST_RED_BLACK_TREE)

#include <cassert>
#include <memory>
#include <algorithm>
#include <numeric>
#include <set>

template <typename V>
class unittest_red_black_tree {
public:

    void operator()() {
        typedef cphstl::red_black_tree<V> set;
        //typedef std::set<V> set;
        set t;
        assert(t.size() == 0);
        assert(t.begin() == t.end());
        typename set::allocator_type a = t.get_allocator();
        std::allocator<V> std_a;
        assert(a == std_a);
        typename set::key_compare c = t.key_compare();
        assert(t.size() <= t.max_size());
        V e[] = {8, 10, 13, 1, 6, 5, 3, 7, 11, 9};
        unsigned int n = sizeof(e)/sizeof(V);

        t.insert(e[0]);
        typename set::iterator p = t.lower_bound(e[0]);
        assert(*p == V(8));
        //**p = 12; /* Compilation should fail */
        for (typename set::iterator q = t.begin(); q != t.end(); ) {
            ++q;
        }
        for (typename set::iterator q = t.end(); q != t.begin(); ) {
            --q;
        }
        for (unsigned int j = 1; j < n; ++j) {
            t.insert(e[j]);
            for (typename set::iterator p = t.begin(); p != t.end(); ) {
                ++p;
            }
            for (typename set::iterator p = t.end(); p != t.begin(); ) {
                --p;
            }
        }
    }
};

```

```

    }
  }
  assert(t.size() == n);

  V sum = V(0);
  for (typename set::iterator p = t.begin(); p != t.end(); ++p) {
    sum += *p;
  }
  assert(sum == std::accumulate(&e[0], &e[n], V(0)));
}
};

int main(int, char**) {
  unittest_red_black_tree<int> t;
  t();
  unittest_red_black_tree<float> u;
  u();
}

#endif // UNITTEST_RED_BLACK_TREE

```

D. Bases

D.1 Program/management.h++

```

/*
Author: Jyrki Katajainen, May 2007

The idea of using a hierarchy of management classes is taken from the
book [P.J. Plauger, Alexander A. Stepanov, Meng Lee, and David
R. Musser. The C++ Standard Template Library. Prentice Hall PTR
(2001)]. The main reason for using this is to save space for the empty
allocator classes (empty-base-class optimization).
*/

#include <functional> // defines std::binary_function

namespace cphstl {

  namespace tree {

    template <typename K, typename V>
    class key_map {
    public:
      // static_assert: V == std::pair<K, X>
      K operator[](V const& e) const {
        return e.first;
      }
    };

    template <typename K>
    class key_map<K, K> {
    public:
      K operator[](K const& k) const {
        return k;
      }
    };

    template <typename K, typename V, typename C>
    class value_compare
      : public std::binary_function<V, V, bool> {
    public:
      typedef K key_type;
      typedef V value_type;

```

```

typedef C key_compare;

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

    key_compare comparator;
};

template <typename K, typename V, typename C>
class comparator_management
    : public cphstl::tree::value_compare<K, V, C> {
protected:
    typedef K key_type;
    typedef V value_type;
    typedef C key_compare;

    comparator_management(key_compare const& c) : value_compare<K, V, C>(c) {
    }
};

template <typename K, typename V, typename C, typename A, typename N>
class node_management
    : public cphstl::tree::comparator_management<K, V, C > {
protected:
    typedef K key_type;
    typedef V value_type;
    typedef C key_compare;
    typedef typename A::template rebind<value_type>::other allocator_type;
    typedef N node_type;
    typedef typename N::link_type link_type;
    typedef typename N::colour_type colour_type;

    node_management(key_compare const& c, allocator_type const& a)
        : comparator_management<K, V, C>(c), node_allocator(a) {
    }

    typename allocator_type::template rebind<N>::other node_allocator;
};

template <typename K, typename V, typename C, typename A, typename N>
class link_management
    : public cphstl::tree::node_management<K, V, C, A, N> {
protected:
    typedef K key_type;
    typedef V value_type;
    typedef C key_compare;
    typedef typename A::template rebind<value_type>::other allocator_type;
    typedef N node_type;
    typedef typename N::link_type link_type;
    typedef typename N::colour_type colour_type;

    link_management(key_compare const& c, allocator_type const& a)
        : node_management<K, V, C, A, N>(c, a), link_allocator(a) {
    }

    typename allocator_type::template rebind<link_type>::other link_allocator;
};

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

```

```

class colour_management
  : public cphstl::tree::link_management<K, V, C, A, N> {
protected:
  typedef K key_type;
  typedef V value_type;
  typedef C key_compare;
  typedef typename A::template rebind<value_type>::other allocator_type;
  typedef N node_type;
  typedef typename N::link_type link_type;
  typedef typename N::colour_type colour_type;

  colour_management(key_compare const& c, allocator_type const& a)
    : link_management<K, V, C, A, N>(c, a), colour_allocator(a) {
  }

  typename allocator_type::template rebind<colour_type>::other colour_allocator;
};

template <typename K, typename V, typename C, typename A, typename N>
class value_management
  : public cphstl::tree::colour_management<K, V, C, A, N> {
protected:
  typedef K key_type;
  typedef V value_type;
  typedef C key_compare;
  typedef typename A::template rebind<value_type>::other allocator_type;
  typedef N node_type;
  typedef typename N::link_type link_type;
  typedef typename N::colour_type colour_type;

  value_management(key_compare const& c, allocator_type const& a)
    : colour_management<K, V, C, A, N>(c, a), value_allocator(a) {
  }
  allocator_type value_allocator;
};

} // namespace tree
} // namespace cphstl

```

E. Nodes

E.1 Program/node.h++

```

/*
Author: Jyrki Katajainen, May 2007

There are two types of nodes: dummies that do not carry any data and
normal nodes that carry some data. A sentinel is a dummy to which
other nodes point if some of their neighbouring nodes are missing. A
header is another dummy via which the whole tree can be accessed.
*/

#ifndef __CPHSTL_NODE__
#define __CPHSTL_NODE__

namespace cphstl {

  namespace tree {

    template <typename V>
    class node {
    public:
      typedef V value_type;

```

```

enum colour_type {black = true, red = false};
typedef node<V> node_type;
typedef node_type* link_type;

link_type parent;
link_type left;
link_type right;
link_type next;
link_type previous;
colour_type colour;
value_type value;

static link_type nil;
static node_type sentinel;

node()
: parent(&sentinel), left(&sentinel), right(&sentinel),
  next(&sentinel), previous(&sentinel), colour(red) {
}

node(value_type const& v)
: parent(&sentinel), left(&sentinel), right(&sentinel),
  next(&sentinel), previous(&sentinel), colour(red), value(v) {
}
};

template <typename V>
node<V> node<V>::sentinel;

template <typename V>
node<V>* node<V>::nil = &node<V>::sentinel;

} // namespace tree
} // namespace cphstl

#if defined(UNITTEST_TREE_NODE)

#include <cassert> // defines assert macro
#include "bidirectional_iterator.h++" // defines cphstl::bidirectional_iterator
#include "property_maps.h++" // defines the property maps used

class unittest_tree_node {
public:

void operator()() {
typedef cphstl::tree::node<int> node;
typedef node::link_type pointer;
pointer past_the_end = new node();
pointer o = new node(1);
pointer p = new node(2);
pointer q = new node(4);
pointer r = new node(5);
pointer s = new node(7);
pointer t = new node(8);
pointer u = new node(9);
pointer v = new node(13);
pointer w = new node(15);

USE_TREE_PROPERTY_MAPS(int, node)
parent[past_the_end] = s; left[past_the_end] = w;
right[past_the_end] = o; successor[past_the_end] = past_the_end;
predecessor[past_the_end] = w;

parent[q] = r; predecessor[q] = p; successor[q] = r;

```

```

parent[w] = v; predecessor[w] = v; successor[w] = past_the_end;

parent[o] = p; predecessor[o] = past_the_end; successor[o] = p;
parent[r] = p; left[r] = q; predecessor[r] = q; successor[r] = s;
parent[t] = u; predecessor[t] = s; successor[t] = u;
parent[v] = u; right[v] = w; predecessor[v] = u; successor[v] = w;

parent[p] = s; left[p] = o; right[p] = r; predecessor[p] = o;
successor[p] = q;
parent[u] = s; left[u] = t; right[u] = v; predecessor[u] = t;
successor[u] = v;
parent[s] = past_the_end; left[s] = p; right[s] = u; predecessor[s] = r;
successor[s] = t;

typedef cphstl::bidirectional_iterator<
    cphstl::tree::value_map<node, int>,
    cphstl::tree::successor_map<node>,
    cphstl::tree::predecessor_map<node> > iterator;

iterator x; // default constructor
iterator first(o); //parametrized constructor
x = first; // operator= (generated automatically by the compiler)
assert(*x == 1); // operator*
iterator last(past_the_end); // copy constructor

assert(*first == 1);
++first; // operator++
assert(*first == 2);
first++; // operator++(int)
assert(*first == 4);
--first; // operator--
assert(*first == 2);
first++; // operator++(int)
first--; // operator--(int)
assert(*first == 2);

int count = 0;
for (iterator i(o); i != iterator(past_the_end); ++i) {
    ++count;
}
assert(count == 9);

iterator i(past_the_end);
do {
    --i;
    --count;
}
while (i != iterator(o));
assert(count == 0);

typedef cphstl::bidirectional_iterator<
    cphstl::tree::value_map<node, int>,
    cphstl::tree::successor_map<node>,
    cphstl::tree::predecessor_map<node>, true> const_iterator;

const_iterator second(first);
assert(*second == 2);
//iterator no_good(second); // Compilation should fail
//*second = 7; // Compilation should fail
iterator other(p);

assert(second == other); //operator ==
++second;
assert(second != other); //operator !=

```

```

    delete w;
    delete v;
    delete u;
    delete t;
    delete s;
    delete r;
    delete q;
    delete p;
    delete o;
    delete past_the_end;
}
};

int main(int, char**) {
    unittest_tree_node t;
    t();
    return 0;
}

#endif // UNITTEST_TREE_NODE
#endif // __CPHSTL_TREE_NODE__

```

F. Property maps

F.1 Program/property_maps.h++

```

/*
Author: Jyrki Katajainen, May 2007

To learn more about property maps, read the documentation available at
www.boost.org.
*/

#ifndef __CPHSTL_TREE_PROPERTY_MAPS__
#define __CPHSTL_TREE_PROPERTY_MAPS__
#include <iostream> // defines standard streams

namespace cphstl {

    namespace tree {

        template <typename N>
        class id_map {
        public:
            typedef N* domain_type;
            typedef N* range_type;

            range_type& operator[](domain_type p) const {
                return p;
            }
        };

        template <typename N, typename V>
        class value_map {
        public:
            typedef N* domain_type;
            typedef V range_type;

            V& operator[](domain_type p) const {
                return (*p).value;
            }
        };

        template <typename N>

```

```

class parent_map {
public:
    typedef N* domain_type;
    typedef N* range_type;

    range_type& operator[](domain_type p) const {
        return (*p).parent;
    }
};

template <typename N>
class left_map {
public:
    typedef N* domain_type;
    typedef N* range_type;

    range_type& operator[](domain_type p) const {
        return (*p).left;
    }
};

template <typename N>
class right_map {
public:
    typedef N* domain_type;
    typedef N* range_type;

    range_type& operator[](domain_type p) const {
        return (*p).right;
    }
};

template <typename N>
class successor_map {
public:
    typedef N* domain_type;
    typedef N* range_type;

    range_type& operator[](domain_type p) const {
        return (*p).next;
    }
};

template <typename N>
class predecessor_map {
public:
    typedef N* domain_type;
    typedef N* range_type;

    range_type& operator[](domain_type p) const {
        return (*p).previous;
    }
};

template <typename N>
class colour_map {
public:
    typedef N* domain_type;
    typedef typename N::colour_type range_type;

    range_type& operator[](domain_type p) const {
        return (*p).colour;
    }
};

```

```

template <typename N>
class is_nil_map {
public:
    typedef N* domain_type;
    typedef bool range_type;

    range_type operator[](domain_type p) const {
        return p ≡ (*p).nil;
    }
};

template <typename N>
class is_past_the_end_map {
public:
    typedef N* domain_type;
    typedef bool range_type;

    range_type operator[](domain_type p) const {
        cphstl::tree::left_map<N> left;
        cphstl::tree::is_nil_map<N> is_nil;
        return (!is_nil[p]) && (p ≡ left[p]);
    }
};

template <typename N>
class is_root_map {
public:
    typedef N* domain_type;
    typedef bool range_type;

    range_type operator[](domain_type p) const {
        cphstl::tree::is_past_the_end_map<N> is_past_the_end;
        cphstl::tree::parent_map<N> parent;
        return is_past_the_end[parent[p]];
    }
};

template <typename T>
inline void
ignore_unused_variable_warning(T const&) {
}

#define USE_TREE_PROPERTY_MAPS(V, N) \
cphstl::tree::parent_map<N > parent; \
cphstl::tree::left_map<N > left; \
cphstl::tree::right_map<N > right; \
cphstl::tree::successor_map<N > successor; \
cphstl::tree::predecessor_map<N > predecessor; \
cphstl::tree::colour_map<N > colour; \
cphstl::tree::is_nil_map<N > is_nil; \
cphstl::tree::is_past_the_end_map<N > is_past_the_end; \
cphstl::tree::is_root_map<N > is_root; \
cphstl::tree::value_map<N, V > value; \
cphstl::tree::ignore_unused_variable_warning(parent); \
cphstl::tree::ignore_unused_variable_warning(left); \
cphstl::tree::ignore_unused_variable_warning(right); \
cphstl::tree::ignore_unused_variable_warning(successor); \
cphstl::tree::ignore_unused_variable_warning(predecessor); \
cphstl::tree::ignore_unused_variable_warning(colour); \
cphstl::tree::ignore_unused_variable_warning(is_nil); \
cphstl::tree::ignore_unused_variable_warning(is_past_the_end); \
cphstl::tree::ignore_unused_variable_warning(is_root); \
cphstl::tree::ignore_unused_variable_warning(value);

} // namespace tree

```

```

} // namespace cphstl

#endif // __CPHSTL_TREE_PROPERTY_MAPS__

```

G. Traits

G.1 Program/traits.h++

```

/*
Author: Jyrki Katajainen, May 2007

Compile time if-then-else construct is part of C++ folklore which is
described, for example, in [David Vandervoorde and Nicolai M.
Josuttis, C++ Templates: The Complete Guide, Addison-Wesley (2003)].
*/

#ifndef __CPHSTL_TYPE_TRAITS__
#define __CPHSTL_TYPE_TRAITS__

namespace cphstl {

    template <bool, typename T, typename U>
    class if_then_else;

    template <typename T, typename U>
    class if_then_else<true, T, U> {
    public:
        typedef T type;
    };

    template <typename T, typename U>
    class if_then_else<false, T, U> {
    public:
        typedef U type;
    };

    template <typename X, typename Y>
    class types {
    public:
        enum { are_same = 0 };
    };

    template <typename X>
    class types<X, X> {
    public:
        enum { are_same = 1 };
    };

} // namespace cphstl

#endif // __CPHSTL_TYPE_TRAITS__

```

H. Bidirectional iterators

H.1 Program/bidirectional_iterator.h++

```

/*
Author: Jyrki Katajainen, May 2006

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

```

```

*/

#ifndef __CPHSTL_BIDIRECTIONAL_ITERATOR__
#define __CPHSTL_BIDIRECTIONAL_ITERATOR__

#include <iterator> // defines std::bidirectional_iterator_tag
#include "traits.h++" // defines cphstl::if_then_else
#include <cstddef> // defines std::ptrdiff_t

namespace cphstl {

    template <typename Value, typename Succ, typename Pred, bool is_const = false>
    class bidirectional_iterator {
    protected:

        typedef typename Value::range_type V;
        typedef typename Value::domain_type D;
        typedef typename cphstl::if_then_else<is_const, const D, D>::type node_pointer;
        D position;

    public:
        typedef std::bidirectional_iterator_tag iterator_category;
        typedef V value_type;
        typedef std::ptrdiff_t difference_type;
        typedef typename if_then_else<is_const, V const*, V*>::type pointer;
        typedef typename if_then_else<is_const, V const&, V&>::type reference;

        bidirectional_iterator() : position(0) {
        }

        bidirectional_iterator(node_pointer p) : position(p) {
        }

        bidirectional_iterator(bidirectional_iterator<Value, Succ, Pred> const& i)
            : position(i.position) {
        }

        operator D () const { // I am not quite satisfied with this solution.
            return position;
        }

        reference
        operator*() const {
            return Value()[position]; //(*position).value;
        }

        pointer
        operator->() const {
            return &Value()[position];
        }

        bidirectional_iterator&
        operator++() {
            position = Succ()[position];
            return *this;
        }

        bidirectional_iterator
        operator++(int) {
            bidirectional_iterator temp = *this;
            ++(*this);
            return temp;
        }

        bidirectional_iterator&

```

```

operator--() {
    position = Pred()[position];
    return *this;
}

bidirectional_iterator
operator--(int) {
    bidirectional_iterator temp(*this);
    --(*this);
    return temp;
}

friend class bidirectional_iterator<Value, Succ, Pred, !is_const>;

template <bool both>
bool
operator==(bidirectional_iterator<Value, Succ, Pred, both> const& f) const {
    return (*this).position == f.position;
}

template <bool both>
bool
operator!=(bidirectional_iterator<Value, Succ, Pred, both> const& f) const {
    return (*this).position != f.position;
}
};

} // namespace cphstl

#ifdef UNITTEST_BIDIRECTIONAL_ITERATOR
#include <cassert> // defines assert macro

class unittest_bidirectional_iterator {
public:

    template <typename T>
    class list_node {
    public:
        T value;
        list_node* next;
        list_node* prev;
        list_node(T const& t, list_node* p, list_node* q)
            : value(t), next(p), prev(q) {
        }
    };

    template <typename V, typename N>
    class null_map {
    public:
        typedef N* domain_type;
        typedef V range_type;
    };

    template <typename V, typename N>
    class value_map {
    public:
        typedef N* domain_type;
        typedef V range_type;

        range_type& operator[](domain_type x) const {
            return (*x).value;
        }
    };

};

template <typename N>

```

```

class next_map {
public:
    typedef N* domain_type;
    typedef N* range_type;

    range_type& operator[](domain_type x) const {
        return (*x).next;
    }
};

template <typename N>
class prev_map {
public:
    typedef N* domain_type;
    typedef N* range_type;

    range_type& operator[](domain_type x) const {
        return (*x).prev;
    }
};

class point {
public:
    double x_;
    double y_;
    point(double x = 0.0, double y = 0.0) :x_(x), y_(y) {
    }
};

void operator()() {
    list_node<int>* p = new list_node<int>(1, 0, 0);
    list_node<int>* q = new list_node<int>(2, 0, p);
    list_node<int>* r = new list_node<int>(3, 0, q);
    (*p).next = q; (*q).next = r;

    typedef cphstl::bidirectional_iterator<
        value_map<int, list_node<int> >, next_map<list_node<int> >,
        prev_map<list_node<int> > > iterator;

    iterator x; // default constructor
    iterator first(p); //parametrized constructor
    list_node<int>* u(first); // conversion: iterator --> pointer
    x = iterator(u);
    x = first; // operator= (generated automatically by the compiler)
    assert(*x == 1); // operator*
    iterator third(r); // parametrized constructor
    iterator last(third); // copy constructor
    assert(*last == 3);

    assert(*first == 1);
    ++first; // operator++
    assert(*first == 2);

    --first; // operator--
    assert(*first == 1);

    first++; // operator++(int)
    first--; // operator--(int)
    assert(*first == 1);

    typedef cphstl::bidirectional_iterator<
        value_map<int, list_node<int> >, next_map<list_node<int> >,
        prev_map<list_node<int> >, true> const_iterator;

    const_iterator second(q);

```

```

assert(*second == 2);
/*second = 7; /* Compilation should fail */

--second;
assert(first == second); //operator ==
assert(first != third); //operator !=

point centre(1.0, 2.0);
list_node<point>* a = new list_node<point>(centre, 0, 0);

typedef cphstl::bidirectional_iterator<
    value_map<point, list_node<point> >,
    null_map<point, list_node<point> >,
    null_map<point, list_node<point> > > point_iterator;

point_iterator middle(a);
assert(middle->x_ == 1.0); // operator->

delete r;
delete q;
delete p;
delete a;
}
};

int main(int, char**) {
    unittest_bidirectional_iterator t;
    t();
    return 0;
}

#endif // UNITTEST_BIDIRECTIONAL_ITERATOR
#endif // __CPHSTL_BIDIRECTIONAL_ITERATOR__

```

I. Benz forms

I.1 Benchmark/sort_int_pentium.py

```

import benz
import os

class case(benz.case):
    def __init__(self, n, element_type, set_type, include_file, macro):
        benz.case.__init__(self)
        self.n = n
        # self.computer = 'cphstl.projektlab.diku.dk'
        # self.connection_protocol = 'ssh'
        # self.transfer_protocol = 'scp'
        self.compiler = 'g++'
        self.compiler_options.extend(['-O3', '-DNUMBER_OF_ELEMENTS=' + str(n), '-D' + macro])
        home = os.environ['HOME']
        self.include_paths.extend([ \
            home + 'CPHSTL/Report/Associative-containers/Program/'])
        self.include_files.extend([include_file])
        self.dual_exists = 0
        # self.constructor_call = \
        #     'experiment<' + element_type + ', ' + set_type + '>(' + str(n) + ')'
        # self.time_unit = 'ns'
        # self.driver_file = self.generate_cpu_time_driver()
        self.driver_file = 'drive.c++'

    def output(self):
        if self.driver_output != "":
            return (self.n, self.driver_output)

```

```

        else:
            return ()

# def tidy_up(self):
#     "Used for debugging to see the files generated"
#     pass

class curve(benz.curve_suite):
    def __init__(self, element_type, set_type, include_file, macro, log_i, log_k):
        benz.curve_suite.__init__(self)
        self.title = set_type
        for j in range(log_i, log_k + 1):
            self.add(case(1 << j, element_type, set_type, include_file, macro))

class plot(benz.plot_suite):
    def __init__(self, element_type, log_i, log_k):
        benz.plot_suite.__init__(self)
        self.title = 'Efficiency of set operations: sort model \
(insert^nerase^n), integer data'
        self.xlabel = 'n'
        self.ylabel = 'Execution time per element [in nanoseconds]'
        x = (1 << log_i) - 10
        y = (1 << log_k) + 10
        self.gnuplot_commands += 'set key left top Left reverse samples 4 \
spacing 1.25 title "" \n'
        self.gnuplot_commands += 'set xrange [' + str(x) + ':' + str(y) + ']'
        self.gnuplot_commands += ""
        set terminal postscript enhanced
        set title '%(title)s'
        set xlabel '%(xlabel)s'
        set ylabel '%(ylabel)s'
        set logscale x
        """ % self.__dict__
        self.add(curve(element_type, 'std::set<' + element_type + '>',
            '<set>', 'STD', log_i, log_k))
        self.add(curve(element_type, 'cphstl::<' + element_type + '>',
            './Program/set.h++', 'US', log_i, log_k))

if __name__ == '__main__':
    benz.main(
        task = plot('int', 20, 20),
        # task = plot('int', 10, 23),
        runner = benz.gnuplot_runner
    )

```

1.2 Benchmark/experiment.cpp

```

#include <vector> // defines std::vector
#include <functional> // defines std::less
#include <cassert> // defines assert macro
#include <algorithm> // defines std::equal
#include <iostream> // defines std streams

template <typename V, typename I>
void decreasing_sequence(I p, I q) {
    int i = q - p;
    while (p ≠ q) {
        *p = V(i);
        --i;
        ++p;
    }
}

template <typename I>
void show(I p, I q) {

```

```

    while (p ≠ q) {
        std::cout << *p << std::endl;
        ++p;
    }
}

template <typename V, typename S>
class experiment {
public:
    experiment(unsigned int n) : n(n) {
        a = std::vector<V>(n, V(n));
        decreasing_sequence<int>(a.begin(), a.end());
        less = std::less<V>();
        b = std::vector<V>(n, V());
    }
    void primal() {
        S t;
        volatile unsigned long s = 0;
        while (s < n) {
            t.insert(a[s]);
            s = s + 1;
        }
        s = n;
        while (s > 0) {
            --s;
            b[s] = *t.begin();
            t.erase(t.begin());
        }
        assert(a.size() ≡ b.size());
        assert(std::equal(a.begin(), a.end(), b.begin()));
    }
private:
    unsigned long n;
    std::vector<V> a;
    std::vector<V> b;
    std::less<V> less;
};

```

1.3 Benchmark/drive.c++

```

#include <iostream> // defines std::cout and std::endl
#include <ctime> // defines std::clock
#include "experiment.c++" // defines experiment

#ifndef NUMBER_OF_ELEMENTS
#define NUMBER_OF_ELEMENTS 1000000
#endif

#define ELEMENT_TYPE unsigned int

#ifdef STD
#include <set> // defines std::set
#define SET_TYPE std::set<ELEMENT_TYPE>
#elif defined(US)
#include "../Program/set.h++" // defines cphstl::set
#define SET_TYPE cphstl::set<ELEMENT_TYPE>
#endif

int main() {
    experiment<ELEMENT_TYPE, SET_TYPE> e(NUMBER_OF_ELEMENTS);
    clock_t primal_start = clock();
    e.primal();
    clock_t primal_ticks = clock() - primal_start;
    double ns = 1.0e9 * double(primal_ticks) / double(CLOCKS_PER_SEC);
    std::cout << ns / double(NUMBER_OF_ELEMENTS) << std::endl;
}

```

```
    return 0;
}
```

J. UNIX makefiles

J.1 Program/makefile

```
default: unittests regression-test
options = -Wall -pedantic -ansi -x c++

unittests: iterator node red_black_tree

iterator:
    g++ -DUNITTEST_BIDIRECTIONAL_ITERATOR $(options) bidirectional_iterator.h++
    ./a.out

node:
    g++ -DUNITTEST_TREE_NODE $(options) node.h++
    ./a.out

red_black_tree:
    g++ -DUNITTEST_RED_BLACK_TREE $(options) red_black_tree.h++
    ./a.out

regression-test:
    g++ $(options) regression-test.c++
    ./a.out
    @make clean

clean:
    rm -f a.out *~ temp
```

J.2 Benchmark/makefile

```
CXX = g++ #$(HOME)/Script/Gfilt/gfilt
options = -Wall -pedantic -ansi -O3 -x c++

testdrive: std us

std:
    $(CXX) $(options) -DSTD drive.c++
    ./a.out

us:
    $(CXX) $(options) -DUS drive.c++
    ./a.out

benz:
    python sort_int_pentium.py

clean:
    -rm -rf *.pyc *~ a.out plot.[0-9]* driver.[0-9]* debug.[0-9]*
```

References

- [1] G. M. Adel'son-Vel'skiĭ and E. M. Landis, An algorithm for the organization of information, *Soviet Mathematics* **3**, 5 (1962), 1259–1263.
- [2] A. Andersson, Optimal bounds on the dictionary problem, *Proceedings of the 2nd International Symposium on Optimal Algorithms, Lecture Notes in Computer Science* **401**, Springer-Verlag (1989), 106–114.

- [3] A. Andersson, Balanced search trees made simple, *Proceedings of the 3rd Workshop on Algorithms and Data Structures, Lecture Notes in Computer Science* **709**, Springer-Verlag (1993), 60–71.
- [4] M. A. Bender, R. Cole, and R. Raman, Exponential structures for efficient cache-oblivious algorithms, *Proceedings of the 29th International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science* **2380**, Springer-Verlag (2002), 195–207.
- [5] British Standards Institute, *The C++ Standard: Incorporating Technical Corrigendum 1*, 2nd Edition, John Wiley and Sons, Ltd. (2003).
- [6] H. Brönnimann and J. Katajainen, Efficiency of various forms of red-black trees, CPH STL Report 2006-2, Department of Computing, University of Copenhagen (2006).
- [7] H. Brönnimann, J. Katajainen, and P. Morin, Putting your data structure on a diet, CPH STL Report 2007-1, Department of Computing, University of Copenhagen (2007).
- [8] S. Cho and S. Sahni, A new weight balanced binary search tree, *International Journal of Foundations of Computer Science* **11**, 3 (2000), 485–513.
- [9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd Edition, The MIT Press (2001).
- [10] Department of Computing, University of Copenhagen, The CPH STL, Website accessible at <http://www.cphstl.dk/> (2000–2007).
- [11] R. Fleischer, A simple balanced search tree with $O(1)$ worst-case update time, *Proceedings of the 4th International Symposium on Algorithms and Computation, Lecture Notes in Computer Science* **762**, Springer-Verlag (1993), 138–146.
- [12] L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts, A new representation for linear lists, *Proceedings of the 9th Annual ACM Symposium on the Theory of Computing*, ACM (1977), 49–60.
- [13] J. G. Hansen and A. K. Henriksen, The (multi)?(map|set) of the Copenhagen STL, CPH STL Report 2001-6, Department of Computing, University of Copenhagen (2001).
- [14] S. Huddleston and K. Mehlhorn, A new data structure for representing sorted lists, *Acta Informatica* **17**, 2 (1982), 157–184.
- [15] J. Katajainen, Making operations on standard-library containers strongly exception safe, ([Ask Jyrki for details].) (2006)
- [16] C. Levcopoulos and M. H. Overmars, A balanced search tree with $O(1)$ worst-case update time, *Acta Informatica* **26**, 3 (1988), 269–277.
- [17] S. Lynge, Implementing the AVL-trees for the CPH STL, CPH STL Report 2004-1, Department of Computing, University of Copenhagen (2004).
- [18] D. Maier and S. C. Salveter, Hysterical B-trees, *Information Processing Letters* **12**, 4 (1981), 199–202.
- [19] H. A. Maurer, T. Ottmann, and H.-W. Six, Implementing dictionaries using binary trees of very small height, *Information Processing Letters* **5**, 1 (1976), 11–14.
- [20] K. Mehlhorn, *Sorting and Searching, Data Structures and Algorithms* **1**, Springer-Verlag (1984).
- [21] J. I. Munro, T. Papadakis, and R. Sedgewick, Deterministic skip lists, *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, ACM-SIAM (1992), 367–375.
- [22] J. Nievergelt and E. M. Reingold, Binary search trees of bounded balance, *SIAM Journal on Computing* **2**, 1 (1973), 33–43.
- [23] P. J. Plauger, A. A. Stepanov, M. Lee, , and D. R. Musser, *The C++ Standard Template Library*, Prentice Hall PTR (2001).
- [24] S. Roura, A new method for balancing binary search trees, *Proceedings of the 28th International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science* **2076**, Springer-Verlag (2001), 469–480.
- [25] Silicon Graphics, Inc., Standard template library programmer’s guide, Website accessible at <http://www.sgi.com/tech/stl/> (1993–2006).
- [26] B. Stroustrup, *The C++ Programming Language*, Special Edition, Addison Wesley

- Longman, Inc. (2000).
- [27] D. Vandevorde and N. M. Josuttis, *C++ Templates: The Complete Guide*, Addison-Wesley (2003).
 - [28] M. A. Weiss, *Data Structures and Problem Solving using Java*, Addison Wesley Longman, Inc. (1998).
 - [29] N. Wirth, *Algorithms and Data Structures*, Prentice/Hall International, Inc. (1986).