Project proposal: A meldable, iterator-valid priority queue

Jyrki Katajainen

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

Abstract. The Standard Template Library (STL) is a library of generic algorithms and data structures that has been incorporated in the C++ standard and ships with all modern C++ compilers. In the CPH STL project the goal is to implement an enhanced edition of the STL. The priority-queue class of the STL is just an adapter that makes any resizable array to a queue in which the elements stored are arranged according to a given ordering function. In the C++ standard no compulsory support for the operations **delete**(), **increase**(), or **meld**() is demanded even if those are utilized in many algorithms solving graph-theoretic or geometric problems. In this project, the goal is to implement a CPH STL extension of the priority-queue class which provides, in addition to the normal priority-queue functionality, the operations **delete**(), **increase**(), and **meld**(). To make the first two of these operations possible, the class must also guarantee that external references to compartments inside the data structure are kept valid at all times.

Keywords. Data structures, priority queues, heaps

Problem formulation

In a generic form, a priority queue is a data structure which depends on six type parameters:

- $\mathcal{V}$: the type of the *elements* (or *values*) manipulated;
- $\mathcal{C}$: the type of the *comparator* which is a function object used in element comparisons;
- $\mathcal{N}$: the type of the *compartments* (or *nodes*) used for storing the elements and satellite data like pointers to other compartments;
- $\mathcal{I}$: the type of the *iterators* used for referring to the compartments;
- $\mathcal{A}$: the type of the *allocator* which provides an interface to allocate, construct, destroy, and deallocate objects; and
- $\mathcal{S}$: the type of the *concrete data structure* (or *container*) used for storing the compartments.

A *meldable priority queue* Q with type parameters $\mathcal{V}, \mathcal{C}, \mathcal{N}, \mathcal{I}, \mathcal{A}, \mathcal{S}$ should provide the following member functions:

CPH STL Report 2005-1, March 2005. Revised May 2006.

$\mathcal{I}$ **find-max():** Return an iterator to the compartment which stores an element that, of all elements in Q, has the maximum value. The ordering criterion used is $\mathcal{C}$. If Q is empty, an iterator referring to the one-past-the-end element is returned.

$\mathcal{I}$ **insert(**$\mathcal{V}$ e**):** Insert element e into Q and return an iterator to the compartment storing e for later use. If there is not space available to accomplish this operation, an iterator referring to the one-past-the-end element is returned. **Requirement:** Element e is passed by reference and is not modified.

void **delete-max():** Remove a maximum element and its compartment from Q. **Requirement:** Q is not empty.

$\mathcal{I}$ **delete(**$\mathcal{I}$ p**):** Remove both the element stored at the compartment referred to by p and the compartment itself from Q. An iterator to the position one past the erased element is returned. **Requirement:** p refers to an existing compartment.

void **increase(**$\mathcal{I}$ p**,** $\mathcal{V}$ e**):** Replace the element stored at the compartment referred to by p with element e. **Requirement:** p refers to an existing compartment. e is no smaller than the old element stored at that compartment. Element e is passed by reference and is not modified.

void **meld(***meldable_priority_queue* R**):** Move all elements together with their compartments from R to Q thereby destroying R. **Requirement:** R is passed by reference.

Some additional member functions—like a constructor, a destructor, and a set of functions to be used for examining the number of elements stored in Q—are necessary to make the data structure useful.

A complete declaration of the priority-queue class is given in appendix A. In this declaration, the standard STL concepts are mixed with those often found in the algorithmic literature. The correspondence between the two sets of concepts is as follows:

Algorithmic concept	**STL concept**
find-max()	`top()`
insert()	`push()`
delete-max()	`pop()`
delete()	`erase()`
pointer to a compartment	iterator

The meaning of the member types and member functions declared should be self-explanatory. Further details can be found from the C++ standard [1, Clause 23], or any other source on the STL.

In the CPH STL [6], the current implementation (see appendix C) of the priority-queue class is based on binary heaps [21]. In this project, the goal is to develop competitors for binary heaps, and to compare the efficiency of these new priority-queue realizations against the binary-heap implementation. The challenges lie in the realization of the member functions **insert**()

and **increase**() having logarithmic worst-case cost for binary heaps, and **meld**() having polylogarithmic worst-case cost for (pointer-based) binary heaps [17, 18, 19]. All these three operations can be realized faster using other meldable priority-queue structures.

Any of the priority-queue structures presented in the algorithmic literature (see, for example, [5, 8] and the references mentioned therein) can be used as the basis of the implementation work. Especially noteworthy alternatives to binary heaps are the following data structures, but other data structures may be considered as well:

Data structure	**Reference**
Binomial queues	[20] (see also [5, Chapter 19])
Brodal's heaps	[2]
Fat heaps	[13, 14]
Leftist trees	[4] (see also [15, Section 5.2.3])
Maxiphobic heaps	[16]
Multiary heaps	[12]
Navigation piles	[11]
Relaxed weak queues	[9]
Run-relaxed heaps	[7, 10]
Space-efficient heaps	[3]

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

Let n denote the number of elements stored or to be stored in the data structure. The resulting implementations should fulfil the following complexity requirements given in the C++ standard:

find-max(): constant time
insert(): at most $\lg n$ element comparisons
delete(): at most $2 \lg n$ element comparisons
general constructor: at most $3n$ element comparisons
iterator functions: constant time (amortized).

Naturally, the resulting implementations should guarantee *iterator validity*. That is, when an element is inserted, an iterator referring to the compartment, where it is stored, should remain the same so that possible later references made by **delete**() and **increase**() are valid. The strength of the iterators supported should be bidirectional.

Appendix: Source code

The source code given in this appendix can be used as an inspiration for the project. This appedix describes the bridge class `meldable_priority_queue` and gives a single realization of that class based on binary heaps. All the files described, including the associated scripts, are available for download on the CPH STL website [6] (see menu item Downloads).

A. Meldable priority queues 4
A.1 `Program/meldable_priority_queue.h++` 4
A.2 `Program/meldable_priority_queue.c++` 6
B. Example 9
B.1 A pointer-based binary heap storing 10 integers 9
C. Binary heaps 10
C.1 `Program/binary_heap.h++` 10
C.2 `Program/binary_heap.c++` 13
D. Heap nodes 20
D.1 `Program/heap_node.h++` 20
E. Property maps 22
E.1 `Program/property_maps.h++` 22
F. Bidirectional iterators 26
F.1 `Program/bidirectional_iterator.h++` 26
G. Skew binary numbers 30
G.1 `Program/skew_binary_number.h++` 30
H. Benz forms 33
H.1 `Benchmark/sort_int_pentium.py` 33
H.2 `Benchmark/experiment.c++` 34
H.3 `Benchmark/drive.c++` 34
I. UNIX makefiles 35
I.1 `Program/makefile` 35
I.2 `Benchmark/makefile` 35

A. Meldable priority queues

A.1 `Program/meldable_priority_queue.h++`

```
/*
Author: Jyrki Katajainen, March 2005; Revised May 2006

A meldable priority queue is a reversible container (as a set and
multiset) which provides constant bidirectional iterators to the
elements stored.

CPH STL guarantees:

- Member function push() returns an iterator (or a handle) to the
  given element and this iterator remains valid the whole life time of
  the element.

- Iterator operations take constant time in the worst case.

- Member function top() is a constant-time operation.

- Member functions push(), pop(), erase(), and increase() have the
  logarithmic worst-case cost.

- Member function meld() is relatively fast having a polylogarithmic
  worst-case cost.
```

```
- The data structure uses a linear number of words in addition to the
  elements stored.

Container requirements not fulfilled [C++ standard §23]:

- For two meldable priority queues a and b, the following expressions
  are not supported: a ≡ b, a ≢ b, a < b, a > b, a ≤ b, and a ≥ b.
*/

#ifndef __CPHSTL_MELDABLE_PRIORITY_QUEUE__
#define __CPHSTL_MELDABLE_PRIORITY_QUEUE__

#include <functional> // defines std::less
#include <iterator> // defines std::reverse_iterator
#include <memory> // defines std::allocator
#include "binary_heap.h++" // defines cphstl::binary_heap

namespace cphstl {

  template <
    typename V,
    typename C = std::less<V>,
    typename A = std::allocator<V>,
    typename S = cphstl::binary_heap<V, C, A>
  >
  class meldable_priority_queue {
  public:
    // types

    typedef typename S::size_type size_type;
    typedef typename S::difference_type difference_type;
    typedef typename S::pointer pointer;
    typedef typename S::const_pointer const_pointer;
    typedef typename S::reference reference;
    typedef typename S::const_reference const_reference;
    typedef typename S::value_type value_type;
    typedef typename S::iterator iterator;
    typedef typename S::const_iterator const_iterator;
    typedef std::reverse_iterator<iterator> reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
    typedef C comparator_type;
    typedef typename S::allocator_type allocator_type
    typedef S container_type;

  protected:
    container_type container_;

  public:
    // structors

    explicit meldable_priority_queue(const C& = C(), const A& = A());
    meldable_priority_queue(const meldable_priority_queue&);
    meldable_priority_queue& operator=(const meldable_priority_queue&);
    ~meldable_priority_queue();

    // observers

    allocator_type get_allocator() const;
    comparator_type get_comparator() 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;

    // access

    iterator top() const;

    // modifiers

    iterator push(const value_type&);
    void pop();
    iterator erase(iterator);
    void increase(iterator, const value_type&);
    void clear();
    void meld(meldable_priority_queue&);
    void swap(meldable_priority_queue&);
  };

  // algorithms

  template <typename V, typename C, typename A, typename S>
  meldable_priority_queue<V, C, A, S>&
  meld(meldable_priority_queue<V, C, A, S>&,
       meldable_priority_queue<V, C, A, S>&);

  template <typename V, typename C, typename A, typename S>
  void swap(meldable_priority_queue<V, C, A, S>&,
            meldable_priority_queue<V, C, A, S>&);

} // namespace cphstl

#include "meldable_priority_queue.c++" // defines cphstl::meldable_priority_queue
#endif // __CPHSTL_MELDABLE_PRIORITY_QUEUE__
```

A.2 Program/meldable_priority_queue.c++

```
/*
Author: Jyrki Katajainen, May 2006

A meldable priority queue is a bridge class that just calls the
functions available in the implementation class.
*/

namespace cphstl {

  template <typename V, typename C, typename A, typename S>
  meldable_priority_queue<V, C, A, S>::meldable_priority_queue (
    const C& comparator, const A& allocator) : container_(comparator, allocator) {
  }

  template <typename V, typename C, typename A, typename S>
  meldable_priority_queue<V, C, A, S>::meldable_priority_queue(
    const meldable_priority_queue& other) : container_(other.container_) {
  };
```

```
template <typename V, typename C, typename A, typename S>
meldable_priority_queue<V, C, A, S>::meldable_priority_queue&
meldable_priority_queue<V, C, A, S>::operator=(
  const meldable_priority_queue& other) {
  container_.operator=(other.container_);
};

template <typename V, typename C, typename A, typename S>
meldable_priority_queue<V, C, A, S>::~meldable_priority_queue() {
};

template <typename V, typename C, typename A, typename S>
typename meldable_priority_queue<V, C, A, S>::allocator_type
meldable_priority_queue<V, C, A, S>::get_allocator() const {
  return container_.get_allocator();
}

template <typename V, typename C, typename A, typename S>
typename meldable_priority_queue<V, C, A, S>::comparator_type
meldable_priority_queue<V, C, A, S>::get_comparator() const {
  return container_.get_comparator();
};

template <typename V, typename C, typename A, typename S>
typename meldable_priority_queue<V, C, A, S>::iterator
meldable_priority_queue<V, C, A, S>::begin() {
  return container_.begin();
};

template <typename V, typename C, typename A, typename S>
typename meldable_priority_queue<V, C, A, S>::const_iterator
meldable_priority_queue<V, C, A, S>::begin() const {
  return const_iterator(container_.begin());
};

template <typename V, typename C, typename A, typename S>
typename meldable_priority_queue<V, C, A, S>::iterator
meldable_priority_queue<V, C, A, S>::end() {
  return container_.end();
};

template <typename V, typename C, typename A, typename S>
typename meldable_priority_queue<V, C, A, S>::const_iterator
meldable_priority_queue<V, C, A, S>::end() const {
  return const_iterator(container_.end());
};

template <typename V, typename C, typename A, typename S>
typename meldable_priority_queue<V, C, A, S>::reverse_iterator
meldable_priority_queue<V, C, A, S>::rbegin() {
  return reverse_iterator(end());
};

template <typename V, typename C, typename A, typename S>
typename meldable_priority_queue<V, C, A, S>::const_reverse_iterator
meldable_priority_queue<V, C, A, S>::rbegin() const {
  return const_reverse_iterator(end());
};

template <typename V, typename C, typename A, typename S>
typename meldable_priority_queue<V, C, A, S>::reverse_iterator
meldable_priority_queue<V, C, A, S>::rend() {
  return reverse_iterator(begin());
};
```

```
template <typename V, typename C, typename A, typename S>
typename meldable_priority_queue<V, C, A, S>::const_reverse_iterator
meldable_priority_queue<V, C, A, S>::rend() const {
  return const_reverse_iterator(begin());
};

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

template <typename V, typename C, typename A, typename S>
typename meldable_priority_queue<V, C, A, S>::size_type
meldable_priority_queue<V, C, A, S>::size() const {
  return container_.size();
};

template <typename V, typename C, typename A, typename S>
typename meldable_priority_queue<V, C, A, S>::size_type
meldable_priority_queue<V, C, A, S>::max_size() const {
  return container_.max_size();
};

template <typename V, typename C, typename A, typename S>
typename meldable_priority_queue<V, C, A, S>::iterator
meldable_priority_queue<V, C, A, S>::top() const {
  return container_.top();
};

template <typename V, typename C, typename A, typename S>
typename meldable_priority_queue<V, C, A, S>::iterator
meldable_priority_queue<V, C, A, S>::push(const value_type& e) {
  return container_.push(e);
};

template <typename V, typename C, typename A, typename S>
void
meldable_priority_queue<V, C, A, S>::pop() {
  (void)(*this).erase((*this).top());
};

template <typename V, typename C, typename A, typename S>
typename meldable_priority_queue<V, C, A, S>::iterator
meldable_priority_queue<V, C, A, S>::erase(iterator p) {
  return container_.erase(p);
};

template <typename V, typename C, typename A, typename S>
void
meldable_priority_queue<V, C, A, S>::increase(
  iterator p, const value_type& e) {
  container_.increase(p, e);
};

template <typename V, typename C, typename A, typename S>
void
meldable_priority_queue<V, C, A, S>::clear() {
  container_.clear();
};

template <typename V, typename C, typename A, typename S>
void
meldable_priority_queue<V, C, A, S>::meld(meldable_priority_queue& r) {
  container_.meld(r);
```

```
  };

  template <typename V, typename C, typename A, typename S>
  void
  meldable_priority_queue<V, C, A, S>::swap(meldable_priority_queue& r) {
    container_.swap(r);
  };

  template <typename V, typename C, typename A, typename S>
  meldable_priority_queue<V, C, A, S>&
  meld(meldable_priority_queue<V, C, A, S>& r,
       meldable_priority_queue<V, C, A, S>& s) {
    r.meld(s);
    return r;
  };

  template <typename V, typename C, typename A, typename S>
  void swap(meldable_priority_queue<V, C, A, S>& r,
            meldable_priority_queue<V, C, A, S>& s) {
    r.swap(s);
  };

} // namespace cphstl
```

B. Example

B.1 A pointer-based binary heap storing 10 integers

In addition to normal nodes, there are four different kinds of dummies:

1. a unique *sentinel* to which other nodes point if some of their neighbouring nodes are missing;
2. *heads* that keep track of the roots of the perfect binary heaps;
3. *tails* that are the last nodes in the lists tying the nodes of a perfect binary heap together; and

4. the *past-the-end head* that is the last node in the list tying the heads together.

C. Binary heaps

C.1 Program/binary_heap.h++

```
/*
Author: Jyrki Katajainen, May 2006

A priority queue implemented as a collection of pointer-based perfect
binary heaps (in the same way as a binomial queue is implemented a
collection of binomial trees). The idea of using the canonical
representation of skew-binary numbers is borrowed from [Eugene
W. Myers. An applicative random-access stack. Information Processing
Letters 17 (1983), 241-248]. 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)].

CPH STL guarantees:

- Let n be the number of elements stored prior to each operation.
  Member functions guarantee the following worst-case complexity
  bounds:

  push(): O(lg n) [O(1) amortized]
  erase(), increase(): O(lg n)
  meld(): O(max{lg m, lg n}^2) [m and n are the sizes of the queues melded]
  clear(): O(n)
  others: O(1)

- The amount of extra space used is 4n + O(lg n) words.
*/

#ifndef __CPHSTL_BINARY_HEAP__
#define __CPHSTL_BINARY_HEAP__

#include <functional> // defines std::less
#include <memory> // defines std::allocator
#include "heap_node.h++" // defines the nodes used
#include "property_maps.h++" // defines the property maps used
#include "bidirectional_iterator.h++" // defines cphstl:bidirectional_iterator
#include "skew_binary_number.h++" // defines cphstl::skew_binary_number

namespace cphstl {
  namespace heap {

    template <typename C>
    class comparator_management {
    protected:
      typedef C comparator_type;

      comparator_management(const comparator_type& c) : comparator_(c) {
      }

      comparator_type comparator_;
    };

    template <typename V, typename C, typename A, typename N>
    class node_management : public cphstl::heap::comparator_management<C> {
    protected:
```

```
      typedef V value_type;
      typedef C comparator_type;
      typedef typename A::template rebind<value_type>::other allocator_type;
      typedef N node_type;
      typedef typename A::template rebind<node_type>::other::pointer link_type;

      node_management(const comparator_type& c, const allocator_type& a)
        : comparator_management<C>(c), node_allocator_(a) {
      }

      typename allocator_type::template rebind<node_type>::other node_allocator_;
    };

    template <typename V, typename C, typename A, typename N>
    class link_management : public cphstl::heap::node_management<V, C, A, N>
{
    protected:
      typedef V value_type;
      typedef C comparator_type;
      typedef typename A::template rebind<value_type>::other allocator_type;
      typedef N node_type;
      typedef typename A::template rebind<node_type>::other::pointer link_type;

      link_management(const comparator_type& c, const allocator_type& a)
        : node_management<V, C, A, N>(c, a), link_allocator_(a) {
      }

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

    template <typename V, typename C, typename A, typename N>
    class value_management : public cphstl::heap::link_management<V, C, A, N> {
    protected:
      typedef V value_type;
      typedef C comparator_type;
      typedef typename A::template rebind<value_type>::other allocator_type;
      typedef N node_type;

      value_management(const comparator_type& c, const allocator_type& a)
        : link_management<V, C, A, N>(c, a), value_allocator_(a) {
      }
      allocator_type value_allocator_;
    };

  } // namespace heap

  template <
    typename V,
    typename C = std::less<V>,
    typename A = std::allocator<V>,
    typename N = cphstl::heap::node<V>
  >
  class binary_heap : public cphstl::heap::value_management<V, C, A, N> {
  public:
    // types

    typedef V value_type;
    typedef C comparator_type;
    typedef typename A::template rebind<value_type>::other allocator_type;
    typedef N node_type;
    typedef binary_heap<V, C, A, N> container_type;
    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;
```

```
    typedef typename allocator_type::reference reference;
    typedef typename allocator_type::const_reference const_reference;
    typedef cphstl::bidirectional_iterator<
      cphstl::heap::value_map<value_type, node_type>,
      cphstl::heap::successor_map<node_type>,
      cphstl::heap::predecessor_map<node_type>, true > const_iterator;
    typedef const_iterator iterator;

  protected:
    typedef cphstl::heap::comparator_management<C> comparator_management;
    typedef cphstl::heap::node_management<V, C, A, N> node_management;
    typedef cphstl::heap::link_management<V, C, A, N> link_management;
    typedef cphstl::heap::value_management<V, C, A, N> value_management;

    cphstl::skew_binary_number<size_type> n_;
    node_type* past_the_end_;
    node_type* top_;

  public:
    // structors

    explicit binary_heap(const comparator_type& c = comparator_type(),
                         const allocator_type& a = allocator_type())
      : value_management(c, a), n_() {
      (*this).init();
    };
    binary_heap(const binary_heap&);
    binary_heap& operator=(const binary_heap&);
    ~binary_heap();

    // observers

    allocator_type get_allocator() const;
    comparator_type get_comparator() const;

    // iterators

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

    // capacity

    size_type size() const;
    size_type max_size() const;

    // access

    iterator top() const;

    // modifiers

    iterator push(const value_type&);
    iterator erase(iterator);
    void increase(iterator, const value_type&);
    void clear();
    void meld(binary_heap&);
    void swap(binary_heap&);

  protected:
    // helpers

    void init();
    node_type* create_dummy();
    void construct_value(node_type*, const value_type&);
    void destruct_value(node_type*);
```

```
    void discard_dummy(node_type*);
    void swap_parent_left(node_type*, node_type*);
    void swap_parent_right(node_type*, node_type*);
    void head_list_erase(node_type*);
    void head_list_insert(node_type*, node_type*);
    node_type* borrow();
    void join(node_type*, node_type*, node_type*);
    void sew_up_inner_edges(node_type*, node_type*);
    node_type* sew_up_outer_edges(node_type*, node_type*);
    void siftdown(node_type*);
    void siftup(node_type*);
    template <typename I>
    void bulk_insert(I, I);
  };

} // namespace cphstl

#include "binary_heap.c++" // defines cphstl::binary_heap
#endif // __CPHSTL_BINARY_HEAP__
```

C.2 Program/binary_heap.c++

```
/*
Author: Jyrki Katajainen, May 2006

In each perfect binary heap the nodes are maintained in a
doubly-linked list. The list ends with a dummy node called a
tail. Each node of a heap has a pointer to its left child and its
parent. The right child is accessed via the left child. The root of
each heap is connected with a dummy called a head. The heads are again
maintained in a doubly-linked list, and the heap is accessed via the
first head which is also used as the past-the-end node.
*/

#include <cmath> // defines ilogb
#include <iostream>
#include <cassert>

extern int ilogb(double) throw();

namespace cphstl {

//template <typename V, typename C, typename A, typename N>
//binary_heap<V, C, A, N>::binary_heap (const C& c, const A& a) {
//  Moved to the declaration because of an error in gcc version 3.3.4
//}

  template <typename V, typename C, typename A, typename N>
  binary_heap<V, C, A, N>::binary_heap(const binary_heap& other)
    : value_management(other.comparator_, other.value_allocator_), n_() {
    (*this).init();
    (*this).bulk_insert(other.begin(), other.end());
  }

  template <typename V, typename C, typename A, typename N>
  binary_heap<V, C, A, N>&
  binary_heap<V, C, A, N>::operator=(const binary_heap& other) {
    if (this ≢ &other) {
      (*this).clear();
      (*this).init();
      (*this).bulk_insert(other.begin(), other.end());
    }
    return *this;
  }
```

```
template <typename V, typename C, typename A, typename N>
binary_heap<V, C, A, N>::~binary_heap() {
  (*this).clear();
  (*this).discard_dummy(past_the_end_);
  n_ = cphstl::skew_binary_number<size_type>();
  past_the_end_ = 0;
  top_ = 0;
}

template <typename V, typename C, typename A, typename N>
typename binary_heap<V, C, A, N>::allocator_type
binary_heap<V, C, A, N>::get_allocator() const {
  return (*this).value_allocator_;
}

template <typename V, typename C, typename A, typename N>
typename binary_heap<V, C, A, N>::comparator_type
binary_heap<V, C, A, N>::get_comparator() const {
  return (*this).comparator_;
}

template <typename V, typename C, typename A, typename N>
typename binary_heap<V, C, A, N>::const_iterator
binary_heap<V, C, A, N>::begin() const {
  USE_HEAP_PROPERTY_MAPS(V, N)
  return const_iterator(south[east[past_the_end_]]);
}

template <typename V, typename C, typename A, typename N>
typename binary_heap<V, C, A, N>::const_iterator
binary_heap<V, C, A, N>::end() const {
  return const_iterator(past_the_end_);
}

template <typename V, typename C, typename A, typename N>
typename binary_heap<V, C, A, N>::size_type
binary_heap<V, C, A, N>::size() const {
  return size_type(n_);
}

template <typename V, typename C, typename A, typename N>
typename binary_heap<V, C, A, N>::size_type
binary_heap<V, C, A, N>::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 heap
  size_type m = (*this).value_allocator_.max_size(); // max number of elements
  // solve n from (4n + 2lg n + 1)b + c = a
  return (((a - c)/b - 1)/2 - ilogb(m))/2;
}

template <typename V, typename C, typename A, typename N>
typename binary_heap<V, C, A, N>::iterator
binary_heap<V, C, A, N>::top() const {
  return iterator(top_);
}

template <typename V, typename C, typename A, typename N>
typename binary_heap<V, C, A, N>::iterator
binary_heap<V, C, A, N>::push(const value_type& v) {
  USE_HEAP_PROPERTY_MAPS(V, N)
  node_type* r = 0;
  if (n_.has_a_two()) {
    try {
```

```
      r = create_dummy();
      construct_value(r, v);
    }
    catch (...) {
      discard_dummy(r);
      throw;
    }
    node_type* a = east[past_the_end_];
    node_type* b = east[a];
    node_type* p = south[a];
    node_type* q = south[b];
    head_list_erase(b);
    north[r] = a;
    south[a] = r;
    join(p, q, r);
  }
  else {
    node_type* h = 0;
    node_type* t = 0;
    try {
      r = create_dummy();
      construct_value(r, v);
      h = create_dummy();
      t = create_dummy();
    }
    catch (...) {
      discard_dummy(t);
      discard_dummy(h);
      destruct_value(r);
      discard_dummy(r);
      throw;
    }
    north[h] = h;
    head_list_insert(h, past_the_end_);
    south[h] = r;
    north[r] = h; east[r] = t; west[r] = t;
    east[t] = r; west[t] = r;
    assert(south[r] ≡ (*r).nil);
  }
  ++n_;
  if (top_ ≡ past_the_end_ || (*this).comparator_(value[top_], value[r])) {
    top_ = r;
  }
  return iterator(r);
}

template <typename V, typename C, typename A, typename N>
typename binary_heap<V, C, A, N>::iterator
binary_heap<V, C, A, N>::erase(iterator p) {
  // not implemented
  --n_;
  return past_the_end_;
}

template <typename V, typename C, typename A, typename N>
void
binary_heap<V, C, A, N>::increase(iterator p, const value_type& v) {
  // not implemented
}

template <typename V, typename C, typename A, typename N>
void
binary_heap<V, C, A, N>::clear() {
  // not implemented
}
```

```
template <typename V, typename C, typename A, typename N>
void
binary_heap<V, C, A, N>::meld(binary_heap& other) {
  // not implemented
}

template <typename V, typename C, typename A, typename N>
void
binary_heap<V, C, A, N>::swap(binary_heap& other) {
  // not implemented
}

// helpers

template <typename V, typename C, typename A, typename N>
void
binary_heap<V, C, A, N>::init() {
  USE_HEAP_PROPERTY_MAPS(V, N)
  past_the_end_ = create_dummy();
  north[past_the_end_] = past_the_end_;
  east[past_the_end_] = past_the_end_;
  south[past_the_end_] = past_the_end_;
  west[past_the_end_] = past_the_end_;
  top_ = past_the_end_;
}

template <typename V, typename C, typename A, typename N>
typename binary_heap<V, C, A, N>::node_type*
binary_heap<V, C, A, N>::create_dummy() {
  node_type* p = (*this).node_allocator_.allocate(1);
  (*this).link_allocator_.construct(&(*p).north, &(*p).sentinel);
  (*this).link_allocator_.construct(&(*p).east, &(*p).sentinel);
  (*this).link_allocator_.construct(&(*p).south, &(*p).sentinel);
  (*this).link_allocator_.construct(&(*p).west, &(*p).sentinel);
  return p;
}

template <typename V, typename C, typename A, typename N>
void
binary_heap<V, C, A, N>::construct_value(node_type* p, const value_type& v) {
  (*this).value_allocator_.construct(&(*p).value, v);
}

template <typename V, typename C, typename A, typename N>
void
binary_heap<V, C, A, N>::destruct_value(node_type* p) {
  if (p ≢ 0) {
    (*this).value_allocator_.destroy(&(*p).value);
  }
}

template <typename V, typename C, typename A, typename N>
void
binary_heap<V, C, A, N>::discard_dummy(node_type* p) {
  if (p ≢ 0) {
    (*this).link_allocator_.destroy(&(*p).west);
    (*this).link_allocator_.destroy(&(*p).south);
    (*this).link_allocator_.destroy(&(*p).east);
    (*this).link_allocator_.destroy(&(*p).north);
    (*this).node_allocator_.deallocate(p, 1);
  }
}

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

```
void
binary_heap<V, C, A, N>::swap_parent_left(node_type* x, node_type* y) {
  USE_HEAP_PROPERTY_MAPS(V, N)
  node_type* x_n = north[x]; node_type* x_e = east[x];
  node_type* x_w = west[x];
  node_type* y_e = east[y]; node_type* y_s = south[y];
  node_type* y_w = west[y];
  if (x_e ≡ y) {
    south[x_n] = y; east[x_w] = y;
    west[y_e] = x;
    if (!is_nil[y_s]) {
      north[y_s] = x;
    }
    north[y] = x_n; east[y] = x; south[y] = x; west[y] = x_w;
    north[x] = y; east[x] = y_e; south[x] = y_s; west[x] = y;
  }
  else {
    south[x_n] = y; east[x_w] = y; west[x_e] = y;
    east[y_w] = x; west[y_e] = x;
    if (!is_nil[y_s]) {
      north[y_s] = x;
    }
    north[y] = x_n; east[y] = x_e; south[y] = x; west[y] = x_w;
    north[x] = y; east[x] = y_e; south[x] = y_s; west[x] = y_w;
  }
}

template <typename V, typename C, typename A, typename N>
void
binary_heap<V, C, A, N>::swap_parent_right(node_type* x, node_type* y) {
  USE_HEAP_PROPERTY_MAPS(V, N)
  node_type* x_n = north[x]; node_type* x_e = east[x];
  node_type* x_w = west[x]; node_type* x_s = south[x];
  node_type* y_e = east[y]; node_type* y_s = south[y];
  node_type* y_w = west[y];
  south[x_n] = y; east[x_w] = y; north[x_s] = y; west[x_e] = y;
  east[y_w] = x; west[y_e] = x;
  if (!is_nil[y_s]) {
    north[y_s] = x;
  }
  north[y] = x_n; east[y] = x_e; south[y] = x_s; west[y] = x_w;
  north[x] = x; east[x] = y_e; south[x] = y_s; west[x] = y_w;
}

template <typename V, typename C, typename A, typename N>
void
binary_heap<V, C, A, N>::head_list_insert(node_type* b, node_type* a) {
  // Insert b after a
  USE_HEAP_PROPERTY_MAPS(V, N)
  node_type* c = east[a];
  east[a] = b;
  west[b] = a;
  east[b] = c;
  west[c] = b;
}

template <typename V, typename C, typename A, typename N>
void
binary_heap<V, C, A, N>::head_list_erase(node_type* b) {
  USE_HEAP_PROPERTY_MAPS(V, N)
  node_type* pred = west[b];
  node_type* succ = east[b];
  east[pred] = succ;
  west[succ] = pred;
  discard_dummy(b);
```

```
}

template <typename V, typename C, typename A, typename N>
typename binary_heap<V, C, A, N>::node_type*
binary_heap<V, C, A, N>::borrow() {
  // not implemented
}

template <typename V, typename C, typename A, typename N>
void
binary_heap<V, C, A, N>::join(node_type* p, node_type* q, node_type* r) {
  USE_HEAP_PROPERTY_MAPS(V, N)
  discard_dummy(west[p]);
  south[r] = p;
  north[p] = r;
  north[q] = q;
  sew_up_inner_edges(p, q);
  node_type* last = sew_up_outer_edges(p, r);
  last = east[last];
  west[r] = last;
  east[last] = r;
  siftdown(r);
}

template <typename V, typename C, typename A, typename N>
void
binary_heap<V, C, A, N>::sew_up_inner_edges(node_type* p, node_type* q) {
  USE_HEAP_PROPERTY_MAPS(V, N)
  while (!is_nil[p]) {
    east[p] = q;
    west[q] = p;
    p = right[p];
    q = left[q];
  }
}

template <typename V, typename C, typename A, typename N>
typename binary_heap<V, C, A, N>::node_type*
binary_heap<V, C, A, N>::sew_up_outer_edges(node_type* p, node_type* r) {
  USE_HEAP_PROPERTY_MAPS(V, N)
  while (!is_nil[p]) {
    west[p] = r;
    east[r] = p;
    p = left[p];
    r = right[r];
  }
  return r;
}

template <typename V, typename C, typename A, typename N>
void
binary_heap<V, C, A, N>::siftdown(node_type* r) {
  USE_HEAP_PROPERTY_MAPS(V, N)
  node_type* p = left[r];
  while (!is_nil[p]) {
    node_type* q = right[r];
    if ((*this).comparator_(value[p], value[q])) {
      if ((*this).comparator_(value[r], value[q])) {
        (*this).swap_parent_right(r, q);
        p = left[r];
      }
      else break;
    }
    else {
      if ((*this).comparator_(value[r], value[p])) {
```

```
          (*this).swap_parent_left(r, p);
          p = left[r];
        }
        else break;
      }
    }
  }

  template <typename V, typename C, typename A, typename N>
  void
  binary_heap<V, C, A, N>::siftup(node_type* p) {
    return p;
    // not implemented
  }

  template <typename V, typename C, typename A, typename N>
  template <typename I>
  void
  binary_heap<V, C, A, N>::bulk_insert(I first, I last) {
    // not implemented
  }

} // namespace cphstl

#undef USE_HEAP_PROPERTY_MAPS
#if defined(UNITTEST_BINARY_HEAP)
#define __main__ main

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

template <typename V>
class unittest_binary_heap {
public:

  void operator()() {
    typedef cphstl::binary_heap<V> heap;
    heap h;
    assert(h.size() ≡ 0);
    assert(h.begin() ≡ h.end());
    typename heap::allocator_type heap_a = h.get_allocator();
    std::allocator<V> std_a;
    assert(heap_a ≡ std_a);
    typename heap::comparator_type heap_c = h.get_comparator();
    assert(h.size() ≤ h.max_size());
    assert(h.top() ≡ h.end());
    V a[] = {8, 10, 13, 1, 6, 5, 3, 7, 11, 9};
    unsigned int n = sizeof(a)/sizeof(V);

    h.push(a[0]);
    assert(h.top() ≡ h.begin());
    typename heap::iterator t = h.top();
    assert(*t ≡ V(8));
    //*t = 12; /* Compilation should fail */
    for (typename heap::iterator p = h.begin(); p ≢ h.end();) {
      ++p;
    }
    for (typename heap::iterator p = h.end(); p ≢ h.begin(); ) {
      --p;
    }
    for (unsigned int j = 1; j < n; ++j) {
      h.push(a[j]);
      assert(*(h.top()) ≡ *(std::max_element(&a[0], &a[j + 1])));
```

```
      for (typename heap::iterator p = h.begin(); p ≢ h.end(); ) {
        ++p;
      }
      for (typename heap::iterator p = h.end(); p ≢ h.begin(); ) {
        --p;
      }
    }
    assert(h.size() ≡ n);

    V sum = V(0);
    for (typename heap::iterator p = h.begin(); p ≢ h.end(); ++p) {
      sum += *p;
    }
    assert(sum ≡ std::accumulate(&a[0], &a[n], V(0)));
  }
};

int __main__(int, char**) {
  unittest_binary_heap<int> t;
  t();
  unittest_binary_heap<float> u;
  u();
}

#undef __main__
#endif // UNITTEST_BINARY_HEAP
```

D. Heap nodes

D.1 Program/heap_node.h++

```
/*
Author: Jyrki Katajainen, May 2006

There are two types of nodes: dummies that do not carry any data and
normal nodes that carry some data.  A sentinel is a special node to
which other nodes point if some of their neighbouring nodes are
missing.
*/

#ifndef __CPHSTL_HEAP_NODE__
#define __CPHSTL_HEAP_NODE__

namespace cphstl {
  namespace heap {

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

      value_type value;
      node_type* north;
      node_type* east;
      node_type* south;
      node_type* west;

      static node_type* nil;
      static node_type sentinel;

      node() : north(&sentinel), east(&sentinel), south(&sentinel),
        west(&sentinel) {
```

```
      }

      node(const value_type& v) : value(v), north(&sentinel),
        east(&sentinel), south(&sentinel), west(&sentinel) {
      }
    };

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

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

  } // namespace heap
} // namespace cphstl

#if defined(UNITTEST_HEAP_NODE)
#define __main__ main

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

class unittest_heap_node {
public:

  void operator()() {
    cphstl::heap::node<int>* past_the_end = new cphstl::heap::node<int>();
    cphstl::heap::node<int>* head = new cphstl::heap::node<int>();
    cphstl::heap::node<int>* root = new cphstl::heap::node<int>(11);
    cphstl::heap::node<int>* p = new cphstl::heap::node<int>(7);
    cphstl::heap::node<int>* q = new cphstl::heap::node<int>(9);
    cphstl::heap::node<int>* tail = new cphstl::heap::node<int>();

    USE_HEAP_PROPERTY_MAPS(int, cphstl::heap::node<int>)
    north[past_the_end] = past_the_end; east[past_the_end] = head;
    south[past_the_end] = past_the_end; west[past_the_end] = head;
    north[head] = head; east[head] = past_the_end; south[head] = root;
    west[head] = past_the_end;
    north[root] = head; east[root] = p; south[root] = p; west[root] = tail;
    north[p] = root; east[p] = q; west[p] = root;
    north[q] = root; east[q] = tail; west[q] = p;
    east[tail] = root; west[tail] = q;

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

    iterator x; // default construtor
    iterator first(root); //parametrized constructor
    x = first; // operator= (generated automatically by the compiler)
    assert(*x == 11); // operator*
    iterator last(past_the_end); // copy constructor

    assert(*first == 11);
    ++first; // operator++
    assert(*first == 7);
    first++; // operator++(int)
    assert(*first == 9);
    --first; // operator--
    assert(*first == 7);
    first++; // operator++(int)
    first--; // operator--(int)
    assert(*first == 7);
```

```
    int count = 0;
    for (iterator i(root); i ≢ iterator(past_the_end); ++i) {
      ++count;
    }
    assert(count ≡ 3);

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

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

    assert(second ≡ other); //operator ≡
    ++second;
    assert(second ≢ other); //operator ≢

    delete tail;
    delete q;
    delete p;
    delete root;
    delete head;
    delete past_the_end;
  }
};

int __main__(int, char**) {
  unittest_heap_node t;
  t();
  return 0;
}

#undef __main__
#endif // UNITTEST_HEAP_NODE
#endif // __CPHSTL_HEAP_NODE__
```

E. Property maps

E.1 Program/property_maps.h++

```
/*
Author: Jyrki Katajainen, May 2006
*/

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

namespace cphstl {
  namespace heap {

    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 V, typename N>
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 north_map {
public:
  typedef N* domain_type;
  typedef N* range_type;

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

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

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

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

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

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

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

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::heap::south_map<N> south;
    return p ≡ south[p];
  }
};

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

  range_type operator[](domain_type p) const {
    cphstl::heap::is_nil_map<N> is_nil;
    cphstl::heap::north_map<N> north;
    return is_nil[north[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::heap::is_tail_map<N> is_tail;
    cphstl::heap::west_map<N> west;
    return is_tail[west[p]];
  }
};

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

  range_type& operator[](domain_type p) const {
    cphstl::heap::south_map<N> south;
    return south[p];
  }
};

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

  range_type operator[](domain_type p) const {
    cphstl::heap::south_map<N> south;
    cphstl::heap::east_map<N> east;
    return east[south[p]];
  }
```

```
    };

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

      range_type operator[](domain_type p) const {
        cphstl::heap::is_past_the_end_map<N> is_past_the_end;
        cphstl::heap::is_root_map<N> is_root;
        cphstl::heap::north_map<N> north;
        cphstl::heap::south_map<N> south;
        cphstl::heap::west_map<N> west;
        if (is_past_the_end[west[north[p]]]) {
          return p;
        }
        if (is_past_the_end[p] || is_root[p]) {
          return west[west[south[west[north[p]]]]];
        }
        return  west[p];
      }
    };

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

      range_type operator[](domain_type p) const {
        cphstl::heap::east_map<N> east;
        cphstl::heap::is_past_the_end_map<N> is_past_the_end;
        cphstl::heap::is_tail_map<N> is_tail;
        cphstl::heap::north_map<N> north;
        cphstl::heap::south_map<N> south;
        if (is_past_the_end[p]) {
          return p;
        }
        p = east[p];
        if (is_tail[p]) {
          return south[east[north[east[p]]]];
        }
        return p;
      }
    };

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

#define USE_HEAP_PROPERTY_MAPS(V, N) \
  cphstl::heap::east_map<N > east; \
  cphstl::heap::is_nil_map<N > is_nil; \
  cphstl::heap::is_past_the_end_map<N > is_past_the_end; \
  cphstl::heap::is_root_map<N > is_root; \
  cphstl::heap::is_tail_map<N > is_tail; \
  cphstl::heap::left_map<N > left; \
  cphstl::heap::north_map<N > north; \
  cphstl::heap::right_map<N > right; \
  cphstl::heap::south_map<N > south; \
  cphstl::heap::value_map<V, N > value; \
  cphstl::heap::west_map<N > west; \
  cphstl::heap::ignore_unused_variable_warning(east); \
```

```
  cphstl::heap::ignore_unused_variable_warning(is_nil); \
  cphstl::heap::ignore_unused_variable_warning(is_past_the_end); \
  cphstl::heap::ignore_unused_variable_warning(is_root); \
  cphstl::heap::ignore_unused_variable_warning(is_tail); \
  cphstl::heap::ignore_unused_variable_warning(left); \
  cphstl::heap::ignore_unused_variable_warning(north); \
  cphstl::heap::ignore_unused_variable_warning(right); \
  cphstl::heap::ignore_unused_variable_warning(south); \
  cphstl::heap::ignore_unused_variable_warning(value); \
  cphstl::heap::ignore_unused_variable_warning(west);

  } // namespace heap
}// namespace cphstl

#endif // __CPHSTL_HEAP_PROPERTY_MAP__
```

F. Bidirectional iterators

F.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 <cstddef> // defines std::ptrdiff_t

namespace cphstl {

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

    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;
    };

    typedef typename Value::range_type V;
    typedef typename Value::domain_type D;
    typedef typename 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_) {
    }

    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
```

```
#if defined(UNITTEST_BIDIRECTIONAL_ITERATOR)
#define __main__ main
#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
    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;
```

```
}

#undef __main__
#endif // UNITTEST_BIDIRECTIONAL_ITERATOR
#endif // __CPHSTL_BIDIRECTIONAL_ITERATOR__
```

G. Skew binary numbers

G.1 Program/skew_binary_number.h++

```
/*
Author: Jyrki Katajainen, May 2006

In skew-binary numbers, digits may be zero, one, or two; not only zero
or one as in normal binary numbers.  The weight of the ith digit is
2^{i+1}-1.  In canonical form, only the lowest non-zero digit may be
two.

Example: 92 could be written as 002101 (least-significant digit first)
since 7 + 7 + 15 + 63 = 92.

In this realization, the bit position 0 is used to indicate whether
the skew binary number has a two or not. In other bit positions, a bit
indicates whether this particular digit is non-zero or not. These bits
are maintained in a word such that the least-significant bit comes
first.
*/

#ifndef __CPHSTL_SKEW_BINARY_NUMBER__
#define __CPHSTL_SKEW_BINARY_NUMBER__

#include <climits> // defines CHAR_BIT
#include <cstddef> // defines std::size_t
#include <cmath> // defines ilogb
#include <stdexcept> // defines std::out_of_range

extern int ilogb(double) throw();

namespace cphstl {

  template <typename W>
  class skew_binary_number {
  public:
    typedef W size_type; // assumed to be unsigned

    skew_binary_number() : n(0), digits(0) {
    }

    operator size_type() const {
      return n;
    }

    skew_binary_number& operator++() { // only prefix ++ supported
      if (n == ~size_type(0) - 1) {
        throw std::out_of_range("skew_binary_number␣operator++():␣overflow");
      }
      ++n;
      if ((*this).has_a_two()) {
        size_type i = ilogb(digits);
        size_type mask = 1 << i;
        digits = digits & (~mask);
        size_type j = ilogb(digits);
        if ((i - 1) == j) {
          digits = digits | 1;
```

```
      }
      else {
        mask = mask >> 1;
        digits = digits | mask;
        digits = digits & all_ones_except_bit_0;
      }
    }
    else {
      if ((digits & all_zeros_except_bit_k) ≡ 0) {
        digits = digits | all_zeros_except_bit_k;
      }
      else {
        digits = digits | 1;
      }
    }
    return *this;
  }

  skew_binary_number& operator--() {
    if (n ≡ 0) {
      throw std::out_of_range("skew_binary_number␣operator--():␣underflow");
    }
    --n;
    if ((digits & all_zeros_except_bit_k) ≢ 0) {
      if ((*this).has_a_two()) {
        digits = digits & all_ones_except_bit_0;
      }
      else {
        digits = digits & all_ones_except_bit_k;
      }
    }
    else {
      if ((*this).has_a_two()) {
        digits = digits & all_ones_except_bit_0;
      }
      else {
        size_type i = ilogb(digits);
        size_type mask = 1 << i;
        digits = digits & (~mask);
        mask = mask << 1;
        digits = digits | mask;
        digits = digits | 1;
      }
    }
    return *this;
  }

  size_type lowest_non_zero() const { // undefined for n ≡ 0
    size_type i = ilogb(digits);
    return k - i;
  }

  bool has_a_two() const {
    return digits & size_type(1);
  }

private:
  size_type n;
  size_type digits;
  enum { k = sizeof(W) * CHAR_BIT - 1 };
  enum { all_ones_except_bit_0 = ~1 };
  enum { all_zeros_except_bit_k = 1 << k };
  enum { all_ones_except_bit_k = ~all_zeros_except_bit_k };
};
```

```
} // namespace cphstl

#if defined(UNITTEST_SKEW_BINARY_NUMBER)
#define __main__ main
#include <cassert> // defines assert macro

template <typename T>
class unittest_skew_binary_number {
public:

  void operator()() {
    cphstl::skew_binary_number<std::size_t> n;
    assert(n ≡ 0); // covertion operator
    ++n;
    assert(n ≡ 1); // operator++
    assert(n.has_a_two() ≡ false);
    ++n;
    assert(n ≡ 2);
    assert(n.has_a_two() ≡ true);
    assert(n.lowest_non_zero() ≡ 0);
    ++n;
    assert(n ≡ 3);
    assert(n.has_a_two() ≡ false);
    assert(n.lowest_non_zero() ≡ 1);
    ++n;
    assert(n ≡ 4);
    assert(n.has_a_two() ≡ false);
    assert(n.lowest_non_zero() ≡ 0);
    ++n;
    assert(n ≡ 5);
    assert(n.has_a_two() ≡ true);
    assert(n.lowest_non_zero() ≡ 0);
    ++n;
    assert(n ≡ 6);
    assert(n.has_a_two() ≡ true);
    assert(n.lowest_non_zero() ≡ 1);
    ++n;
    assert(n ≡ 7);
    assert(n.has_a_two() ≡ false);
    assert(n.lowest_non_zero() ≡ 2);
    ++n;
    assert(n ≡ 8);
    assert(n.has_a_two() ≡ false);
    assert(n.lowest_non_zero() ≡ 0);
    --n; // operator--
    assert(n ≡ 7);
    assert(n.has_a_two() ≡ false);
    assert(n.lowest_non_zero() ≡ 2);
    --n;
    assert(n ≡ 6);
    assert(n.has_a_two() ≡ true);
    assert(n.lowest_non_zero() ≡ 1);
    ++n;
    assert(n ≡ 7);
    assert(n.has_a_two() ≡ false);
    assert(n.lowest_non_zero() ≡ 2);
  }
};

int __main__(int, char**) {
  unittest_skew_binary_number<unsigned int> s;
  s();
  unittest_skew_binary_number<unsigned char> t;
  t();
  return 0;
```

```
}

#undef __main__
#endif // UNITTEST_SKEW_BINARY_NUMBER
#endif // __CPHSTL_SKEW_BINARY_NUMBER__
```

H. Benz forms

H.1 Benchmark/sort_int_pentium.py

```
import benz
import os

class case(benz.case):
  def __init__(self, n, element, heap, include, directory):
    benz.case.__init__(self)
    self.n = n
#    self.computer = 'cphstl.projektlab.diku.dk'
#    self.connection_protocol = 'ssh'
#    self.transfer_protocol = 'scp'
#    self.compiler = 'g++-3'
    self.compiler = 'g++'
    self.compiler_options.extend(['-O3', '-DN=' + str(n), '-D' + directory])
    home = os.environ['HOME']
    self.include_paths.extend([ \
        home + '/CPHSTL/Report/Project-proposal/' + directory + '/'])
    self.include_files.extend([include])
    self.dual_exists = 0
#    self.constructor_call = \
#      'experiment<' +  element + ', ' + heap + '>(' + 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, heap, include, directory, algotitle,
        log_i, log_k):
    benz.curve_suite.__init__(self)
    self.title = algotitle
    for j in range(log_i, log_k + 1):
      self.add(case(1 << j, element, heap, include, directory))

class plot(benz.plot_suite):
  def __init__(self, element, log_i, log_k):
    benz.plot_suite.__init__(self)
    self.title = 'Efficiency␣of␣priority-queue␣operations:␣sort␣model␣\
(push^n␣pop^n),␣unsigned␣integers'
    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␣samplen␣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, 'std::priority_queue<' + element + '>',
      '<queue>', 'STD', 'std::priority\\\\_queue', log_i, log_k))
    self.add(curve(element, 'binary_heap_ref<' + element + '>',
      'binary_heap_ref.c++', 'Referent_heap', 'referent␣heap', log_i, log_k))
    self.add(curve(element, 'cphstl::meldable_priority_queue<' + element + '>',
      'meldable_priority_queue.h++', 'Program', 'meldable␣heap', log_i, log_k))

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

H.2 Benchmark/experiment.c++

```
#include <vector> // defines std::vector
#include <functional> // defines std::less
#include "input_generation.c++" // defines random_sequence()
#include <cassert> // defines assert macro

template <typename V, typename S>
class experiment {
public:
  experiment(unsigned int n) : n(n) {
    a = std::vector<V>(n, V(n));
    increasing_sequence(a.begin(), a.end());
    less = std::less<V>();
    b = std::vector<V>(n, V());
  }
  void primal() {
    S h;
    volatile unsigned long s = 0;
    while (s < n) {
      h.push(a[s]);
      s = s + 1;
    }
    s = n;
    while (s > 0) {
      --s;
      b[s] = h.top();
      h.pop();
    }
    assert(a == b);
  }
private:
  unsigned long n;
  std::vector<V> a;
  std::vector<V> b;
  std::less<V> less;
};
```

H.3 Benchmark/drive.c++

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

```
#ifndef N
#define N 1000000
#endif

#define ELEMENT unsigned int

#if defined(STD)
#include <queue> // defines std::priority_queue
#define HEAP std::priority_queue<ELEMENT>
#elif defined(Referent_heap)
#include "binary_heap_ref.c++" // defines binary_heap_ref
#define HEAP binary_heap_ref<ELEMENT>
#else
#include "meldable_priority_queue.h++" // chpstl::meldable_priority_queue
#define HEAP cphstl::meldable_priority_queue<ELEMENT>
#endif

int main() {
  experiment<ELEMENT, HEAP> e(N);
  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(N) << std::endl;
  return 0;
}
```

I. UNIX makefiles

I.1 Program/makefile

```
gfilt-dir = $(HOME)/Script/Gfilt
options = -Wall -pedantic -x c++

unittests: iterator node number heap

iterator:
        $(gfilt-dir)/gfilt -DUNITTEST_BIDIRECTIONAL_ITERATOR $(options) bidirectional_iterator.
        ./a.out

node:
        $(gfilt-dir)/gfilt -DUNITTEST_HEAP_NODE $(options) heap_node.h++
        ./a.out

number:
        g++ -DUNITTEST_SKEW_BINARY_NUMBER $(options) skew_binary_number.h++
        ./a.out

heap:
        $(gfilt-dir)/gfilt -DUNITTEST_BINARY_HEAP $(options) binary_heap.h++
        ./a.out

stresstest:
        $(gfilt-dir)/gfilt $(options) stress_test.c++

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

I.2 Benchmark/makefile

```
CXX = $(HOME)/Script/Gfilt/gfilt
options = -Wall -O3 -x c++
incl =  -I $(HOME)/CPHSTL/Tool/Benz/Helperbase
```

```
testdrive: std referent meldable

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

referent:
        $(CXX) $(options) $(incl) -I ../Referent_heap -DReferent_heap drive.c++
        ./a.out

meldable:
        $(CXX) $(options) $(incl) -I ../Program -DProgram drive.c++
        ./a.out

benchmark:
        python sort_int_pentium.py

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

References

[1] British Standards Institute. *The C++ Standard: Incorporating Technical Corrigendum 1*, BS ISO/IEC 14882:2003 (2nd Edition), John Wiley and Sons, Ltd. (2003).

[2] G.S. Brodal. Worst-case efficient priority queues. *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms*, ACM/SIAM (1996), 52–58.

[3] H. Brönnimann, J. Katajainen, and P. Morin. Putting your data structure on a diet. In preparation (2006). [Ask Jyrki for details.]

[4] S. Cho and S. Sahni. Weight-biased leftist trees and modified skip lists. *The ACM Journal of Experimental Algorithms* **3** (1998), article 2.

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

[6] Department of Computing, University of Copenhagen. *The CPH STL*, Website accessible at `http://www.cphstl.dk/` (2000–2005).

[7] J.R. Driscoll, H.N. Gabow, R. Shrairman, and R.E. Tarjan. Relaxed heaps: an alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM* **31** (1988), 1343–1354.

[8] A. Elmasry, C. Jensen, and J. Katajainen. A framework for speeding up priority-queue operations. CPH STL Report **2004-3**, Department of Computing, University of Copenhagen (2004). Available at `http://www.cphstl.dk/`.

[9] A. Elmasry, C. Jensen, and J. Katajainen. Relaxed weak queues: an alternative to run-relaxed heaps. CPH STL Report **2005-2**, Department of Computing, University of Copenhagen (2005). Available at `http://www.cphstl.dk/`.

[10] A. Elmasry, C. Jensen, and J. Katajainen. Two-tier relaxed heaps. In preparation (2006). [Ask Jyrki for details.]

[11] C. Jensen and J. Katajainen. An experimental evaluation of navigation piles. CPH STL Report **2006-3**, Department of Computing, University of Copenhagen (2006). Available at `http://www.cphstl.dk/`.

[12] D.B. Johnson. Priority queues with update and finding minimum spanning trees. *Information Processing Letters* **4**, 53–57.

[13] H. Kaplan, N. Shafrir, and R. E. Tarjan. Meldable heaps and boolean union-find. *Proceedings of 34th Annual ACM Symposium on Theory of Computing*. ACM (2002), 573–582.

[14] H. Kaplan and R.E. Tarjan. New heap data structures. Technical Report TR-597-99. Department of Computer Science, Princeton University (1999).

[15] D.E. Knuth. *Sorting and Searching*, *The Art of Computer Programming* **3**, 2nd Edition, Addison Wesley Longman (1998).

[16] C. Okasaki. Alternatives to two classic data structures. *Proceedings of the 36th Technical Symposium on Computer Science Education*, ACM (2005), 162–165.

[17] J.-R. Sack and T. Strothotte. An algorithm for merging heaps. *Acta Informatica* **22** (1985), 171–186.

[18] J.-R. Sack and T. Strothotte. A characterization of heaps and its applications. *Information and Computation* **86** (1990), 69–86.

[19] T. Strothotte and J.-R. Sack. Heaps in heaps. *Congressus Numerantium* **49** (1985), 223–235.

[20] J. Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM* **21** (1978), 309–315.

[21] J.W.J. Williams. Algorithm 232: Heapsort. *Communications of the ACM* **7** (1964), 347–348.